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

Logic Program Updates

por
João Alexandre Carvalho Pinheiro Leite

Dissertação apresentada na Faculdade de Ciências e Tecnologia
da Universidade Nova de Lisboa para a obtenção do Grau de
Mestre em Engenharia Informática.

Orientador: Professor Doutor Luís Moniz Pereira

Lisboa
1997

to my parents

Acknowledgments

First of all I would like to thank my supervisor Prof. Dr. Luís Moniz Pereira for his great help and experience. He was present at all times, encouraging me when I needed most. Without him, we would still not be able to update programs.

I would also like to thank Dr. José Julio Alferes for his very pertinent comments and all his help.

I'm also very grateful to Professor Teodor Przymusinski and Professor Halina Przymusinska for all their ideas and help.

Vitor: thanks for helping with the meta-interpreter and always putting up with me.

For proof reading parts of this dissertation I would like to thank Dr. Joaquim Aparício.

I would like to acknowledge the Departamento de Informática for the working conditions provided. A special word of gratitude goes to the Secretarial staff for being so friendly and efficient.

This dissertation was partially supported by JNICT-PRAXISXXI BM/437/94.

This work was conceived out within the Mental project PRAXIS XXI 2/2.1/TIT/1593/95, directed by my supervisor.

Last, but definitely not least, I would like to thank my parents, to whom I dedicate this dissertation: Part of it is theirs.

Abstract

The field of theory update has seen some major improvements in what concerns model updating. The update of models is governed by update rules and by inertia applied to the literals not directly affected by the update program. This is important but it is still necessary to tackle as well the updating of programs. Some results have been obtained on the issue of updating a logic program which encodes a set of models, to obtain a new program whose models are the desired updates of the initial models. But here the program only plays the rôle of a means to encode the models.

A logic program encodes much more than a set of models: it encodes the relationships between the elements of those models. In this dissertation we advocate that the principle of inertia is advantageously applied to the rules of the initial program rather than to the individual literals in a model. Indeed, we show how this concept of program update generalizes simple interpretation update. We will consider both the 2-valued and 3-valued cases, for normal as well as for logic programs extended with explicit negation.

Sumário

A área de actualização de teorias tem vindo a conhecer grandes progressos no que respeita à actualização dos seus modelos. A actualização de modelos é governada por regras de actualização bem como por regras que codificam a lei da inércia, aplicadas aos literais não afectados directamente pelo programa de actualização. Isto é importante mas é ainda necessário abordar a questão da actualização de programas em lógica. Existem alguns resultados no que respeita à actualização de programas em lógica que codificam um conjunto de modelos, por forma a obter um novo programa cujos modelos são as actualizações dos modelos iniciais. Mas neste caso, o programa em lógica serve apenas de meio para codificar o conjunto de modelos.

Um programa em lógica codifica muito mais do que um conjunto de modelos: codifica também as relações entre os elementos desses modelos. Nesta dissertação, defendemos que a lei da inércia, em vez de ser aplicada aos literais de um modelo, deve ser aplicada às regras do programa em lógica inicial. Assim demonstramos como este conceito de actualização de programas generaliza o conceito de actualização de interpretações. Serão considerados os casos de semanticas a 2 e 3 valores, aplicados a programas em lógica normais bem como extendidos com a negação explícita.

Contents

Chapter 1

Introduction and Motivation

1.1 Introduction

When dealing with modifications to a knowledge base represented by a propositional theory, two kinds of abstract frameworks have been distinguished both by Keller and Winslett in [22] and by Katsuno and Mendelzon in [21]. One, theory revision, deals with incorporating new knowledge about a static world. The other deals with changing worlds, and is known as theory update. For insight on the subject of theory revision, the reader is referred to [1][3][4][36][37]. In this work, we are concerned with theory update only, and, in particular, within the framework of extended logic programming [4].

A key insight into the issue of updating theories is due to Winslett [35], who showed that one must consider the effect of an update in each of the states of the world that are consistent with our current knowledge of its state. Following this approach, most of the work done concerns the updating of models on a one by one basis.

In a series of papers, [25][26][27], Marek and Truszczyński introduced revision programs, a logic-based framework for specifying interpretation updates, and showed how logic programs with stable semantics are embeddable into that framework. So that no confusion arises between updates and revisions, we will dub Marek and Truszczyński's revision programs of *update programs*. In [34], Przymusinski and Turner show how, conversely, there is a simple embedding of update programs into logic programs with stable semantics. They go on by showing how one can further extend the scope of updates by allowing an update to be specified by an arbitrary rule. They also show how this can be embedded into default logic.

As we've mentioned before, all of this work was done with the goal of updating individual models. The common intuition behind the update of a model has been based on what is referred to as the commonsense law of inertia, i.e. things do not change unless they are expressly made to, in this case that the truth value of each element of the model to be updated should remain the same unless explicitly changed by the update. Suppose for example that we have a model in which "sunshine" is true and "raining" is false; if later we receive the information that the sun is no longer shining, we conclude that "sunshine" is false due to the update and that "rain" is still false by inertia.

The first work, to our knowledge, on updating a logic program was done by Alferes and Pereira. In [5] they show how a logic program which encodes a set of initial models can be transformed, following the specification of some update program, such that its new models become the desired updates of the initial models. This transformation works as intended if we want to update the models, and the program serves just as a means to encode them.

1.2 Motivation

Suppose that our vision of the world is described by a logic program and we want to update it. Is updating each of its models enough? Is all the information borne by a logic program contained within its set of models? The answer to both these questions is negative. A logic program encodes more than the set of its individual models. It encodes the relationships between the elements of a model, which are lost if we envisage updates simply on a model by model basis, as proposed in [21].

To show that a logic program encodes more than its set of models, consider the following situation where an alarm signal is present:

Example 1.2.1 *Take the normal program P and its model M:*

$$\begin{aligned} P: \quad & sleep \leftarrow not\ alarm \\ & panic \leftarrow alarm \\ & alarm \leftarrow \end{aligned}$$

$$M = \{alarm, panic\}$$

Now consider an update program [25] stating that the alarm goes off:

$$U: \quad out(alarm) \leftarrow$$

According to [25] and model updating we obtain as the single justified update of M the following model:

$$M_U = \{panic\}$$

Stating that, although we know that the alarm is off, we are in a state of panic and not asleep. But looking at the program and at the update program, we arguably conclude that M_U *doesn't represent the intended meaning of the update of* P *by* U *for a commonsensical reasoner. Since "panic" was true because the "alarm" was on, the removal of "alarm" should make one expect "panic" false. The same kind of reasoner expects "sleep" to become true. The intended update model of the example presumably is:*

$$M_U^{'} = \{sleep\}$$

Another symptomatic example, but using explicit negation is this:

Example 1.2.2 *Given the statements about someone's state of mind on the existence of God:*

- *If I've seen something that is unexplainable then I've seen a miracle.*
- *If I've seen a miracle then God exists.*
- *I've seen something.*
- *It is not explainable.*

They can be represented by the following extended logic program:

$$\begin{array}{ll} P: & seen\_miracle \leftarrow seen\_something, not\ explainable \\ & god\_exists \leftarrow seen\_miracle \\ & seen\_something \leftarrow \\ & \neg explainable \leftarrow \end{array}$$

whose answer-set [19] M *is:*

$$M = \{seen\_something, \neg explainable, seen\_miracle, god\_exists\}$$

Now consider the following update program U *stating that we now have an explanation:*

$$U: \ \ in(explainable) \leftarrow$$

According to model updating we obtain as the single justified update of M *the following model* M_U*:*

$$M_U = \{seen\_something, explainable, seen\_miracle, god\_exists\}$$

Stating that, although there is an explanation for what was seen, it is still believed that it was a miracle and that God exists. Again, looking at the program and at the update program, we arguably conclude that M_U *doesn't represent the intended meaning of the update of* P *by* U *for a commonsensical reasoner. Since "god_exists" was true because "seen_miracle" was true, which in turn depended on the falsity of "explainable", obtaining the explanation, i.e. "explainable" becoming true, should make one expect "seen_miracle" and "god_exists" to become false. The intended update model of the example presumably is:*

$$M_U = \{seen\_something, explainable, not\ seen\_miracle, not\ god\_exists\}$$

Why are we obtaining these somewhat unintuitive results? To answer this question we must first consider the role of inertia in updates.

Newton's first law, also known as the law of inertia, states that: *"every body remains at rest or moves with constant velocity in a straight line, unless it is compelled to change that state by an unbalanced force acting upon it"* (adapted from [29])(although this concept of a straight line is somewhat strange in face of Einstein's theories). One often tends to interpret this law in a commonsensical way, as things keeping as they are unless some kind of force is applied to them. This is true but it doesn't exhaust the meaning of the law. It is the result of all applied forces that governs the outcome. Take a body to which several forces are applied, and which is in a state of equilibrium due to those forces canceling out. Later one of those forces is removed and the body starts to move.

The same kind of behaviour presents itself when updating programs. Before obtaining the truth value, by inertia, of those elements not directly affected by the update program, one should verify whether the truth of such elements is not indirectly affected by the updating of other elements.

Going back to the alarm example, before stating that "*panic*" is true by inertia, since it wasn't directly affected by the update program, we should verify that since the alarm is no longer ringing, there is no way to prove "*panic*" and therefore its truth value should no longer be 'true'. A similar reasoning should be carried out to check the truth value of "*sleep*".

In the example about the existence of God, as in the alarm example, before stating that "*god_exists*" is true by inertia, since it wasn't directly affected by the update program, one should verify for instance whether "*explained*" is still not true, for otherwise there is no longer a way to prove "*god_exists*" and therefore its truth value should no longer be 'true'.

Another way to view program updating, and in particular the role of inertia, is to say that the rules of the initial program carry over to the updated program, due to inertia, instead of the truth of literals, just in case they aren't overruled by the update program. Once again this should be so because the rules encode more information than their models.

To conclude, we argue that the truth of any element in the updated models should be supported by some rule, i.e. one with a true body, originating either in the update program or in the given program.

The structure of the remainder of this work is as follows:

- in chapter 2, for the sake of completeness and self containment, we briefly overview the language and semantics of logic programs;
- in chapter 3 we recapitulate some background concepts related to the update of interpretations;
- in chapter 4 we formalize the normal logic program update process under a stable semantics, present a transformation providing the intended results, and prove some properties;
- in chapter 5 we extend these results to the iterated case, i.e., when we want to update a given program more than once;
- in chapter 6 we extend updates to partial stable semantics;
- in chapter 7 we generalize the results to the case of updating logic programs extended with explicit negation;
- in chapter 8 we illustrate our results with a realistic example;
- in chapter 9 we conclude and provide hints on future developments.

Chapter 2

Semantics of Logic Programs

In this chapter we briefly introduce logic programs and their semantics. Among a great number of semantics proposed for logic programs, the stable semantics and the well founded semantics are the most widely accepted. We will review both for the case of normal logic programs, as well as their counterparts for extended logic programs, the answer-set semantics and the well founded semantics with explicit negation, respectively. For more complete and detailed reviews of semantics of logic programs, their properties as well as their historical background and motivation, the reader is referred to [4][8][12][14][28].

2.1 Normal Logic Programs

This section is devoted to the semantics of normal logic programs, following [28]. We start with a brief description of program and their language. We go on to define interpretations and models, as well as some orderings among them. Finally, we present their stable as well as their well founded semantics.

2.1.1 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 $f(t_1, ..., t_n)$, where f is a function symbol of $\mathcal{A}$, and t_i's are terms. An *atom* over $\mathcal{A}$ is an expression of the form $p(t_1, ..., t_n)$, where p is a predicate symbol of $\mathcal{A}$, and the t_i's are terms. A *literal* over $\mathcal{A}$ is either an atom A or its negation *not* A. We dub default literals those of the form *not* L. 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

$$A_0 \leftarrow B_1, ..., B_m, not\ C_1, ..., not\ C_n \qquad (m, n \geq 0)$$

where A_0, B_i $(0 \leq i \leq m)$ and C_j $(0 \leq j \leq n)$ are atoms.

A normal logic program is called definite if for all rules $n = 0$, i.e. 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. The Herbrand base of P, denoted by $\mathcal{H}(P)$, is the Herbrand base of $\mathcal{A}$.

By grounded version of a normal logic program P we mean the (possibly infinite) set of grounded rules obtained from P by substituting in all possible ways each of the variables in P by elements of its Herbrand universe.

We restrict ourselves to Herbrand interpretations and models, thus dropping the qualification Herbrand.

2.1.2 Interpretations and Models

Next we define 2 and 3-valued interpretations and models of normal logic programs.

Definition 2.1.1 (Two-valued interpretation) *A* two-valued interpretation *I of a normal logic program P is any subset of $\mathcal{H}(P)$.*

Any 2-valued interpretation I can be viewed as a set

$$T \cup not\ F$$

where $T = I$, and $F = \mathcal{H}(P) - T$ are subsets of $\mathcal{H}(P)$. The set T contains all ground atoms which are true in I, and the set F contains all ground atoms which are false in I. The 2-valued designation comes from the fact that $T \cup F = \mathcal{H}(P)$. $\diamondsuit$

Since we must be able to represent incomplete knowledge, i.e express the fact that some atoms are neither true nor false but rather undefined, we need 3-valued interpretations:

Definition 2.1.2 (Three-valued interpretation) *A three-valued interpretation I of a normal logic program P is any set*

$$T \cup not\ F$$

where T and F are disjoint subsets of $\mathcal{H}(P)$.

The set T contains all ground atoms which are true in I, the set F contains all ground atoms which are false in I. The truth value of the remaining atoms is unknown (or undefined). Two-valued interpretations are a special case of 3-valued ones, for which $T \cup F = \mathcal{H}(P)$ is further imposed. $\diamondsuit$

Proposition 2.1.1 *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:*

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

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

Definition 2.1.3 (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:*

-if A is a ground atom then $\hat{I}(A) = I(A)$.

-if S is a formula then $\hat{I}(not\ S) = 1 - I(S)$.

-if S and V are formulae then:

- $\hat{I}((S, V)) = \min(\hat{I}(S), \hat{I}(V))$.
- $\hat{I}(L \leftarrow S) = 1$ *if* $\hat{I}(S) \leq \hat{I}(L)$, *and* 0 *otherwise.* $\diamondsuit$

Definition 2.1.4 (Three-valued model) *A 3-valued interpretation I is called a 3-valued (or partial) model of a program P iff for every ground instance of a program rule $H \leftarrow B$ we have $\hat{I}(H \leftarrow B) = 1$.* $\diamondsuit$

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

Definition 2.1.5 (Two-valued model) *A 2-valued interpretation I is called a 2-valued (or total) model of a program P iff for every ground instance of a program rule $H \leftarrow B$ we have $\hat{I}(H \leftarrow B) = 1$.* $\diamondsuit$

Next we define two orderings among interpretations and models:

Definition 2.1.6 (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.* $\diamondsuit$

Definition 2.1.7 (Fitting ordering) *If I and J are two interpretations then we say that $I \leq_F J$ [15]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.* $\diamondsuit$

The classical ordering is related with the amount of true atoms, whereas the Fitting ordering is related with the amount of information, i.e. non-undefinedness.

2.1.3 Stable model semantics

In [17], the authors introduce the so-called "stable model semantics". 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 [16]. If the consequences completely corroborate the hypotheses made, then they form a stable model. Formally:

Definition 2.1.8 (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 containing a default literal not A such that $A \in I$;*
- *Remove from all remaining rules all default literals.*

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

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 2.1.9 (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 semantics iff A belongs to all stable models of P. $\diamondsuit$

2.1.4 Well-founded semantics

Despite its advantages, stable model semantics has some important drawbacks such as the inexistance of stable models for some programs; the impossibility of determining the semantics based on top-down rewriting techniques, among others. For a more comprehensive description of its drawbacks see [10][11][14][24].

The well-founded semantics was introduced in [20], and overcomes all of the major drawbacks of the stable model semantics. We will use the equivalent definition of [28], for it is technically closer to the definition of the stable model semantics. Indeed, it consists of a natural generalization for 3-valued interpretations of the stable model semantics. In order to formalize the notion of partial stable models, the authors first expand the language of programs with the additional propositional constant $\mathbf{u}$ with the property of being undefined in every interpretation i.e., 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 $\mathbf{u}$. In [28], 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 2.1.10 (Γ^*-operator) *Let P be a normal logic program and I a 3-valued interpretation. The extended 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 containing 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 $\mathbf{u}$;*
- *Remove from all remaining rules all default literals.*

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

Definition 2.1.11 (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. We dub this unique F-least partial stable model the well founded model. $\diamondsuit$

2.2 Extended Logic Programs

Several authors have recently shown the importance of including, beside default negation, a symmetric kind of negation $\neg$ in logic programs. For the motivation for such extension, the reader is referred to [4][6][19][31][32][33].

This section is devoted to the semantics of these logic programs extended with a symmetric kind of negation $\neg$. We begin by a brief description of the language of such programs. We go on to present the answer-set semantics [19] which is the natural extension of the stable semantics to extended logic programs. Finally we present the well founded semantics with explicit negation or WFSX for short [2][4].

2.2.1 Language of extended logic programs

As for normal logic programs, an *atom* over an alphabet $\mathcal{A}$ is an expression of the form $p(t_1, ..., t_n)$, where p is a predicate symbol of $\mathcal{A}$, and the t_i's 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 symmetric negation $\neg A$. We also use the symbol $\neg$ to denote complementary literals in the sense of symmetric 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 $\mathcal{A}$.

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

$$H \leftarrow L_1, ..., L_m \qquad (m \geq 0)$$

where H is an objective literal and each of the L_is is a literal.

As for normal logic programs, 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. The extended Herbrand base of P, denoted by $\mathcal{H}(P)$, is the extended Herbrand base of $\mathcal{A}$.

A set of rules stands for all its ground instances.

2.2.2 Answer-set semantics

Introduced in [18], the answer-set semantics is the first semantics defined for extended programs. It is a generalization of the stable model semantics for the language of such programs. Roughly, an answer-set of an extended logic program P is a stable model of the normal logic program obtained from P by replacing objective literals of the form $\neg L$ by new atoms, say $\neg\_L$. Formally:

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 new atoms, say $\neg\_A$;

-replacing both in P and I, these objective literals by their new denotation;

-then performing the following operations:

- *Remove from P all rules containing a default literal* not A *such that $A \in I$;*
- *Remove from all remaining rules all default literals.*

Since the resulting program $\frac{P}{I}$ 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 $\Gamma(I) = J'$. $\diamondsuit$

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 if $\neg L$ is true; otherwise L is unknown.[1] $\diamondsuit$

2.2.3 Well-founded semantics with explicit negation

Introduced in [30], the well founded semantics with explicit negation is the counterpart to the well founded semantics for the case of extended logic programs. It's major contribution is the so-called coherence principle that, unlike other generalizations of the well founded semantics, provides the missing relation between both negations: if $\neg L$ holds then *not* L should too (similarly, if L then *not* $\neg L$). For a complete description of the WFSX, its properties and applications, the reader is refered to [4].

As for the well-founded semantics, it is necessary to 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 $\mathbf{u}$. The next definition is an extension of the P modulo I transformation of [28], itself an extension of the Gelfond-Lifschitz modulo transformation [17].

[1] In the remainder of this dissertation, we say that an objective literal L is false in an answer-set M if L doesn't belong to M.

Definition 2.2.3 ($\frac{P}{I}$ transformation) *[30]Let P be an extended logic program and let I be a 3-valued 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 in the body 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* **u**. ◊

The resulting program $\frac{P}{I}$ is, by definition, non-negative, and therefore has a least three-valued model $least(\frac{P}{I})$, defined in the following way:

Definition 2.2.4 (Least operator) *[30]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 three-valued model of P'.*
- *$T \cup not\ F$ is obtained from $T' \cup not\ F'$ by reversing the replacements above.* ◊

The least three-valued model of a non-negative program can be defined as the least fixed-point of the following generalization of the van Emden-Kowalski least model operator Ψ for definite logic programs [16]:

Definition 2.2.5 (Ψ^* operator) *[30]Suppose that P is a non-negative program, I a 3-valued interpretation of P and 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 $A \leftarrow A_1, ..., A_n$ in P such that $I(A_i) = 1$ for all $i \leq n$.*
- *$\Psi^*(I)(A) = 0$ iff for every rule $A \leftarrow A_1, ..., A_n$ there is an $i \leq n$ such that $I(A_i) = 0$.*
- *$\Psi^*(I)(A) = \frac{1}{2}$ otherwise.* ◊

Definition 2.2.6 (Three-valued least model) *[30]The three-valued least model of a non-negative program is:*

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

The following theorem guarantees the uniqueness of the least three-valued model.

Theorem 2.2.1 *[30]least(P) uniquely exists for every non-negative program P.* $\diamondsuit$

It is worth noting that $least(\frac{P}{I})$ may violate both non-contradiction and coherence conditions. To impose coherence, when contradiction is not present, a partial operator is defined that transforms any non-contradictory set of literals into an interpretation.

Definition 2.2.7 (The *Coh* operator) *[30]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*

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

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

Next is introduced the generalization of the Γ^* operator of [28].

Definition 2.2.8 (The Φ operator) *[30]Let P be a logic program, I a 3-valued 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. $\diamondsuit$

Definition 2.2.9 (WFSX, PSM and WFM) *[30]A 3-valued interpretation I of an extended logic program P is called a 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. $\diamondsuit$

The following two theorems provide us with important results:

Theorem 2.2.2 (PSMs are models) *[30]Every PSM of a program P is a model of P.* $\diamondsuit$

Theorem 2.2.3 (Generalization of the well-founded semantics) *[30]For programs without explicit negation WFSX coincides with well-founded semantics.* $\diamondsuit$

Chapter 3

Interpretation Updates

In this chapter we recall some basic definitions as well as some important results related to the issue of interpretation updating. For a more detailed motivation and description of this topic the reader is refered to [5][25][26][27][34]. Some of these definitions will be slightly different, though equivalent, from the original ones, for the purpose of making their relationship clearer.

With the purpose of self containment and of eliminating any confusions between updates and revisions, and since we are dealing only with updates, instead of using the original vocabulary of revision rule, revision program and justified revision [25], we will speak of update rule, update program and justified update, as in [5].

We start our overview with the 2-valued case. Then we present the extensions for the 3-valued case, as well as for the update of interpretations extended with explicit negation.

3.1 Two-valued Interpretations

The framework of update programming was first introduced in a series of papers [25][26][27] with the purpose of specifying interpretation update in a language similar to that of logic programming.

Their authors start by defining update programs. They are collections of update rules, which in turn are built out of atoms by means of special operators: $\leftarrow$, *in*, *out* and ",".

Definition 3.1.1 (Update Programs) *[25] Let U be a countable set of atoms. An update in-rule or, simply, an in-rule, is any expression of the form:*

$$in(p) \leftarrow in(q_1), ..., in(q_m), out(s_1), ..., out(s_n) \tag{3.1}$$

where p, q_i, $1 \leq i \leq m$, and s_j, $1 \leq j \leq n$, are all in U, and m, $n \geq 0$.

An update out-rule or, simply, an out-rule, is any expression of the form:

$$out(p) \leftarrow in(q_1), ..., in(q_m), out(s_1), ..., out(s_n) \tag{3.2}$$

where p, q_i, $1 \leq i \leq m$, and s_j, $1 \leq j \leq n$, are all in U, and m, $n \geq 0$.
A collection of in-rules and out-rules is called an update program. $\diamondsuit$

Update programs can syntactically be viewed as logic programs. However they are assigned with a special update semantics. In order to present the semantics we first need the definition of the necessary change determined by an update program.

Definition 3.1.2 (Necessary Change) *[25]Let P be an update program with least model M. The* necessary change *determined by P is the pair (I_P, O_P), where*

$$I_P = \{a : in(a) \in M\}$$
$$O_P = \{a : out(a) \in M\}$$

Intuitively, atoms in I_P are those that must be added and atoms in O_P are those that must be deleted. If $I_P \cap O_P = \{\}$ then P is said coherent. $\diamondsuit$

And the semantics of P-justified update is defined in the following way:

Definition 3.1.3 (P-justified update) *[25]Let P be an update program and I_i and I_u two interpretations. The reduct $P_{I_u|I_i}$ with respect to I_i and I_u is obtained by the following operations:*

1. *Removing from P all rules whose body contains some $in(a)$ and $a \notin I_u$;*
2. *Removing from P all rules whose body contains some $out(a)$ and $a \in I_u$;*
3. *Removing from the body of any remaining rules of P all $in(a)$ such that $a \in I_i$;*
4. *Removing from the body of any remaining rules of P all $out(a)$ such that $a \notin I_i$.*

Let (I, O) be the necessary change determined by $P_{I_u|I_i}$. Whenever P is coherent, I_u is a P-justified update *of I_i wrt P if the following stability condition holds:*

$$I_u = (I_i - O) \cup I$$

We will denote it by:

$$I_u = I_i \oplus P \ \diamondsuit$$

Intuitively, the first two operations delete rules which are useless given I_u. The stability condition preserves the initial interpretation in the final one as much as possible. It is therefore a way to impose the inertia law.

Example 3.1.1 *Consider the update program P and the initial interpretation I_i:*

$$\begin{array}{lll} P: & in(a) \leftarrow out(b) & \quad I_i = \{b\} \\ & out(b) \leftarrow & \\ & in(c) \leftarrow out(a) & \end{array}$$

Intuitively, the second rule will delete **b** *from I_i and as a consequence,* **a** *is inserted via the first rule. Since* **a** *was inserted, the third rule should have no effect. Thus, the P-justified update should be:*

$$I_u = \{a\}$$

Indeed, if we determine the reduct $P_{I_u|I_i}$, we obtain:

$$\begin{array}{ll} P_{I_u|I_i}: & in(a) \leftarrow out(b) \\ & out(b) \leftarrow \end{array}$$

and the necessary change is:

$$I = \{a\} \qquad O = \{b\}$$

since

$$(I_i - O) \cup I = (\{b\} - \{b\}) \cup \{a\} = I_u$$

I_u is a P-justified update. Any other interpretation would violate the stability condition, and so I_u is the single P-justified update of I_i.

An alternative way to characterize P-justified revisions, set forth in [25], is to choose, among all interpretations that are models of the update program, those that are closer to the initial interpretation. To formalize this characterization, they first introduce the notion of model of an update program.

Definition 3.1.4 (Model of an Update Program) *[25]Let I be an interpretation. I is a model of $in(a)$ (resp. $out(a)$) if $a \in I$ (resp. $a \notin I$). I is a model of the body of a rule if it is a model of each $in(\_)$ and each $out(\_)$ of the body. I is a model of a rule c if the following condition holds: whenever I is a model of the body of c, then I is a model of the head of c. I is a* model of an update program *P if it is a model of each rule of P.* $\diamondsuit$

Theorem 3.1.1 (Alternative Characterization of P-justified updates) *[25]Let P be an update program, with initial interpretation I_i. An interpretation I_u is a P-justified update of I_i if and only if:*

1. *I_u is a model of P;*
2. *no other model of P is closer to I_i.* $\diamondsuit$

Where the closeness relationship is defined as follows:

Definition 3.1.5 (Closeness relationship) *[35]Given the interpretations I, I' and I'', we say that I' is closer to I than I'' is if*

$$(I' \setminus I \cup I \setminus I') \subset (I'' \setminus I \cup I \setminus I'') \diamondsuit$$

Example 3.1.2 *Considering the P and I_i from example 3.1.1, we have two interpretations that are models of P, namely:*

$$I_1 = \{a\} \qquad I_2 = \{a, c\}$$

Since $(I_1 \setminus I \cup I \setminus I_1) \subset (I_2 \setminus I \cup I \setminus I_2)$, we have that I_1 is, as expected, the only P-justified update.

The similarity of the P-justified updates to the stable models semantics for logic programs led to an embedding of the later in the former. For that, Marek and Truszczyński start by proposing a transformation from logic program clauses to update rules such that:

Definition 3.1.6 (Translation of logic programs into update programs) *[25] Given a logic program clause* c

$$p \leftarrow q_1, ..., q_m,\ not\ s_1, ...,\ not\ s_n$$

its corresponding update rule up(c) *is:*

$$in(p) \leftarrow in(q_1), ..., in(q_m),\ out(s_1), ..., out(s_n)$$

Given a logic program P, the corresponding revision program rp(P) *is*

$$rp(P) = \{rp(c) : c \in P\}\ \diamondsuit$$

With this translation, they prove the following theorem:

Theorem 3.1.2 (Embedding of logic programs in update programs) *[25]Let P be a logic program. An interpretation M is a stable model of P if and only if M is a $rp(P)$-justified revision of M_0, where $M_0 = \emptyset$.* $\diamondsuit$

In [34], Przymusinski and Turner show that, conversely, there is a simple embedding of update programs into logic programs with stable semantics, and that the two formalisms are equivalent. This embedding provides us with an easy tool to demonstrate several properties of update programming, known from logic programming. It uses extended (with strong negation) logic programs under the stable semantics [17]. For this, the update program is first translated into an extended logic program and some rules are added to code the initial interpretation as well as to reflect the commonsense law of inertia. The following definition differs from the original one in what the kind of negation used is concerned: strong negation has been replaced by explicit negation because they are equivalent for programs under stable semantics (for a detailed study of strong and explicit negation the reader is refered to [6]).

Definition 3.1.7 (Translating updates into extended logic programming) *[34] Let P be an update program, with initial interpretation I_i. Let $\pi(P)$ be the positive extended logic program obtained from P by replacing each in-rule 3.1 in P with the corresponding rule*

$$p \leftarrow q_1, ..., q_m, \neg s_1, ..., \neg s_n$$

and similarly replacing each out-rule 3.2 in P with the corresponding rule

$$\neg p \leftarrow q_1, ..., q_m, \ \neg s_1, ..., \ \neg s_n$$

Let $P^*(P, I_i)$ be the extended logic program that is obtained by augmenting $\pi(P)$ with the rule

$$a \leftarrow not \ \ \neg a$$

for each atom $a \in I_i$ and the rule

$$\neg a \leftarrow not \ a$$

for each atom $a \notin I_i$. $\diamondsuit$

The following theorem establishes the relationship with the semantics of P-justified updates.

Theorem 3.1.3 (Embedding updates in extended logic programming) *[34]Let P be an update program, with initial interpretation I_i. An interpretation I_u is a P-justified update of I_i if and only if I_u is an extended stable model of $P^*(P, I_i)$. Moreover, every extended stable model of $P^*(P, I_i)$ is a P-justified update of I_i.* $\diamondsuit$

Example 3.1.3 *Applying the previous translation to the P and I_i from example 3.1.1, we obtain:*

$$
\begin{array}{lll}
P^*(P, I_i): & a \leftarrow \neg b & \neg a \leftarrow not\ a \\
& \neg b \leftarrow & b \leftarrow not\ \ \neg b \\
& c \leftarrow \neg a & \neg c \leftarrow not\ c
\end{array}
$$

whose only extended stable model is:

$$I_u = \{a, \neg b, \neg c\}$$

which (modulo strongly negated elements) is the only P-justified update.

For a more detailed study of the relationship between update programs and logic programs under the stable semantics, an extension to allow for updates to be specified by means arbitrary inference rules, as well as an embedding into default logic, the reader is refered to [34].

3.2 Three-valued Interpretations

Alferes and Pereira [5] generalized interpretation updates to partial interpretations.

Definition 3.2.1 (Three-valued necessary change) *[5] Let P be an update program with least three-valued model $M = \langle T_M, F_M \rangle$. The* three-valued necessary change *determined by P is the tuple (I_P, O_P, NI_P, NO_P), where*

$$
\begin{aligned}
I_P &= \{a : in(a) \in T_M\} \\
O_P &= \{a : out(a) \in T_M\} \\
NI_P &= \{a : out(a) \notin F_M\} \\
NO_P &= \{a : in(a) \notin F_M\}
\end{aligned}
$$

Atoms in I_P (resp. O_P) are those that must become true (resp. false). Atoms in NI_P (resp. NO_P) are those that cannot remain true (resp. false). If $I_P \cap O_P = \{\}$ then P is said coherent. $\diamondsuit$

And the semantics of P-justified update is defined in the following way:

Definition 3.2.2 (Three-valued P-justified update) *[5] Let P be an update program and $I_i = \langle T_i, F_i \rangle$ and $I_u = \langle T_u, F_u \rangle$ two partial (or three-valued) interpretations. The reduct $P_{I_u|I_i}$ with respect to I_i and I_u is obtained by the following operations, where u is a reserved atom undefined in every interpretation:*

1. *Removing from P all rules whose body contains some $in(a)$ and $a \in F_u$;*
2. *Removing from P all rules whose body contains some $out(a)$ and $a \in T_u$;*
3. *Substituting in the body of any remaining rules of P the reserved atom* **u** *for every $in(a)$, such that a is undefined in I_u and $a \in T_i$;*
4. *Substituting in the body of any remaining rules of P the reserved atom* **u** *for every $out(a)$, such that a is undefined in I_u and $a \in F_i$;*
5. *Removing from the body of any remaining rules of P all $in(a)$ such that $a \in T_i$;*
6. *Removing from the body of any remaining rules of P all $out(a)$ such that $a \in F_i$.*

Let (I, O, NI, NO) be the three-valued necessary change determined by $P_{I_u|I_i}$. Whenever P is coherent, I_u is a three-valued P-justified update *of I_i wrt P if the two stability conditions hold:*

$$T_u = (T_i - NI) \cup I \quad and \quad F_u = (F_i - NO) \cup O$$

We will denote it by:

$$I_u = I_i \oplus P \ \diamondsuit$$

Example 3.2.1 *Consider the update program P and the initial interpretation $I_i = \langle T_i, F_i \rangle$:*

$$\begin{array}{rll} P: & in(a) \leftarrow out(b) & \quad I_i = \langle \{\}, \{a, b, c\} \rangle \\ & in(b) \leftarrow out(a) & \\ & in(c) \leftarrow out(c) & \end{array}$$

Intuitively, the third rule will make **c** *undefined, while the first 2 two rules will produce three possible results where* **a** *(resp.* **b***) is true (resp. false), false (resp. true), or undefined (resp. undefined). Thus, the P-justified updates should be:*

$$I_{u1} = \langle \{a\}, \{b\} \rangle \qquad I_{u2} = \langle \{b\}, \{a\} \rangle \qquad I_{u3} = \langle \{\}, \{\} \rangle$$

Indeed, if we determine the reducts $P_{I_{un}|I_i}$, *we obtain:*

$$P_{I_{u1}|I_i}: \begin{array}{l} in(a) \leftarrow \\ in(c) \leftarrow \mathbf{u} \end{array} \qquad P_{I_{u2}|I_i}: \begin{array}{l} in(b) \leftarrow \\ in(c) \leftarrow \mathbf{u} \end{array} \qquad P_{I_{u3}|I_i}: \begin{array}{l} in(a) \leftarrow \mathbf{u} \\ in(b) \leftarrow \mathbf{u} \\ in(c) \leftarrow \mathbf{u} \end{array}$$

and the necessary changes are:

$$\begin{array}{lll} I_{P1} = \{a\} & I_{P2} = \{b\} & I_{P3} = \{\} \\ O_{P1} = \{\} & O_{P2} = \{\} & O_{P3} = \{\} \\ NI_{P1} = \{\} & NI_{P2} = \{\} & NI_{P3} = \{\} \\ NO_{P1} = \{a, c\} & NO_{P2} = \{b, c\} & NO_{P3} = \{a, b, c\} \end{array}$$

since for all three cases we have

$$T_{un} = (T_i - NI) \cup I \quad \text{and} \quad F_{un} = (F_i - NO) \cup O$$

with $n = 1, 2, 3$, *all* I_{un} *are three-valued P-justified updates. Any other interpretation would violate the stability conditions.*

The following theorem ensures us that the two-valued definitions of [26] are a special case of the definitions above.

Theorem 3.2.1 (Three-valued generalization of updates) *[5] The three-valued versions of necessary change and P-justified update both reduce to their two-valued versions [26] whenever the initial and final interpretations are total.* $\diamondsuit$

In [5] the authors also propose a program transformation that produces a new program whose models are exactly the justified revisions of the models of the initial program, according to the above definition. Note that here the program only plays the rôle of a means to encode the models.

Definition 3.2.3 (Update transformation of a normal program) *[5]Consider an update program* U *and its corresponding* $\pi(U)$. *For any normal logic program* P, *its updated program* P_U *with respect to* $\pi(U)$ *is obtained through the operations:*

·All rules of $\pi(U)$ *belong to* P_U;

·All rules of P *belong to* P_U, *subject to these changes:*

For each atom A *figuring in the head of a rule of* $\pi(U)$:

- replace in every rule of P_U *originated in* P *all occurences of* A *by* A' *where* A' *is a new atom;*

- include in P_U the rules:

$$A \leftarrow A', not\ \neg A \quad and \quad \neg A \leftarrow not\ A', not\ A \quad \diamondsuit$$

The following theorem establishes the relation between the models of the resulting program and the P-justified updates of the models of the initial program:

Theorem 3.2.2 (Correctness of the update transformation) *[5]Let P be a normal logic program and U a coherent update program. Modulo any primed and explicitly negated elements and their defaults, the $WFSX$ models of the updated program P_U of P with respect to $\pi(U)$ are exactly the three-valued U-Justified Updates of the WFS models of P.* $\diamondsuit$

Example 3.2.2 *Consider the normal logic program P with well-founded model M:*

$$\begin{array}{lll} P: & a \leftarrow not\ b & M = \langle \{a,d,e\}, \{b,c\} \rangle \\ & d \leftarrow e & \\ & e \leftarrow & \end{array}$$

now consider the update program U and its corresponding logic program $\pi(U)$:

$$\begin{array}{llll} U: & in(c) \leftarrow out(a) & \pi(U): & c \leftarrow \neg\ a \\ & in(b) \leftarrow & & b \leftarrow \\ & out(e) \leftarrow in(a) & & \neg e \leftarrow a \end{array}$$

And the updated program P_U is (where the rules for A stand for all their ground instances):

$$\begin{array}{lll} c \leftarrow \neg a & a \leftarrow not\ b' & A \leftarrow A', not\ \neg A \\ b \leftarrow & d \leftarrow e' & \neg A \leftarrow not\ A', not\ A \\ \neg e \leftarrow a & e' \leftarrow & \end{array}$$

whose only WFSX model (modulo A', and explicitly negated atoms) is:

$$M_U = \langle \{a,b,d\}, \{c,e\} \rangle$$

Indeed, M_U is the only U-justified update of M, according to Def.3.2.2.

Note that, as argued in the introduction of this dissertation, M_U doesn't represent the intended result of the update of P by U. Indeed, the insertion of **b** renders **a** no longer supported (there aren't any rules (from P or U) for **a** with a true body) and thus should be false; since **a** should be false, **c** should become true due to the first rule of the update program; the last rule of U should be ineffective since **a** should be false; since **e** should no longer be updated, and is still supported, should remain true by inertia; finally **d** should remains true because it is not updated and is still supported by **e**. The expected result should be:

$$M_U = \langle \{b, c, d, e\}, \{a\} \rangle$$

3.3 Extended Interpretations

In [5] the issue of updating interpretations extended with explicit negation was also addressed. The authors start by defining update programs for this new extended language.

Definition 3.3.1 (Update rules for objective literals) *[5]Let K be a countable set of objective literals. Update in-rules or, simply, in-rules, and update out-rules or, simply, out-rules, are as per Definition 3.1.1, but with respect to this new set K.* $\diamondsuit$

Alferes and Pereira show that, much in the same way as extended logic programs, updates over this extended language should comply with coherence: if due to some update we have $in(\neg A)$, we should aditionally have $out(A)$. For simplicity, instead of changing the definition of justified update, they impose coherence by explicitly adding, to every update program, the following two coherence rules:

$$out(A) \leftarrow in(\neg A) \quad \text{and} \quad out(\neg A) \leftarrow in(A)$$

for every atom A in the language, and then determine the justified updates as before.

Definition 3.3.2 (Extended P-justified Update) *[5]Let P be an update program with explicit negation, and $I_i = \langle T_i, F_i \rangle$ and $I_u = \langle T_u, F_u \rangle$ two partial interpretations with explicit negation.*

Let $P' = P \cup \{out(L) \leftarrow in(\neg L) : L \in K\}$, where K is the set of all objective literals. I_u is an extended P-justified update (or P-justified update for short) of I_i iff it is a P'-justified update according to Definition 3.2.2, where all explicitly negated atoms are simply viewed as new atoms. $\diamondsuit$

As for normal update programs, they provide a translation of these extended update programs into extended logic programs. The translation is slightly more complex for now we can no longer translate $out(L)$ into $\neg L$. This is so because if we simply translate them as in Def.3.1.7 we wouldn't be able to differentiate between having $in(A)$ and $out(\neg A)$ for both would translate into A (since $A = \neg\neg A$). The intuition behind the following translation is to convert an atom A into a new atom A^p and an explicitly negated atom

$\neg A$ into a new atom A^n and ensure coherence. This way, we no longer have explicitly negated atoms in the heads of the rules of update programs and so we can use explicit negation $\neg L$ to code the $out(L)$ in the heads of rules. Then we have to map the A^n and A^p back to their original atoms. Formally we have:

Definition 3.3.3 (Translation of extended UPs into ELPs) *[5]Given an update program with explicit negation UP, its* translation into the extended logic program U *is defined as follows*[1]*:*

1. *Each in-rule*

$$in(L_0) \leftarrow in(L_1), ..., in(L_m), out(L_{m+1}), ..., out(L_n)$$

where m, $n \geq 0$, and L_i are objective literals, translates into:

$$L_0^* \leftarrow L_1, ..., L_m, not\ L_{m+1}, ..., not\ L_n$$

where $L_0^* = A^p$ if $L_0 = A$, or $L_0^* = A^n$ if $L_0 = \neg A$;

2. *Each out-rule*

$$out(L_0) \leftarrow in(L_1), ..., in(L_m), out(L_{m+1}), ..., out(L_n)$$

where m, $n \geq 0$, and L_i are objective literals, translates into:

$$\neg L_0^* \leftarrow L_1, ..., L_m, not\ L_{m+1}, ..., not\ L_n$$

where $L_0^* = A^p$ if $L_0 = A$, or $L_0^* = A^n$ if $L_0 = \neg A$;

3. *For every objective literal L such that $in(L)$ belongs to the head of some in-rule of UP, U contains $\neg L^* \leftarrow L$ where $L^* = A^n$ if $L = A$, or $L^* = A^p$ if $L = \neg A$;*

4. *For every atom A, U contains the rules $A \leftarrow A^p$ and $\neg A \leftarrow A^n$.* $\diamondsuit$

As for normal logic programs, they provide a transformation that, given an extended logic program coding a set of interpretations and an update program with explicit negation, produces an extended logic program whose WFSX models are the extended justified updates of the models of the initial program.

[1]This translation employs the results in [13], namely the expressive power of WFSX to capture the semantics of extended logic programs with default literals in the heads of rules, via the program transformation P^{not} (see pp.99).

Definition 3.3.4 (Update transformation of an extended program) *[5]Given an update program with explicit negation UP and its corresponding U. For any extended program P, its updated program P_U with respect to U (or UP) is obtained through the operations:*

·All rules of U belong to P_U;

·All rules of P belong to P_U, subject to these changes:

For each atom A figuring in the head of a rule of UP:

- replace in every rule of P_U originated in P all occurences of A by A' where A' is a new atom;

- include in P_U the rules:

$$\begin{array}{ll} A^p \leftarrow A', not\ \neg A^p & A^n \leftarrow \neg A', not\ \neg A^n \\ \neg A^p \leftarrow not\ A', not\ A^p & \neg A^n \leftarrow not\ \neg A', not\ A^n\ \diamondsuit \end{array}$$

Chapter 4

Updating Normal Logic Programs under Stable Semantics

4.1 Introduction

As we've seen in the introduction, updating on the basis of models isn't enough if we want to take advantage of the information encoded by a logic program not expressed in the set of its models. Let's have a look at another example:

Example 4.1.1 *Consider program P stating that the switch is on, and if the switch is on then the light is on. Later on, an update U informs that the switch has been turned off:*

$$P: \quad switch\_on \leftarrow \qquad\qquad U: \quad out(switch\_on) \leftarrow$$
$$light\_on \leftarrow switch\_on$$

the only stable model of P is

$$M = \{switch\_on, light\_on\}$$

According to model updating and program updating in [5] we obtain as the single justified update of M the following model:

$$M_U = \{light\_on\}$$

stating that, although the switch isn't on, the light is still on. Looking at the program and the update, we argue that any commonsensical reasoner expects the light to be off, and thus obtain the following model as the only update:

$$M_U' = \{\}$$

When we generalize the notion of p-justified update, from interpretations to the new case where we want to actually update programs, the resulting update program should be made to depend only on the initial program and on the update program, but not on any specific initial interpretation. An interpretation should be a model of a normal logic program updated by an update program if the truth of each of its literals is either supported by a rule of the update program with true body in the interpretation or, in case there isn't one, by a rule of the initial program whose conclusion is not contravened by the update program.

Another way to view program updating, and in particular the rôle of inertia, is to say that the rules of the initial program carry over to the updated program, due to inertia, instead of the truth of interpretation literals like in [5], just in case they are not overruled by the update program. This is to be preferred because the rules encode more information than the literals. Inertia of literals is a special case of rule inertia since literals can be coded as factual rules. Accordingly, program updating generalizes model updating.

To achieve rule inertia we start by defining the sub program of the initial program which contains the rules that should persist in the updated program due to inertia. We use this program together with the update program to characterize the models of the resulting updated program, i.e. the program-justified updates. Afterwards, we present a joint program transformation of the initial and the update programs, which introduces also inertia rules, to produce an updated program whose models are the required program-justified updates. Stable model semantics and its answer-sets generalization to extended logic programs [18] will be used to define the models of programs.

4.2 Preliminary Definitions

As will be seen, in order to update normal logic programs over a language $\mathcal{L}$, we have to extend it with explicit negation $\neg$. The following definition relates interpretations over this extended language with interpretations over $\mathcal{L}$.

Definition 4.2.1 (Interpretation Restriction) *Given a language $\mathcal{L}$ that does not contain explicit negation $\neg$, let $M_\neg$ be an interpretation, over the language $\mathcal{L}_\neg$, obtained by augmenting $\mathcal{L}$ with the set $E = \{\neg A : A \in \mathcal{L}\}$.*

We define the corresponding restricted interpretation *M, over $\mathcal{L}$, as:*

$$M = \{A : A \in M_\neg \text{ and } A \in \mathcal{L}\} \ \diamondsuit$$

We will also make use of the following operations over sets:

Definition 4.2.2 *Let A and B be two sets. Then:*

$$\begin{aligned} A + B &= \{x : x \in A \text{ or } x \in B\} \\ A - B &= \{x : x \in A \text{ and } x \notin B\} \end{aligned}$$

Note that $A + B = A \cup B$. ♢

Let us start by translating update programs as in Def.3.1.1 into extended logic programs. Following the reverse of Def.3.1.6, and using explicit negation to code $out(\_)$ in the head of update rules, we define:

Definition 4.2.3 (Translation of UPs into LPs) *Given an update program UP, over a language $\mathcal{L}$, its* translation into an extended logic program *U over $\mathcal{L}_\neg$ is obtained from UP by replacing each in-rule (3.1) with the corresponding rule:*

$$p \leftarrow q_1, ...q_m, not\ s_1, ..., not\ s_n$$

and similarly replacing each out-rule (3.2) with the corresponding rule:

$$\neg p \leftarrow q_1, ...q_m, not\ s_1, ..., not\ s_n \ \diamondsuit$$

From now onwards, and unless otherwise stated, whenever we refer to an update program we mean its reversible translation into an extended logic program according to the previous definition. Notice that such programs do not contain explicitly negated atoms in the body of its rules.

Also, for the sake of generalization, we will allow for the initial program to be of the same form of the translated update program into a logic program. Although they are extended logic programs defined over the extended language $\mathcal{L}_\neg$, the use of explicit negation is limited for we are only concerned with the models over the normal language $\mathcal{L}$, i.e. a language without explicit negation. For this reason, we will dub these programs normal logic programs. Note that any answer-set of such program over $\mathcal{L}_\neg$, when restricted to $\mathcal{L}$, is a stable model of the program consisting only of the rules for positive atoms, i.e., the only effect of introducing such rules is the possible introduction of contradiction. Since dealing with contradictory programs is beyond the scope of this work, we will only consider non-contradictory programs.

4.3 <P,U>-Justified Updates

In this section we will define and characterize the so called <P,U>-justified updates, i.e., the set of interpretations that are accepted as characterizing the update of a normal logic program P by an update program U. To achieve this, we start by defining the sub-program of the initial program which contains the rules that should persist in the updated program due to inertia. We use this program together with the update program to characterize the models of the resulting updated program, i.e. the program-justified updates, whatever the updated program may be.

Definition 4.3.1 (Inertial Sub-Program) *Let P be a normal logic program over $\mathcal{L}_\neg$, U an update program over $\mathcal{L}_\neg$ and $M_\neg$ an interpretation over $\mathcal{L}_\neg$. Let:*

$$\begin{aligned} Rejected(M_\neg, P, U) = \ & \{L \leftarrow body \in P : M_\neg \models body \\ & \ and\ \exists \neg L \leftarrow body' \in U : M_\neg \models body' \ \} \end{aligned}$$

where L is an objective literal. We define Inertial Sub-Program $P_{inertial}(M_\neg, P, U)$ *as:*

$$P_{inertial}(M_\neg, P, U) = P - Rejected(M_\neg, P, U) \ \diamondsuit$$

Intuitively, the rules for some objective literal L that belong to $Rejected(M_\neg, P, U)$ are those that belong to the initial program but, although their body is still verified by the interpretation, there is an update rule that overrides them, by contravening their conclusion.

Definition 4.3.2 (<P,U>-Justified Updates) *Let P be a normal logic program, U an update program, and M an interpretation over the language of P. M is a* <P,U>-Justified Update *of P updated by U, iff $M_\neg$ is an answer-set of $P^*(M_\neg, P, U)$, where*

$$P^*(M_\neg, P, U) = P_{inertial}(M_\neg, P, U) + U \ \diamondsuit$$

Notice that the new definition of program-justified update doesn't depend on any initial model. Once again this is because inertia applies to rules and not model literals. To achieve inertia of model literals it is enough to include them as fact rules, as shown in the sequel.

The following example will illustrate the rôle played by $Rejected(M_\neg, P, U)$ when determining the <P,U>-Justified Updates.

Example 4.3.1 *Consider program P stating that someone is a pacifist and that a pacifist is a reasonable person. Later on, an update U states that it is not clear whether we're at war or at peace, and that a state of war will make that person no longer a pacifist:*

$$\begin{array}{llll} P: & pacifist \leftarrow & U: & \neg pacifist \leftarrow war \\ & reasonable \leftarrow pacifist & & peace \leftarrow not\ war \\ & & & war \leftarrow not\ peace \end{array}$$

Intuitively, when performing the update of P by U, we should obtain two models, namely

$$M_1 = \{pacifist, reasonable, peace\} \quad and \quad M_2 = \{war\}$$

Let's check whether they are $<P, U>$-justified updates. M_1 is $M_{\neg 1}$ restricted to the language of P:

$$M_{\neg 1} = \quad \{pacifist, reasonable, peace\}$$

Since

$$Rejected(M_{\neg 1}, P, U) = \{\}$$

$$P^*(M_{\neg 1}, P, U) = P + U - \{\}$$

$M_{\neg 1}$ is an answer-set of $P^*(M_{\neg 1}, P, U)$, and so M_1 is a $<P,U>$-justified update.

M_2 is $M_{\neg 2}$ restricted to the language of P:

$$M_{\neg 2} = \quad \{war, \neg pacifist\}$$

Since

$$Rejected(M_{\neg 2}, P, U) = \{pacifist \leftarrow\}$$

$$P^*(M_{\neg 2}, P, U) = P + U - \{pacifist \leftarrow\}$$

$M_{\neg 2}$ is an answer-set of $P^*(M_{\neg 2}, P, U)$ and so M_2 is a $<P,U>$-justified update.

Let's check if the model

$$M_X = \{reasonable, war\}$$

is a $<P, U>$-justified update. Intuitively it should not be one because the truth value of **reasonable** *should be determined by the evaluation of the rule of P,* **reasonable $\leftarrow$ pacifist**, *on the strength of the truth of* **pacifist** *in the updated model, and therefore should be false. Note, however, that this model would be a justified update of the only stable model of P, determined according to interpretation updating.*

Once again M_X is $M_{\neg X}$ restricted to the language of P:

$$M_{\neg X} = \quad \{reasonable, war, \neg pacifist\}$$

Since

$$Rejected(M_{\neg X}, P, U) = \{pacifist \leftarrow\}$$

$$P^*(M_{\neg X}, P, U) = P + U - \{pacifist \leftarrow\}$$

As expected, $M_{\neg X}$ is not an answer-set of $P^*(M_{\neg X}, P, U)$, and therefore M_X is not a $<P, U>$-justified update.

4.3.1 Properties of <P,U>-Justified Updates

Given the definition of <P,U>-justified updates we can demonstrate the following properties:

Theorem 4.3.1 (Properties of <P,U>-justified updates) *Let Q be a normal logic program over $\mathcal{L}_\neg$, and $\emptyset$ an empty program. Then:*

1. *the answer-sets of Q are exactly the $<Q,\emptyset>$-justified updates;*
2. *the answer-sets of Q are exactly the $<\emptyset,Q>$-justified updates;*
3. *the answer-sets of Q are exactly the $<Q,Q>$-justified updates.* $\diamondsuit$

Proof. Let $M_\neg$ an interpretation over $\mathcal{L}_\neg$. $M_\neg$ is a <P,U>-justified update iff it is an answer set of $P^*(M_\neg,P,U) = P_{inertial}(M_\neg,P,U) + U$. Note that $P_{inertial}(M_\neg,P,U) \subseteq P$.

1. Since $U = \emptyset$, then $P_{inertial}(M_\neg,Q,\emptyset) = Q$ and $P^*(M_\neg,Q,\emptyset) = Q$, and the answer-sets coincide.
2. Since $U = Q$ and $P_{inertial}(M_\neg,\emptyset,Q) \subseteq \emptyset$ then $P^*(M_\neg,\emptyset,Q) = Q$, and the answer-sets coincide.
3. Since $U = Q$ and $P_{inertial}(M_\neg,Q,Q) \subseteq Q$, then the answer-sets of $P^*(M_\neg,Q,Q) = Q + P_{inertial}(M_\neg,Q,Q)$ coincide with those of Q because we can eliminate from a program repeated occurences of some rule, still preserving its semantics. $\blacksquare$

Theorem 4.3.2 (Associativity of Updates) *Let P, Q and R be three normal logic programs over $\mathcal{L}_\neg$. Let $P^*(M_\neg,P,U)$ be the program obtained by Def.4.3.2, corresponding to the update of P by U given $M_\neg$. Let $M_\neg$ be an interpretation over $\mathcal{L}_\neg$. If P updated by Q (resp. Q updated by R) has at least one $<P,Q>$-justified update (resp. $<Q,R>$-justified update) then:*

- *$M_\neg$ is an answer-set of the program $P^*\left(M_\neg,P^*\left(M_\neg,P,Q\right),R\right)$ iff $M_\neg$ is an answer-set of the program $P^*\left(M_\neg,P,P^*\left(M_\neg,Q,R\right)\right)$.* $\diamondsuit$

Proof. We will first consider $P^*\left(M_\neg,P^*\left(M_\neg,P,Q\right),R\right)$.According to Def.4.3.2, if $M_\neg$ is an answer-set of $P^*(M_\neg,P,Q)$, it is a $<P,Q>$-justified update. Let $Rejected(M_\neg,P,Q)$ be the set of rules from P, rejected due to Q, given $M_\neg$, according to Def.4.3.1. We then have:

$$P^*(M_\neg,P,Q) = Q + (P - Rejected\,(M_\neg,P,Q))$$

and the update of this program by R is:

$$\begin{aligned} P^*\left(M_\neg, P^*\left(M_\neg, P, Q\right), R\right) &= R + \left[\left(Q + \left(P - Rejected\left(M_\neg, P, Q\right)\right)\right) - \right. \\ &\left. - \; Rejected\left(M_\neg, P^*\left(M_\neg, P, Q\right), R\right)\right] \end{aligned} \tag{4.1}$$

where:

$$Rejected\left(M_\neg, P^*\left(M_\neg, P, Q\right), R\right) = Rejected\left(M_\neg, \left(Q + \left(P - Rejected\left(M_\neg, P, Q\right)\right)\right), R\right)$$

note that we can further separate the set of rejected rules into two sets, namely the ones that belonged to Q and the ones that belonged to P and weren't rejected by Q, and obtain:

$$\begin{aligned} P^*\left(M_\neg, P^*\left(M_\neg, P, Q\right), R\right) &= Rejected\left(M_\neg, Q, R\right) + \\ &+ Rejected\left(M_\neg, \left(P - Rejected\left(M_\neg, P, Q\right)\right), R\right) \end{aligned} \tag{4.2}$$

Furthermore the set of rules that belong to $Rejected\left(M_\neg, \left(P - Rejected\left(M_\neg, P, Q\right)\right), R\right)$ is necessarily a sub-set of the rules of P that would be rejected due to R, i.e.

$$Rejected\left(M_\neg, \left(P - Rejected\left(M_\neg, P, Q\right)\right), R\right) \subseteq Rejected\left(M_\neg, P, R\right)$$

and the difference between those two sets of rules does not belong to $P^*(M_\neg, P, Q)$, i.e. it does not belong to $\left(Q + \left(P - Rejected\left(M_\neg, P, Q\right)\right)\right)$.

$$\begin{aligned} & Rejected\left(M_\neg, P, R\right) - Rejected\left(M_\neg, \left(P - Rejected\left(M_\neg, P, Q\right)\right), R\right) \subseteq \\ \subseteq \; & \left(Q + \left(P - Rejected\left(M_\neg, P, Q\right)\right)\right) \end{aligned}$$

therefore we can safely replace in 4.1 $Rejected\left(M_\neg, P^*\left(M_\neg, P, Q\right), R\right)$ by $\left[Rejected\left(M_\neg, Q, R\right) + Rejected\left(M_\neg, P, R\right)\right]$ and obtain:

$$\begin{aligned} P^*\left(M_\neg, P^*\left(M_\neg, P, Q\right), R\right) &= R + \left[\left(Q + \left(P - Rejected\left(M_\neg, P, Q\right)\right)\right) - \right. \\ &\left. - \; \left[Rejected\left(M_\neg, Q, R\right) + Rejected\left(M_\neg, P, R\right)\right]\right] \end{aligned}$$

which can further be rearranged to engender:

$$\begin{aligned} P^*\left(M_\neg, P^*\left(M_\neg, P, Q\right), R\right) &= R + \left[Q - Rejected\left(M_\neg, Q, R\right)\right] + \\ &+ \left[P - \left(Rejected\left(M_\neg, P, R\right) + \right.\right. \\ &\left.\left. + \; Rejected\left(M_\neg, P, Q\right)\right)\right] \end{aligned} \tag{4.3}$$

Intuitively, $P^*\left(M_\neg, P^*\left(M_\neg, P, Q\right), R\right)$ is composed of the rules of R, plus the rules of Q that weren't rejected due to R, plus the rules of P that weren't rejected due to R nor Q.

We will now consider the case where P is updated by the result of the update of Q by R:

$$P^*(M_\neg, Q, R) = R + (Q - \mathit{Rejected}\,(M_\neg, Q, R))$$

and

$$\begin{aligned} P^*\left(M_\neg, P, P^*\left(M_\neg, Q, R\right)\right) &= R + [Q - \mathit{Rejected}\,(M_\neg, Q, R)] + \\ &\quad + [P - \mathit{Rejected}\,(M_\neg, P, P^*\left(M_\neg, Q, R\right))] \end{aligned} \tag{4.4}$$

where:

$$\mathit{Rejected}\,(M_\neg, P, P^*\left(M_\neg, Q, R\right)) = \mathit{Rejected}\,(M_\neg, P, (R + (Q - \mathit{Rejected}\,(M_\neg, Q, R))))$$

we have that:

$$\begin{aligned} \mathit{Rejected}\,(M_\neg, P, (R + (Q - \mathit{Rejected}\,(M_\neg, Q, R)))) &= \mathit{Rejected}\,(M_\neg, P, R) + \\ &\quad + \mathit{Rejected}\,(M_\neg, P, (Q - \mathit{Rejected}\,(M_\neg, Q, R))) \end{aligned} \tag{4.5}$$

which is further equal to

$$\begin{aligned} \mathit{Rejected}\,(M_\neg, P, (R + (Q - \mathit{Rejected}\,(M_\neg, Q, R)))) &= \mathit{Rejected}\,(M_\neg, P, R) + \\ &\quad + (\mathit{Rejected}\,(M_\neg, P, Q) - \varepsilon) \end{aligned}$$

Before we go on to explain what ε is, let us first replace this last result back in 4.4 and obtain:

$$\begin{aligned} P^*\left(M_\neg, P, P^*\left(M_\neg, Q, R\right)\right) &= R + [Q - \mathit{Rejected}\,(M_\neg, Q, R)] + \\ &\quad + [P - (\mathit{Rejected}\,(M_\neg, P, R) \; + \\ &\quad + \; [\mathit{Rejected}\,(M_\neg, P, Q) - \varepsilon])] \end{aligned} \tag{4.6}$$

Now, note that ε is the set of rules from P that would be rejected due to a rule of Q, but are not rejected because that rule of Q has been rejected by R. Let $L \leftarrow B_p, not\ C_p$ (where L is an objective literal and each B and C a disjoint set of atoms) be one of such rules from P. Then there must be a rule in Q of the form $\neg L \leftarrow B_q, not\ C_q$ such that $M_\neg \models B_q, not\ C_q$ (note also that $M_\neg \models B_p, not\ C_p$). But if that rule was rejected due to R, there must be a rule in R of the form $L \leftarrow B_r, not\ C_r$ such that $M_\neg \models B_r, not\ C_r$. Since all rules of R belong to $P^*\left(M, P, P^*\left(M, Q, R\right)\right)$, then so will $L \leftarrow B_r, not\ C_r$ which makes the presence of the rule $L \leftarrow B_p, not\ C_p$ in $P^*\left(M, P, P^*\left(M, Q, R\right)\right)$ irrelevant (L would be concluded either way). Since the rules of ε are irrelevant, we can make $\varepsilon = \{\}$ and $P^*\left(M, P, P^*\left(M, Q, R\right)\right)$ in 4.6 is exactly equal to $P^*\left(M, P^*\left(M, P, Q\right), R\right)$ in 4.3. So if $M_\neg$ is an answer-set of one of them, it is an answer-set of both. ■

4.4 Update Transformation of a Normal Program

Next we present a program transformation that produces an updated program from an initial program and an update program. The answer-sets of the updated program so obtained will be exactly the <P,U>-justified updates, according to Theorem 4.4.1 below. The updated program can then be used to compute them.

Definition 4.4.1 (Update transformation of a normal program) *Consider an update program U over a language $\mathcal{L}_\neg$. For a normal logic program P over $\mathcal{L}_\neg$, its updated program P_U with respect to U , written in the extended language $\mathcal{L}_\neg + \{A', \neg\_A', A^U, \neg A^U : A \in \mathcal{L}\}$ is obtained via the operations:*

- *All rules of U and P belong to P_U subject to the changes:*
  - *in the head of every rule of P_U originated in U replace literal L by a new literal L^U;*
  - *in the head of every rule of P_U originated in P replace atom A (resp. $\neg A$) by a new atom A' (resp $\neg\_A'$);*
- *Include in P_U, for every atom A in $\mathcal{L}$, the defining rules:*

$$\begin{array}{ll} A \leftarrow A', not\ \neg A^U & A \leftarrow A^U \\ \neg A \leftarrow \neg\_A', not\ A^U & \neg A \leftarrow \neg A^U\ \diamondsuit \end{array}$$

The above definition assumes that in the language $\mathcal{L}_\neg$ there are no symbols of the form A', $\neg\_A'$, A^U and $\neg A^U$. This transformation is reminiscent of the one presented in [5], where the goal was to update a set of models encoded by a logic program. In [5], literals figuring in the head of a rule of U (but it could be for any literal) originate replacement of the corresponding atom in both the head and body of the rules of the initial program, whereas in the above transformation this replacement occurs only in the head (for all rules). This has the effect of exerting inertia on the rules instead of on the model literals because the original rules will be evaluated in the light of the updated model. The defining rules establish that, after the update, a literal is either implied by inertia or forced by an update rule. Note that only update rules are allowed to inhibit inertia rules, in contrast to the usual inertia rules for model updates. In model updates there are no rule bodies in the coding of the initial interpretation as fact rules, so the conclusion of these rules cannot change, in contradistinction to the case of program updates. Hence the new inertia rules, which apply equally well to model updating (cf. justification in Theorem 4.4.2) and so are more general. Their intuitive reading is: A can be true (resp. false) either by inertia or due to the update program.

Example 4.4.1 *Consider the normal logic program P with single stable model M:*

$$P: \quad \begin{array}{l} a \leftarrow not\ b \\ d \leftarrow e \\ e \leftarrow \end{array} \qquad M = \{a, d, e\}$$

now consider the update program U:

$$U: \quad \begin{array}{l} c \leftarrow not\ a \\ b \leftarrow \\ \neg e \leftarrow a \end{array}$$

And the updated program P_U is (where the rules for A stand for all their ground instances):

$$\begin{array}{llll} c^U \leftarrow not\ a & a' \leftarrow not\ b & A \leftarrow A', not\ \neg A^U & A \leftarrow A^U \\ b^U \leftarrow & d' \leftarrow e & \neg A \leftarrow \neg\_A', not\ A^U & \neg A \leftarrow \neg A^U \\ \neg e^U \leftarrow a & e' \leftarrow & & \end{array}$$

whose only answer-set (modulo A', A^U and explicitly negated atoms) is:

$$M_U = \{b, c, d, e\}$$

This corresponds to the intended result: the insertion of **b** *renders* **a** *no longer supported and thus false; since* **a** *is false,* **c** *becomes true due to the first rule of the update program; the last rule of U is ineffective since* **a** *is false;* **e** *is still supported and not updated, so it remains true by inertia; finally* **d** *remains true because still supported by* **e**.

If we consider this same example but performing the updating on a model basis instead, we would get as the only U-justified update of M: $M' = \{a, b, d\}$. The difference, for example in what **a** is concerned, is that in M' **a** is true by inertia because it is true in M and there are no rules for **a** in U. According to our definition, since there aren't any rules (with a true body) in U for **a**, the rule in P for **a** is still valid by inertia and re-evaluated in the final interpretation, where since **b** is true **a** is false.

Example 4.4.2 *Consider the P and U of example 4.3.1. The updated program P_U of P by U is:*

$$\begin{array}{ll} pacifist' \leftarrow & A \leftarrow A', not\ \neg A^U \\ reasonable' \leftarrow pacifist & \neg A \leftarrow \neg\_A', not\ A^U \\ \neg pacifist^U \leftarrow war & A \leftarrow A^U \\ peace^U \leftarrow not\ war & \neg A \leftarrow \neg A^U \\ war^U \leftarrow not\ peace & \end{array}$$

whose answer-sets (modulo A', A^U and explicitly negated atoms) are:

$$M_1 = \{pacifist, reasonable, peace\}$$
$$M_2 = \{war\}$$

coinciding with the two <P, U>-justified updates determined in example 4.3.1.

The following theorem establishes the relationship between the models of the update transformation of a program and its <P,U>-justified updates.

Theorem 4.4.1 (Correctness of the update transformation) *Let P be a normal logic program over $\mathcal{L}_\neg$ and U an update program over $\mathcal{L}_\neg$. Modulo any primed, L^U and explicitly negated literals, the answer-sets of the updated program P_U are exactly the <P, U>-Justified Updates of P updated by U.* $\diamondsuit$

Proof. Let P be a normal logic program consisting of rules of the form:

$$A \leftarrow B_i, not\ C_i$$
$$\neg A \leftarrow B_m, not\ C_m$$

and U an update program consisting of rules of the form:

$$A \leftarrow B_j, not\ C_j$$
$$\neg A \leftarrow B_k, not\ C_k$$

where A is an atom and each B and C is some finite set of atoms .

Let $P_U^*(M_\neg, P, U)$ be the program obtained according to Def. 4.3.2:

$$P_U^*(M_\neg, P, U) = U + P_{inertial}(M_\neg, P, U)$$

and note that $P_{inertial}(M_\neg, P, U) \subseteq P$.

Let P_U be the program obtained according to Def. 4.4.1:

$$P_U : \quad \begin{array}{ll} \left.\begin{array}{l} A' \leftarrow B_i, not\ C_i \\ \neg\_A' \leftarrow B_m, not\ C_m \end{array}\right\} & \text{for all rules from } P \\ \left.\begin{array}{l} A \leftarrow A', not\ \neg A^U \\ \neg A \leftarrow \neg\_A', not\ A^U \\ A \leftarrow A^U \\ \neg A \leftarrow \neg A^U \end{array}\right\} & \text{for all } A \\ \left.\begin{array}{l} A^U \leftarrow B_j, not\ C_j \\ \neg A^U \leftarrow B_k, not\ C_k \end{array}\right\} & \text{for all rules from } U \end{array}$$

We will show that P_U is equivalent to $P_U^*(M_\neg, P, U)$ for our purposes. Performing on P_U a partial evaluation of A^U and $\neg A^U$ on the rules $A \leftarrow A^U$ and $\neg A \leftarrow \neg A^U$ we obtain:

$$
\begin{array}{lll}
P_U^{'}: & A' \leftarrow B_i, not\ C_i & (1) \\
& \neg\_A' \leftarrow B_m, not\ C_m & (2) \\
& A \leftarrow A', not\ \neg A^U & (3) \\
& \neg A \leftarrow \neg\_A', not\ A^U & (4) \\
& A \leftarrow B_j, not\ C_j & (5) \\
& \neg A \leftarrow B_k, not\ C_k & (6) \\
& A^U \leftarrow B_j, not\ C_j & (7) \\
& \neg A^U \leftarrow B_k, not\ C_k & (8)
\end{array}
$$

Note that rules (5) and (6) are exactly the update program.

These rules can be simplified. In particular we don't need the rules for A^U and $\neg A^U$. For some arbitrary A, consider first the case where both A^U and $\neg A^U$ are false. We can then perform the following simplifications on $P_U^{'}$: replace in (3) (resp. (4)) A' (resp.$\neg\_A'$) by the body of (1) (resp. (2)) and remove *not* $\neg A^U$ (resp.*not* A^U) to obtain (3*) (resp. (4*)): $A \leftarrow B_i, not\ C_i$(resp.$\neg A \leftarrow B_m, not\ C_m$) ; now we no longer need rules (7) and (8). Since we don't care about primed literals in the updated models we can now remove rules (1) and (2). The so mutilated $P_U^{'}$ preserves the semantics of $P_U^{'}$ when both A^U and $\neg A^U$ are false, apart primed and U literals, and looks like this:

$$
\begin{array}{ll}
A \leftarrow B_i, not\ C_i & (3*) \\
\neg A \leftarrow B_m, not\ C_m & (4*) \\
A \leftarrow B_j, not\ C_j & (5) \\
\neg A \leftarrow B_k, not\ C_k & (6)
\end{array}
$$

which corresponds exactly to $P_U^*(M_\neg, P, U)$ when $P_{inertial}(M_\neg, P, U) = P$ when both A^U and $\neg A^U$ are false, and hence their answer-sets are the same in that case.

For the case where $\neg A^U$ is true and A^U is false, we can delete rule (3); replace in (4) $\neg\_A'$ by the body of (2) and remove *not* A^U to obtain (4*): $\neg A \leftarrow B_m, not\ C_m$; now we no longer need rules (7) and (8). Since we don't care about primed literals in the updated models we can now remove rules (1) and (2). The so mutilated $P_U^{'}$ preserves the semantics of $P_U^{'}$ when $\neg A^U$ is true, apart primed and U literals, and looks like this:

$$
\begin{array}{ll}
\neg A \leftarrow B_m, not\ C_m & (4*) \\
A \leftarrow B_j, not\ C_j & (5) \\
\neg A \leftarrow B_k, not\ C_k & (6)
\end{array}
$$

which is semantically equal to $P_U^*(M_\neg, P, U)$. Indeed, note that when $\neg A^U$ is true, the rules of P for A are rejected if $M_\neg \models B_i, not\ C_i$ and don't belong to $P_U^*(M_\neg, P, U)$. So the only possible difference between the simplified $P_U^{'}$ and $P_U^*(M_\neg, P, U)$ would be the existence of some extra rules in $P_U^*(M_\neg, P, U)$ such that for any answer-set $M_\neg$ we would have $M_\neg \not\models B_i, not\ C_i$, which does not affect the semantics.

For the case where A^U is true and $\neg A^U$ is false, we can delete rule (4); replace in (3) A' by the body of (1) and remove $not\ \neg A^U$ to obtain (3*): $A \leftarrow B_i, not\ C_i$; now we no longer need rules (7) and (8). Since we don't care about primed literals in the updated models we can now remove rules (1) and (2). The so mutilated $P_U^{'}$ preserves the semantics of $P_U^{'}$ when $\neg A^U$ is true, apart primed and U literals, and looks like this:

$$\begin{array}{ll} A \leftarrow B_i, not\ C_i & (3*) \\ A \leftarrow B_j, not\ C_j & (5) \\ \neg A \leftarrow B_k, not\ C_k & (6) \end{array}$$

which is semantically equal to $P_U^*(M_\neg, P, U)$. Indeed, note that when A^U is true, the rules of P for $\neg A$ are rejected if $M_\neg \models B_m, not\ C_m$ and don't belong to $P_U^*(M_\neg, P, U)$. So the only possible difference between the simplified $P_U^{'}$ and $P_U^*(M_\neg, P, U)$ would be the existence of some extra rules in $P_U^*(M_\neg, P, U)$ such that for any answer-set $M_\neg$ we would have $M_\neg \not\models B_m, not\ C_m$, which does not affect the semantics.

For the case where both A^U and $\neg A^U$ are true (this case is contradictory, but we will check it anyway), we can delete rule (3) and (4); now we no longer need rules (7) and (8). Since we don't care about primed literals in the updated models we can now remove rules (1) and (2). The so mutilated $P_U^{'}$ preserves the semantics of $P_U^{'}$ when both A^U and $\neg A^U$ are true, apart primed and U literals, and looks like this:

$$\begin{array}{ll} A \leftarrow B_j, not\ C_j & (5) \\ \neg A \leftarrow B_k, not\ C_k & (6) \end{array}$$

which is semantically equal to $P_U^*(M_\neg, P, U)$. Indeed, note that when both A^U and $\neg A^U$ are true, the rules of P with a true body in $M_\neg$ are all rejected. So, again, the only possible difference between the simplified $P_U^{'}$ and $P_U^*(M_\neg, P, U)$ would be the existence of some extra rules in $P_U^*(M_\neg, P, U)$ such that for any answer-set $M_\neg$ we would have $M_\neg \not\models B_m, not\ C_m$, which does not affect the semantics, i.e. a contradiction. ■

The next theorem establishes the relationship between program update and interpretation update. For this we begin by defining a transformation from an interpretation into the arguably simplest normal logic program that encodes it.

Definition 4.4.2 (Factual LP) *Let I be an interpretation over a language $\mathcal{L}$. We define the normal logic program associated with I, $P^*(I)$, as:*

$$P^*(I) = \{L \leftarrow : L \in I\} \ \diamondsuit$$

It is worth pointing out that the translation of update programs into extended logic programs, making use of explicit negation $\neg$ to code the out's in the heads of update rules and default negation *not* to code the out's in the bodies of the same rules, allows for some pairs of answer-sets, one of which will always be closer than the other to the initial interpretation. This is best illustrated by the following example:

Example 4.4.3 *Let $I = \{a\}$ and $U = \{\neg a \leftarrow not\ a\}$ where U is the translation of $U' = \{out(a) \leftarrow out(a)\}$ according to Def.4.2.3. The updated program is:*

$$\begin{aligned} P_U: \quad & a' \leftarrow \\ & a \leftarrow a', not\ \neg a^U \\ & \neg a^U \leftarrow not\ a \end{aligned}$$

with two answer-sets whose restrictions are $M_1 = \{a\}$ and $M_2 = \{\}$. Note that M_1 is closer to I than M_2.

The closeness condition in theorem 4.4.2 below, exists to eliminate such farther models in order to obtain the U-justified updates only. As mentioned, this phenomena is due to the translation of the update programs.

This translation has the virtue of not excluding such models, just in case they are seen as desired.

Theorem 4.4.2 (Generalization of Updates) *Let U be an update program and I an interpretation. Then:*

1. *Every U-justified update of I is a $<P^*(I), U>$-justified update.*
2. *A $<P^*(I), U>$-justified update I_u is a U-justified update of I iff*

$$\nexists I_u' \text{ closer to } I \text{ than } I_u$$

where I_u' is a $<P^*(I), U>$-justified update. $\diamondsuit$

Proof.

1. Let U be an update program consisting of rules of the form:

$$A \leftarrow B_j, not\ C_j$$
$$\neg A \leftarrow B_k, not\ C_k$$

where A is an atom and each B and C is some finite set of atoms.

According to [5], an interpretation I_u is a U-justified update of I iff it is a total (or two-valued) WFSX model (modulo primed and explicitly negated elements) of the corresponding program P_U:

$$\begin{array}{lll} P_U: & A' \leftarrow & \text{for all } A \in I \\ & \left.\begin{array}{l} A \leftarrow A', not\ \neg A \\ \neg A \leftarrow not\ A', not\ A \end{array}\right\} & \text{for all } A \\ & \left.\begin{array}{l} A \leftarrow B_j, \neg C_j \\ \neg A \leftarrow B_k, \neg C_k \end{array}\right\} & \text{for all rules from } U \end{array}$$

according to Def. 4.4.1, an interpretation $I_u^{'}$ is a $<P^*(I), U>$-justified update iff it is the restriction to the language of I of an answer-set of the program $P_U^{'}$ (we ommit the inertia rules of the form $\neg A \leftarrow \neg\_A', not\ A^U$ because there are no rules for $\neg\_A'$):

$$\begin{array}{lll} P_U^{'}: & A' \leftarrow & \text{for all } A \in I \\ & \left.\begin{array}{l} A \leftarrow A', not\ \neg A^U \\ A \leftarrow A^U \\ \neg A \leftarrow \neg A^U \end{array}\right\} & \text{for all } A \\ & \left.\begin{array}{l} A^U \leftarrow B_j, not\ C_j \\ \neg A^U \leftarrow B_k, not\ C_k \end{array}\right\} & \text{for all rules from } U \end{array}$$

Notice the difference in the translation of update rules in what the kind of negation used in their bodies is concerned. We will show that for every total (or two-valued) WFSX model I_u of the program P_U, there is an answer-set $I_u^{'}$ of $P_U^{'}$ such that $I_u = I_u^{'}$ restricted to the language of I.

Performing a partial evaluation of A^U and $\neg A^U$ on the rules $A \leftarrow A^U$ and $\neg A \leftarrow \neg A^U$ we obtain:

$$\begin{array}{lll} P_U^{'}: & A' \leftarrow & (1) \\ & A \leftarrow A', not\ \neg A^U & (2) \\ & A \leftarrow B_j, not\ C_j & (3) \\ & \neg A \leftarrow B_k, not\ C_k & (4) \\ & A^U \leftarrow B_j, not\ C_j & (5) \\ & \neg A^U \leftarrow B_k, not\ C_k & (6) \end{array}$$

We can safely replace $not\ \neg A^U$ by $not\ \neg A$ in rule (2), for the only rules for $\neg A$ and $\neg A^U$ have the same body. Now, and since we don't care about A^U and $\neg A^U$ in the updated models, we can remove rules (5) and (6) and obtain the following program P_U'':

$$\begin{array}{llll}
P_U'': & A' \leftarrow & (1) & P_U: \quad A' \leftarrow \\
& A \leftarrow A', not\ \neg A & (2) & \qquad A \leftarrow A', not\ \neg A \\
& & (3) & \qquad \neg A \leftarrow not\ A', not\ A \\
& A \leftarrow B_j, not\ C_j & (4) & \qquad A \leftarrow B_j, \neg C_j \\
& \neg A \leftarrow B_k, not\ C_k & (5) & \qquad \neg A \leftarrow B_k, \neg C_k
\end{array}$$

It is easy to see that the only differences between P_U'' and P_U are the kind of negation used in the body of the rules from the update program, and the extra rule (3) in P_U. Suppose that we add rule (3) to P_U'': if rule (3) has a true body, rule (2) must have a false body; since we are not concerned about $\neg A$ in the final models, and $\neg A$ doesn't appear in the body of any other rules, adding rule (3) to P_U'' wouldn't change the restricted models. Now, the only difference is the kind of negation used, but since by coherence we have that if $\neg C$ is true then $not\ C$ is also true, we have that all total WFSX models of P_U are also answer-sets of P_U''.

2. There now remains to be proved the closeness part of the theorem, i.e. that the set of interpretations $S = Q - R$, where

$$\begin{array}{l}
Q = \{I_u : I_u \text{ is a } < P^*(I), U > \text{-justified update}\} \\
R = \{I_u : I_u \text{ is a } U\text{-justified update of } I\}
\end{array}$$

is such that for every I_u' in S, there is an I_u in R such that I_u is closer to I than I_u', and thus eliminated by the closeness condition. According to [25], I_u is a U-justified update of I iff it satisfies the rules of U (as per Def.3.1.1 and where I satisfies $in(a)$ (resp. $out(a)$) if $a \in I$ (resp. $a \notin I$)), and is closest to I among such interpretations. From definition 4.3.2, every $<P, U>$-justified update must satisfy the rules of U, of the form:

$$\begin{array}{l}
A \leftarrow B_j, not\ C_j \\
\neg A \leftarrow B_k, not\ C_k
\end{array} \tag{4.7}$$

Since for any answer-set if $\neg a \in I$ then $a \notin I$, we have that any $<P, U>$-justified update, because it satisfies the rules of (4.7), must also satisfy the update rules with in's and out's of the form (4.8)

$$\begin{array}{l}
in(A) \leftarrow in(B_j), out(C_j) \\
out(A) \leftarrow in(B_k), out(C_k)
\end{array} \tag{4.8}$$

Let X be the set of all interpretations that satisfy the rules of (4.8). Then the interpretations in $X - R$ are the ones eliminated by the closeness condition, to obtain the U-justified updates, according to [25]. Since $R \subseteq Q$ (first part of the theorem), and every interpretation of Q satisfies the rules of (4.8), we have that $S \subseteq X$ and thus any interpretation in S is eliminated by the closeness condition of this theorem. ■

Therefore the notion of program update presented here is a generalization of the updates carried out on a model basis.

Chapter 5

Iterated Updates

5.1 Introduction

What happens if we want to update a program more than once? Since the definitions in the previous chapter only apply to programs without explicit negation in the body of their rules, and updated programs obtained from applying def.4.4.1 include such a form of negation in the body of inertia rules, those definitions cannot be used to update previously updated programs.

But if we take a closer look at the rôle of inertia rules, as well as the intuition behind the characterization of the justified updates, we can see that inertia rules don't need to be updated. Indeed, given a sequence of update programs to be applied to an initial program (which, as we've seen in theorem 4.3.1, can itself be envisaged as an update program applied to the empty program), the justified updates should be answer-sets of a program composed by the rules of the last update program, together with some rules from each of the previous update programs, that haven't been overriden by a subsequent update. Equally, to construct a program whose answer-sets are the justified updates, we might just use the rules of each update program, together with some rules enacting inertia from one stage to the next.

In this chapter we will generalize the notions presented in the previous one to the case where we have a sequence of update programs. For this we start with the following definition, allowing us to "time-stamp" updates.

Definition 5.1.1 (State Set) *Let $(T, <)$ be a countably infinite totally ordered set with the smallest element t_0 such that for any $t \in T - \{t_0\}$ there exists, in T, an element $t-1$ immediately preceding t, i.e., such that:*

- $t - 1 < t$;
- for any $s < t$ we have $s \leq t - 1$.

We further assume that T is well founded, i.e. every strictly decreasing sequence of its elements is necessarily finite. This guarantees that by taking a finite number of predecessors $t - 1$, the smallest element t_0 will eventually be reached.

We call members of T states, the first element t_0 the initial state and T the *State Set*. Also, for every T, we call the last element t_c, such that $\forall_{t \in T - \{t_c\}}, t_c > t$, the current state. For simplicity, we assume T to be a sequence of natural numbers.

5.2 S-justified Updates

We will now characterize the set of interpretations that should be accepted as justified updates, at any given state, in a sequence of updates. They will be called the S-justified updates. Like for <P,U>-justified updates, we start by defining the set of rules, of each update program, that should persist up to the state under consideration.

Definition 5.2.1 (Inertial Sub-Program of state t_1 at state t) *Let $S = \{U_t : t \in T\}$ be an ordered sequence in T of update logic programs U_t over $\mathcal{L}_\neg$, and $M_\neg$ an interpretation over $\mathcal{L}_\neg$. Let:*

$$Rejected(M_\neg, t_1, t) = \bigcup_{t_1 < t_2 \leq t} Rejected_i(M_\neg, t_1, t_2)$$

$$\begin{aligned} Rejected_i(M_\neg, t_1, t_2) = \ & \{L \leftarrow body \in U_{t_1} : M_\neg \models body \\ & and\ \exists \neg L \leftarrow body' \in U_{t_2} : M_\neg \models body'\} \end{aligned}$$

where L is an atom. We define Inertial Sub-Program of state t_1 at state t*, $U^{inertial}(M_\neg, t_1, t)$ as:*

$$U^{inertial}(M_\neg, t_1, t) = U_{t_1} - Rejected(M_\neg, t_1, t) \ \diamondsuit$$

Note that $Rejected(M_\neg, t, t) = \{\}$ and so $U^{inertial}(M_\neg, t, t) = U_{t_1}$.

Intuitively, the rules for some literal L that belong to $Rejected(M_\neg, t_1, t)$ are those that belong to U_{t_1} but, although their body is still verified by the model, there is a rule of a subsequent update program that overrides them, by contravening their conclusion.

Definition 5.2.2 (S-Justified Updates at a given state t) *Let $S = \{U_t : t \in T\}$ be an ordered sequence in T of update logic programs U_t over $\mathcal{L}_\neg$, with the smallest element*

U_{t_0}, and M an interpretation over $\mathcal{L}$. M is a S-Justified Update at a given state t *iff $M_\neg$ is an answer-set of $P^*_S(t)$, where*

$$P^*_S(t) = \bigcup_{t_i \le t} \left(U^{inertial}(M_\neg, t_i, t)\right)$$

and $t_i \in T$, if there is a non-contradictory S-justified update at every state t_i, $t_0 \le t_i \le t$. Otherwise there is no S-justified update at state t.

Intuitively, if $M_\neg$ is an answer-set of the program consisting of the rules from the update program at state t, together with every rule belonging to a program of a previous state whose conclusion has not been contravened by a rule of a subsequent update program (up to t). The non-contradiction condition for every previous state ensures that we reached state t without passing a contradictory state. Adressing contradiction is beyond the scope of this work.

Note that if a rule is rejected by some rule, which itself is subsequently rejected, there is no need to reinstate the original rule because the rule rejecting the rejector rule takes it place.

Theorem 5.2.1 *Let $S = \{U_0, ..., U_n\}$be an ordered sequence of update logic programs U_n over $\mathcal{L}_\neg$, with the smallest element U_0, If there is at least one S-justified update at state $n-1$ then the S-justified updates at state n are exactly the $<P^*_S(n-1), U_n>$-justified updates.*

Proof. According to def. 5.2.2, M is a S-Justified Update at state n iff $M_\neg$ is an answer-set of $P^*_S(n)$, where

$$P^*_S(t) = \bigcup_{t_i \le t} \left(U^{inertial}(M_\neg, t_i, n)\right)$$

since $U^{inertial}(M_\neg, t, t) = U_{t_1}$, we obtain

$$P^*_S(n) = U_n + \bigcup_{t_i < n} \left(U^{inertial}(M_\neg, t_i, n)\right) \tag{5.1}$$

Note that since we are sure to have at least one S-justified update at state $n-1$, we are also sure to have at least one S-justified update at every state $r \in T, r < n$. Solving for $U^{inertial}$, and bringing one term outside the union, we obtain:

$$P^*_S(n) = U_n + (U_{n-1} - Rejected(M_\neg, n-1, n)) + \bigcup_{t_i < n-1} (U_{t_i} - Rejected(M_\neg, t_i, n))$$

solving it for $Rejected(M_{\neg},\_,\_)$ we obtain:

$$\begin{aligned} P^*_S(n) &= U_n + (U_{n-1} - Rejected_i(M_{\neg}, n-1, n)) + \\ &+ \bigcup_{t_i<n-1} \left(U_{t_i} - \bigcup_{t_i<t_2\leq n} Rejected_i(M_{\neg}, t_i, t_2) \right) \end{aligned} \tag{5.2}$$

Let's now look at the $<P^*_S(n-1),U_n>$-justified updates. $P^*_S(n-1)$ is given by:

$$P^*_S(n-1) = \bigcup_{t_i\leq n-1} \left(U^{inertial}(M_{\neg}, t_i, n-1) \right)$$

which is the same as

$$P^*_S(n-1) = U_{n-1} + \bigcup_{t_i<n-1} \left(U^{inertial}(M_{\neg}, t_i, n-1) \right)$$

that can further be rewritten as:

$$P^*_S(n-1) = U_{n-1} + \bigcup_{t_i<n-1} \left(U_{t_i} - \bigcup_{t_i<t_2\leq n-1} Rejected_i(M_{\neg}, t_i, t_2) \right)$$

According to def.4.3.2, M is a $<P^*_S(n-1),U_n>$-justified update iff $M_{\neg}$ is an answer-set of $P^*(M_{\neg}, P^*_S(n-1), U_n)$ (or P^* for short), where

$$\begin{aligned} P^* &= U_n + \left[\left(U_{n-1} + \left(\bigcup_{t_i<n-1} U_{t_i} - \bigcup_{t_i<t_2\leq n-1} Rejected_i(M_{\neg}, t_i, t_2) \right) \right) - \right. \\ &\quad \left. - Rejected(M_{\neg}, P^*_S(n-1), U_n) \right] \end{aligned}$$

where $Rejected(M_{\neg}, P^*_S(n-1), U_n)$ are the rules of $P^*_S(n-1)$ that are rejected given U_n according to def. 4.3.2. Given the equivalence between this definition and the one for $Rejected_i(M_{\neg}, t_i, t_2)$. Note that we can determine $Rejected(M_{\neg}, P^*_S(n-1), U_n)$ for all the rules in $U_t, t \leq n-1$. Although some of those rules have already been rejected, it will not affect the result, i.e. if we subtract some rules to a program that doesn't have them, it is the same as if we didn't sutract them. We then obtain:

$$Rejected(M_{\neg}) = \bigcup_{t_i\leq n-1} Rejected_i(M_{\neg}, t_i, n)$$

replacing $Rejected(M_{\neg}, P^*_S(n-1), U_n)$ in $P^*(M_{\neg}, P^*_S(n-1), U_n)$ we obtain:

$$\begin{aligned} P^* &= U_n + \left[\left(U_{n-1} + \bigcup_{t_i<n-1} \left(U_{t_i} - \bigcup_{t_i<t_2\leq n-1} Rejected_i(M_{\neg}, t_i, t_2) \right) \right) - \right. \\ &\quad \left. - \bigcup_{t_i\leq n-1} Rejected_i(M_{\neg}, t_i, n) \right] \end{aligned}$$

which is equivalent to (moving one element outside the last union and combinig the rest with the other union):

$$P^* = U_n + (U_{n-1} - Rejected_i(M_\neg, n-1, n)) + \\ + \bigcup_{t_i < n-1} \left(U_{t_i} - \bigcup_{t_i < t_2 \leq n} Rejected_i(M_\neg, t_i, t_2) \right) \quad (5.3)$$

Since 5.3 is equal to 5.2, the theorem follows. ■

As an immediate corollary we obtain:

Corollary 5.2.2 (S-justified updates generalize <P,U>-justified updates) *Let P and U be update programs over $\mathcal{L}_\neg$. Let $T = \{0, 1\}$ and $S = \{U_0, U_1\}$ such that $U_0 = P$ and $U_1 = U$. Then every S-justified update at state 1 is a <P,U>-justified update. Conversely, every <P,U>-justified update is an S-justified update at state 1.*

5.3 Iterated Update Transformation

Next, we define a transformation that, given a sequence of update logic programs, produces a logic program. It does so by introducing inertia rules to 'link' the intermediate programs in the given sequence. This produced program, together with defining rules as will be seen below, has exactly as models the S-justified Updates of the update sequence.

Definition 5.3.1 (Iterated Program Updates) *Let $S = \{U_t : t \in T\}$be an ordered sequence of update logic programs U_t over $\mathcal{L}_\neg$, with the smallest element U_{t_0}. The* Iterated Update Program *P_S defined by S, in the extended language $\mathcal{L}^*_\neg = \mathcal{L}_\neg + \{A_t, \neg\_A_t, A^U_t, \neg\_A^U_t : A \in \mathcal{L}, t \in T\}$ is obtained via the operations:*

- *All rules of $U_t \in S$ belong to P_S subject to the changes:*
  - *in the head of every rule of P_S originated in U_t replace atom A (resp. $\neg A$) by a new atom A^U_t (resp $\neg\_A^U_t$);*
- *Include in P_S, for every atom A of $\mathcal{L}$, and for any $t > t_0$, the defining rules:*

$$\begin{array}{ll} A_t \leftarrow A_{t-1}, not\ \neg\_A^U_t & A_t \leftarrow A^U_t \\ \neg\_A_t \leftarrow \neg\_A_{t-1}, not\ A^U_t & \neg\_A_t \leftarrow \neg\_A^U_t \end{array}$$

- *Include in P_S, for every atom A of $\mathcal{L}$, the rules:*

$$A_{t_0} \leftarrow A^U_{t_0} \quad and \quad \neg\_A_{t_0} \leftarrow \neg\_A^U_{t_0} \ \diamond$$

The above definition assumes that in the language $\mathcal{L}_\neg$ there are no symbols of the form A_t, $\neg\_A_t$, A_t^U and $\neg\_A_t^U$. Note that P_S does not contain any rules for atoms of $\mathcal{L}_\neg$, i.e. neither A nor $\neg A$ appear in the heads of the rules of P_S.

Now, in order to find out what is the result of the updates up to a given state t, we add the "current state" defining rules to obtain the program at a given state t:

Definition 5.3.2 (Iterated Update Program at a current state t.) *Given a non-contradictory iterated update program P_S, the* Iterated Update Program at a given current state t $\in T$, *$P_S(t)$, is the program P_S augmented with the* current state *defining rules:*

$$A \leftarrow A_t \quad and \quad \neg A \leftarrow \neg\_A_t$$

for all $A \in \mathcal{L}$, if $P_S(r)$ is non-contradictory for all $r \in T, r < t$. Otherwise $P_S(t)$ is not defined.

Theorem 5.3.1 *Let $S = \{U_0, ..., U_n\}$be an ordered sequence of update logic programs U over $\mathcal{L}_\neg$, with the smallest element U_0. Whenever $P_S(n)$ is defined, it is semantically equivalent to the updated program of $P_S(n-1)$ by U_n, P_U.*

Proof. Let U_t consist of rules of the form:

$$\begin{array}{l} A \leftarrow B_t, not\ C_t \\ \neg A \leftarrow B_{\neg t}, not\ C_{\neg t} \end{array}$$

where A is an atom and each B and C is some finite set of atoms, and $t \leq n$. According to Def.5.3.2, $P_S(n)$ is:

$$P_S(n):$$

$$\left.\begin{array}{l} A_0 \leftarrow A_0^U \\ \neg\_A_0 \leftarrow \neg\_A_0^U \end{array}\right\} \text{for all } A \text{ in } \mathcal{L} \qquad \left.\begin{array}{l} A_t \leftarrow A_{t-1}, not\ \neg\_A_t^U \\ \neg\_A_t \leftarrow \neg\_A_{t-1}, not\ A_t^U \\ A_t \leftarrow A_t^U \\ \neg\_A_t \leftarrow \neg\_A_t^U \end{array}\right\} \begin{array}{l} \text{for all } A \text{ in } \mathcal{L} \\ 0 < t \leq n \end{array}$$

$$\left.\begin{array}{l} A \leftarrow A_n \\ \neg A \leftarrow \neg\_A_n \end{array}\right\} \text{for all } A \text{ in } \mathcal{L}$$

$$\left.\begin{array}{l} A_t^U \leftarrow B_t, not\ C_t \\ \neg\_A_t^U \leftarrow B_{\neg t}, not\ C_{\neg t} \end{array}\right\} \begin{array}{l} \text{rules from } U_t \\ t \leq n \end{array}$$

this program can be rewritten to the following form:

$$
\begin{array}{ll}
P_S(n): & \\
\left.\begin{array}{l} A_0 \leftarrow A_0^U \\ \neg\_A_0 \leftarrow \neg\_A_0^U \end{array}\right\} \text{for all } A \text{ in } \mathcal{L} &
\left.\begin{array}{l} A_t \leftarrow A_{t-1}, not\ \neg\_A_t^U \\ \neg\_A_t \leftarrow \neg\_A_{t-1}, not\ A_t^U \\ A_t \leftarrow A_t^U \\ \neg\_A_t \leftarrow \neg\_A_t^U \end{array}\right\} \begin{array}{l} \text{for all } A \text{ in } \mathcal{L} \\ 0 < t \leq n-1 \end{array} \\
\left.\begin{array}{l} A \leftarrow A_n \\ \neg A \leftarrow \neg\_A_n \end{array}\right\} \text{for all } A \text{ in } \mathcal{L} & \\
\left.\begin{array}{l} A_n^U \leftarrow B_n, not\ C_n \\ \neg\_A_n^U \leftarrow B_{\neg n}, not\ C_{\neg n} \end{array}\right\} \begin{array}{l} \text{rules} \\ \text{from } U_n \end{array} &
\left.\begin{array}{l} A_t^U \leftarrow B_t, not\ C_t \\ \neg\_A_t^U \leftarrow B_{\neg t}, not\ C_{\neg t} \end{array}\right\} \begin{array}{l} \text{rules from } U_t \\ t \leq n-1 \end{array} \\
\left.\begin{array}{l} A_n \leftarrow A_{n-1}, not\ \neg\_A_n^U \\ \neg\_A_n \leftarrow \neg\_A_{n-1}, not\ A_n^U \\ A_n \leftarrow A_n^U \\ \neg\_A_n \leftarrow \neg\_A_n^U \end{array}\right\} \text{for all } A \text{ in } \mathcal{L} &
\end{array}
$$

Performing a partial evaluation of A_n and $\neg\_A_n$ on the rules $A \leftarrow A_n$ and $\neg A \leftarrow \neg\_A_n$, and removing the rules for those atoms (for we don't care about them in the final models and as they don't appear in any other rules); also performing a partial evaluation of A_n^U and $\neg\_A_n^U$ on the last two rules we obtain $P_S(n)'$:

$$
\begin{array}{ll}
P_S(n)': & \\
\left.\begin{array}{l} A_0 \leftarrow A_0^U \\ \neg\_A_0 \leftarrow \neg\_A_0^U \end{array}\right\} \text{for all } A \text{ in } \mathcal{L} &
\left.\begin{array}{l} A_t^U \leftarrow B_t, not\ C_t \\ \neg\_A_t^U \leftarrow B_{\neg t}, not\ C_{\neg t} \end{array}\right\} \begin{array}{l} \text{rules from } U_t \\ t \leq n-1 \end{array} \\
\left.\begin{array}{l} A_t \leftarrow A_{t-1}, not\ \neg\_A_t^U \\ \neg\_A_t \leftarrow \neg\_A_{t-1}, not\ A_t^U \\ A_t \leftarrow A_t^U \\ \neg\_A_t \leftarrow \neg\_A_t^U \end{array}\right\} \begin{array}{l} \text{for all } A \text{ in } \mathcal{L} \\ 0 < t \leq n-1 \end{array} &
\left.\begin{array}{l} A \leftarrow A_{n-1}, not\ \neg\_A_n^U \\ \neg A \leftarrow \neg\_A_{n-1}, not\ A_n^U \\ A \leftarrow B_n, not\ C_n \\ \neg A \leftarrow B_{\neg n}, not\ C_{\neg n} \end{array}\right\} \text{for all } A \text{ in } \mathcal{L} \\
\left.\begin{array}{l} A_n^U \leftarrow B_n, not\ C_n \\ \neg\_A_n^U \leftarrow B_{\neg n}, not\ C_{\neg n} \end{array}\right\} \text{rules from } U_n &
\end{array}
$$

The updated program of $P_S(n-1)$ by U_n, P_U is by Def. 4.4.1:

$$
P_U: \quad
\begin{array}{ll}
\left.\begin{array}{l}
(A_0)' \leftarrow A_0^U \\
(\neg\_A_0)' \leftarrow \neg\_A_0^U
\end{array}\right\} & \text{rules from } P_S(n-1) \\
\left.\begin{array}{l}
(A_t^U)' \leftarrow B_t, not\ C_t \\
(\neg\_A_t^U)' \leftarrow B_{\neg t}, not\ C_{\neg t}
\end{array}\right\} & \text{rules from } P_S(n-1), t \leq n-1 \\
\left.\begin{array}{l}
(A_t)' \leftarrow A_{t-1}, not\ \neg\_A_t^U \\
(\neg\_A_t)' \leftarrow \neg\_A_{t-1}, not\ A_t^U \\
(A_t)' \leftarrow A_t^U \\
(\neg\_A_t)' \leftarrow \neg\_A_t^U
\end{array}\right\} & \text{rules from } P_S(n-1), t \leq n-1 \\
\left.\begin{array}{l}
A' \leftarrow A_{n-1} \\
\neg\_A' \leftarrow \neg\_A_{n-1}
\end{array}\right\} & \text{rules from } P_S(n-1) \\
\left.\begin{array}{l}
A^U \leftarrow B_n, not\ C_n \\
\neg A^U \leftarrow B_{\neg n}, not\ C_{\neg n}
\end{array}\right\} & \text{rules from } U_n \\
\left.\begin{array}{l}
A \leftarrow A', not\ \neg A^U \\
\neg A \leftarrow \neg\_A', not\ A^U \\
A \leftarrow A^U \\
\neg A \leftarrow \neg A^U
\end{array}\right\} & \text{for all } A \text{ in } \mathcal{L}^*
\end{array}
$$

Note that, although $P_S(n-1)$ has explicitly negated atoms in the body of its rules, which would make Def.4.4.1 not applicable, they are all of the form $\neg\_A_{t-1}$ and $\neg\_A_t^U, t \leq n-1$. Since they don't belong to U_n, and there aren't any atoms of the form $\_A_t^U$, nor $\_A_{t-1}$, we could simply rename them (as positive atoms), before applying the update transformation of Def.4.4.1. We opt for their initial designation to make things clearer, now that the applicability of the update transformation has been shown.

Since there are no rules for atoms of the form $(A_t)^U$, we can partially evaluate all primed atoms that don't belong to $\mathcal{L}$ over the rules $A \leftarrow A', not\ \neg A^U$ and $\neg A \leftarrow \neg\_A', not\ A^U$, remove the rules for $(A_t)'$, and restrict the last for rules to all A in $\mathcal{L}$. Also, we perform a partial evaluation of A' and $\neg\_A'$ on the same rules, and remove the rules for A' and $\neg\_A'$; finally we perform a partial evaluation of A^U and $\neg A^U$ on the rules $A \leftarrow A^U$ and

$\neg A \leftarrow \neg A^U$, to obtain:

$$P'_U: \quad \left.\begin{array}{l} A_0 \leftarrow A_0^U \\ \neg\_A_0 \leftarrow \neg\_A_0^U \end{array}\right\} \text{for all } A \text{ in } \mathcal{L}$$
$$\left.\begin{array}{l} A_t^U \leftarrow B_t, not\ C_t \\ \neg\_A_t^U \leftarrow B_{\neg t}, not\ C_{\neg t} \end{array}\right\} t \leq n-1$$
$$\left.\begin{array}{l} A_t \leftarrow A_{t-1}, not\ \neg\_A_t^U \\ \neg\_A_t \leftarrow \neg\_A_{t-1}, not\ A_t^U \\ A_t \leftarrow A_t^U \\ \neg\_A_t \leftarrow \neg\_A_t^U \end{array}\right\} t \leq n-1$$
$$\left.\begin{array}{l} A^U \leftarrow B_n, not\ C_n \\ \neg A^U \leftarrow B_{\neg n}, not\ C_{\neg n} \end{array}\right\} \text{rules from } U_n$$
$$\left.\begin{array}{l} A \leftarrow A_{n-1}, not\ \neg A^U \\ \neg A \leftarrow \neg\_A_{n-1}, not\ A^U \\ A \leftarrow B_n, not\ C_n \\ \neg A \leftarrow B_{\neg n}, not\ C_{\neg n} \end{array}\right\} \text{for all } A \text{ in } \mathcal{L}$$

which is equal to $P_S(n)'$, apart from the renaming of atoms introduced by the transformation, which we do not care about. ■

From this theorem there immediately follows a corollary, establishing that the iterated program updates generalize the program updates.

Corollary 5.3.2 (Iterated Program Updates generalize Program Updates) *Let P and U be update programs over $\mathcal{L}_\neg$. Let $T = \{0, 1\}$ and let $U_0 = P$ and $U_1 = U$. Then the iterated update program $P_S(1)$ at state $t = 1$ is semantically equivalent to the updated program P_U of P by U.*

The next theorem establishes the relationship between the iterated program and the S-justified updates at a given state t.

Theorem 5.3.3 (Correctness of the iterated update program at a state t) *Let $S = \{U_t : t \in T\}$ be an ordered sequence of update logic programs U_t over $\mathcal{L}_\neg$, with the smallest element U_{t_0}. Restricted to $\mathcal{L}$, the answer-sets of the iterated update program at a given current state t, are exactly the S-justified updates at state t.*

To prove the theorem we make use of the lemmas:

Lemma 5.3.4 (Correctness of the iterated update program at state $t = 0$) *Let $S = \{U_t : t \in T\}$ be an ordered sequence of update logic programs U_t over $\mathcal{L}_\neg$, with the* *smallest element U_{t_0}. Restricted to $\mathcal{L}$, the answer-sets of the iterated update program at state t_0, are exactly the S-justified updates at state t_0.*

Proof. According to Def.5.2.2, an interpretation is an S-justified update iff it is an answer-set of $P_S^*(0) = U_0$. Together with theorem 4.3.1 and corollary 5.3.2, the lemma then follows. ■

Lemma 5.3.5 (Correctness of the iterated update program at state $t = 1$) *Let P and U be update programs over $\mathcal{L}_\neg$. Let $T = \{0, 1\}$ and $S = \{U_0, U_1\}$ such that $U_0 = P$ and $U_1 = U$. Restricted to $\mathcal{L}$, the answer-sets of the iterated update program are exactly the S-justified updates at state $t = 1$.*

Proof. Given the equivalence between S-justified updates at state $t = 1$ and <P,U>-justified update (Cor.5.2.2) and the equivalence between the iterated update program at state $t = 1$ and the updated program P_U of P by U (Cor.5.3.2), the lemma follows from theorem 4.4.1. ■

Proof. (of theorem 5.3.3)We have to prove that the answer-sets of $P_S(t)$ are exactly the S-justified updates at state t, when restricted to $\mathcal{L}$. Proof by induction: For $t = 0$, it is proved by lemma 5.3.4. For $t = 1$, it is proved by lemma 5.3.5.

Assuming the correctness of the theorem for $t = n$, i.e. the answer-sets of $P_S(n)$ are exactly the S-justified updates at state n, let us prove it for $t = n + 1$. Let $S = \{U_0, ..., U_{n-1}, U_n, U_{n+1}\}$be an ordered sequence of update logic programs U_t over $\mathcal{L}_\neg$, with the smallest element U_0. Let $P \uplus U$ denote the updated program of P by U. By theorem 5.3.1, we have that $P_S(n + 1)$ is semantically equivalent to $P_S(n) \uplus U_{n+1}$. By theorem 5.2.1 we know that the S-justified updates at state $n + 1$ are exactly the <$P_S^*(n)$,U_{n+1}>-justified updates. Finally, by theorem 4.4.1, we know that the answer-sets of $P_S(n) \uplus U_{n+1}$ are exactly the <P,U>-justified updates of $P_S(n)$ by U_{n+1}. ■

In chapter 8 an illustrative example of iterated updating is provided.

Chapter 6

Updating Normal Logic Programs under Partial Stable Semantics

In this chapter we extend the notions of logic program updating to the case where a partial stable semantics is used. To bring out the implications of such an extension in a simple way, it will just be presented for the case of normal logic programs, and of a single update. The extension to the iterated case, is similar to the one presented in the previous chapter.

As will be seen, we need only extend the characterization of <P,U>-justified updates. The update transformation used before for normal logic programs is valid for the three-valued case simply by using the partial stable semantics to compute models.

As for 2-valued interpretations, we need to relate 3-valued (or partial) interpretations over a language $\mathcal{L}_\neg$, with 3-valued interpretations over $\mathcal{L}$. For this we define:

Definition 6.0.3 (3-valued Interpretation Restriction) *Given a language $\mathcal{L}$ that does not contain explicit negation $\neg$. Let $M_\neg = \langle T_{M_\neg}, F_{M_\neg} \rangle$ be a 3-valued interpretation, over the language $\mathcal{L}_\neg$, obtained by augmenting $\mathcal{L}$ with the set $E = \{\neg A : A \in \mathcal{L}\}$.*

We define the corresponding 3-valued restricted interpretation *M, over $\mathcal{L}$, as:*

$$M = \langle T_M, F_M \rangle$$

where:

$$T_M = T_{M_\neg} \text{ restricted to } \mathcal{L}$$
$$F_M = F_{M_\neg} \text{ restricted to } \mathcal{L} \ \diamondsuit$$

For simplicity, we will also represent 3-valued interpretations as sets of objective and default literals.

The translation of update programs into logic programs is the same as for the 2-valued case, as per Def. 4.2.3.

6.1 Three-valued <P,U>-Justified Updates

The characterization of <P,U>-justified updates will have to suffer a modification to allow for partial iterpretations and models. To better understand the need for such modifications, and the intuition behind them, let us look at a very simple example:

Example 6.1.1 *Consider the following initial program P and update program U (next to each rule of U we exhibit the original update rule):*

$$P: \quad a \leftarrow \qquad U: \quad \begin{array}{ll} \neg a \leftarrow c & [out(a) \leftarrow in(c)] \\ c \leftarrow not\ c & [in(c) \leftarrow out(c)] \end{array}$$

Any 3-valued U-justified update of P should be a 3-valued <P,U>-justified update as well.because P contains just facts. According to Def.3.2.2, the following interpretation M is the only 3-valued U-justified update:

$$M = \{\}$$

i.e., both **a** *and* **c** *are undefined. Looking at the rules of both P and U we can note that it is impossible to have a partial stable model (PSM) where* **a** *is undefined, in a program composed of those rules. This is so because having* $\neg$**a** *undefined doesn't affect the truth value of* **a**.

The previous example suggests that the characterization of 3-valued <P,U>-justified updates can no longer be made to depend on a composition of the rules of P and U alone. We will have, under certain conditions, to explicitly undefine some objective literals. Informally, this should be done whenever a literal L is true due to a rule of P but $\neg L$ is undefined due to a rule of U. Formally we have that:

Definition 6.1.1 (Three-valued Inertial Sub-Program) *Let P be a normal logic program over* $\mathcal{L}_\neg$*, U an update program over* $\mathcal{L}_\neg$ *and* $M_\neg = \langle T_{M_\neg}, F_{M_\neg} \rangle$ *a 3-valued interpretation over* $\mathcal{L}_\neg$*. Let:*

$$\begin{array}{rl} Rejected(M_\neg, P, U) = & \{L \leftarrow body \in P : \hat{M}_\neg(body) = 1 \\ & and\ \exists \neg L \leftarrow body' \in U : \hat{M}_\neg(body') \neq 0\ \} \end{array} \tag{6.1}$$

where L is an objective literal. We define Three-valued Inertial Sub-Program $P_{inertial}(M_\neg, P, U)$ *as:*

$$P_{inertial}(M_\neg, P, U) = P - Rejected(M_\neg, P, U)\ \Diamond$$

Intuitively, the rules for some objective literal L that belong to $Rejected(M_\neg, P, U)$ are those that belong to the initial program but, although their body is still verified by the interpretation, there is an update rule that overrides them, by either truthifying or undefining its complementary literal.

Now we define the set of literals that are to be explicitly undefined:

Definition 6.1.2 (Undefining Rules) *Let P be a normal logic program over $\mathcal{L}_\neg$, U an update program over $\mathcal{L}_\neg$ and $M_\neg = \langle T_{M_\neg}, F_{M_\neg} \rangle$ a 3-valued interpretation over $\mathcal{L}_\neg$. Then, the set of* Undefining Rules, *$Undefining(M_\neg, P, U)$ is defined as:*

$$\begin{aligned} Undefining(M_\neg, P, U) = \ & \{L \leftarrow \mathbf{u} : L \leftarrow body \in Rejected(M_\neg, P, U) \\ & \ and\ \nexists \neg L \leftarrow body' \in U : \hat{M}_\neg(body') = 1\ \} \ \diamondsuit \end{aligned}$$

$Undefining(M_\neg, P, U)$ contains rules to undefine precisely those literals L such that there is an update rule undefining its complementary literal $\neg L$, and none truthifying it, which are required to reject rules for L. Recall that $\mathbf{u}$ stands for the propositional symbol with the property of being undefined in every interpretation.

Finally, the three-valued <P,U>-justified updates are characterized as follows:

Definition 6.1.3 (Three-valued <P,U>-Justified Updates) *Let P be a normal logic program, U an update program, and M a 3-valued interpretation over the language of P. M is a* Three-valued <P,U>-Justified Update *of P updated by U, iff $M_\neg$ is a partial stable model of $P^*(M_\neg, P, U)$, where*

$$P^*(M_\neg, P, U) = U + P_{inertial}(M_\neg, P, U) + Undefining(M_\neg, P, U) \ \diamondsuit$$

The next example shows the rôle played by $Undefining(M_\neg, P, U)$ when determining the 3-valued <P,U>-Justified Updates.

Example 6.1.2 *Consider the pacifist situation of example 4.3.1 where P and U were:*

$$\begin{array}{llll} P: & pacifist \leftarrow & U: & \neg pacifist \leftarrow war \\ & reasonable \leftarrow pacifist & & peace \leftarrow not\ war \\ & & & war \leftarrow not\ peace \end{array}$$

Intuitively, examining the two 2-valued $<P,U>$-justified updates:

$$\begin{aligned} M_1 &= \{pacifist, reasonable, peace, not\ war\} \\ M_2 &= \{not\ pacifist, not\ reasonable, not\ peace, war\} \end{aligned}$$

one should expect the interpretation $M_3 = \{\}$ *to be a 3-valued* $<P,U>$*-justified update. Let's check whether it is a 3-valued* $<P,U>$*-justified update.* M_3 *is* $M_{\neg 3}$ *restricted to the language of* P*:*

$$M_{\neg 3} = \quad \{\}$$

Since the truth value of war in the rule $\neg pacifist \leftarrow war$ *is not false, we have that the rule* $pacifist \leftarrow$ *should be rejected:*

$$Rejected(M_{\neg 3}, P, U) = \{pacifist \leftarrow\}$$

Now, we have to include an undefining rule for $pacifist$ *just in case there aren't any rules in* U *for* $\neg pacifist$ *with a true body. Since that is the case,*

$$Undefining(M_{\neg 3}, P, U) = \{pacifist \leftarrow \mathbf{u}\}$$

and we obtain the program:

$$P^*(M_{\neg 3}, P, U) = U + (P - \{pacifist \leftarrow\}) + \{pacifist \leftarrow \mathbf{u}\}$$

i.e.:

$$\begin{array}{llll} P^*(M_{\neg 3}, P, U): & pacifist \leftarrow \mathbf{u} & & \neg pacifist \leftarrow war \\ & reasonable \leftarrow pacifist & & peace \leftarrow not\ war \\ & & & war \leftarrow not\ peace \end{array}$$

Since $M_{\neg 3}$ *is a PSM of* $P^*(M_{\neg 3}, P, U)$*,* M_3 *is a three-valued* $<P,U>$*-justified update.*

The theorem below shows that the (2-valued) <P,U>-justified updates are a special case of the 3-valued <P,U>-justified updates:

Theorem 6.1.1 (Generalization of <P,U>-justified updates) *The definition of 3-valued* $<P,U>$*-justified update reduces to that of (2-valued)* $<P,U>$*-justified update whenever interpretations are total.*

Proof. Let $M_\neg$ be a 2-valued interpretation: Then, in (6.1), condition $\hat{M}_\neg(body') \neq 0$ is equivalent to $M_\neg \models body'$ for if $body'$ isn't false, it can only be true. This shows that, whenever $M_\neg$ is a total interpretation, (6.1) reduces to the definition of $Rejected(M_\neg, P, U)$ for the 2-valued case. Then, for every rejected rule there must be a rule of the form $\neg L \leftarrow body'$, in U, such that $\hat{M}_\neg(body') = 1$, i.e. $M_\neg \models body'$. This way, the set $Undefining(M_\neg, P, U)$ will always be empty, and both $P^*(M_\neg, P, U)$ of the 3-valued <P,U>-justified update definition and $P^*(M_\neg, P, U)$ of the (2-valued) <P,U>-justified update definition are equivalent. ■

The 3-valued <P,U>-justified updates can be determined from the updated program constructed from the initial program and the update program, as per Def. 4.4.1, if we consider its partial stable models (or WFSX models).

Example 6.1.3 *Consider P and U of Example 6.1.2. The updated program P_U of P by U is:*

$$\begin{array}{ll} pacifist' \leftarrow & A \leftarrow A', not\ \neg A^U \\ reasonable' \leftarrow pacifist & \neg A \leftarrow \neg\_A', not\ A^U \\ \neg pacifist^U \leftarrow war & A \leftarrow A^U \\ peace^U \leftarrow not\ war & \neg A \leftarrow \neg A^U \\ war^U \leftarrow not\ peace & \end{array}$$

whose partial stable models (modulo A', A^U and explicitly negated atoms) are:

$$\begin{array}{l} M_1 = \{pacifist, reasonable, peace, not\ war\} \\ M_2 = \{not\ pacifist, not\ reasonable, not\ peace, war\} \\ M_3 = \{\} \end{array}$$

coinciding with the three 3-valued <P,U>-justified updates determined in Example 6.1.2.

The following theorem establishes the relationship between the partial stable models of the update transformation of a program and its 3-valued <P,U>-justified updates.

Theorem 6.1.2 (Correctness of the update transformation) *Let P be a normal logic program over $\mathcal{L}_\neg$ and U an update program over $\mathcal{L}_\neg$. Modulo any primed, L^U and explicitly negated literals, the partial stable models of the updated program P_U are exactly the 3-valued <P,U>-Justified Updates of P updated by U.*

Proof. Let P be a normal logic program consisting of rules of the form:

$$\begin{array}{l} A \leftarrow B_i, not\ C_i \\ \neg A \leftarrow B_m, not\ C_m \end{array}$$

and U an update program consisting of rules of the form:

$$\begin{array}{l} A \leftarrow B_j, not\ C_j \\ \neg A \leftarrow B_k, not\ C_k \end{array}$$

where A is an atom and each B and C is some finite set of atoms.

Let $P_U^*(M_\neg, P, U)$ be the program obtained according to Def. 6.1.3:

$$P_U^*(M_\neg, P, U) = U + P_{inertial}(M_\neg, P, U) + Undefining(M_\neg, P, U)$$

and note that $P_{inertial}(M_\neg, P, U) \subseteq P$.

Let P_U be the program obtained according to Def. 4.4.1:

$$P_U : \begin{array}{ll} \left.\begin{array}{l} A' \leftarrow B_i, not\ C_i \\ \neg\_A' \leftarrow B_m, not\ C_m \end{array}\right\} & \text{for all rules from } P \\ \left.\begin{array}{l} A \leftarrow A', not\ \neg A^U \\ \neg A \leftarrow \neg\_A', not\ A^U \\ A \leftarrow A^U \\ \neg A \leftarrow \neg A^U \end{array}\right\} & \text{for all } A \\ \left.\begin{array}{l} A^U \leftarrow B_j, not\ C_j \\ \neg A^U \leftarrow B_k, not\ C_k \end{array}\right\} & \text{for all rules from } U \end{array}$$

We will show that P_U is equivalent to $P_U^*(M_\neg, P, U)$ for our purposes. Performing on P_U a partial evaluation of A^U and $\neg A^U$ on the rules $A \leftarrow A^U$ and $\neg A \leftarrow \neg A^U$ we obtain:

$$\begin{array}{lll} P_U' : & A' \leftarrow B_i, not\ C_i & (1) \\ & \neg\_A' \leftarrow B_m, not\ C_m & (2) \\ & A \leftarrow A', not\ \neg A^U & (3) \\ & \neg A \leftarrow \neg\_A', not\ A^U & (4) \\ & A \leftarrow B_j, not\ C_j & (5) \\ & \neg A \leftarrow B_k, not\ C_k & (6) \\ & A^U \leftarrow B_j, not\ C_j & (7) \\ & \neg A^U \leftarrow B_k, not\ C_k & (8) \end{array}$$

Note that rules (5) and (6) are exactly the update program.

These rules can be simplified. In particular we don't need the rules for A^U and $\neg A^U$. Since we've already considered the cases where neither A^U nor $\neg A^U$ are undefined, in Theorem 4.4.1, (as we've seen in Theorem 6.1.1, $P_{inertial}(M_\neg, P, U)$ is the same and $Undefining(M_\neg, P, U)$ is empty), we will only consider the case where at least one of A^U or $\neg A^U$ are undefined.

This way, for some arbitrary A, consider first the case where both A^U and $\neg A^U$ are undefined. We can then perform the following simplifications on P_U': replace in (3) (resp. (4)) A' (resp.$\neg\_A'$) by the body of (1) (resp. (2)) and replace $not\ \neg A^U$ (resp.$not\ A^U$) by $\mathbf{u}$ (recall that $\mathbf{u}$ is undefined in every interpretation) to obtain (3*) (resp. (4*)): $A \leftarrow B_i, not\ C_i, \mathbf{u}$ (resp.$\neg A \leftarrow B_m, not\ C_m, \mathbf{u}$) ; now we no longer need rules (7) and (8). Since we don't care about primed literals in the updated models we can remove rules (1) and (2). The so mutilated P_U' preserves the semantics of P_U' when both A^U and $\neg A^U$ are undefined, apart primed and U literals, and looks like this:

$$\begin{array}{ll} A \leftarrow B_i, not\ C_i, \mathbf{u} & (3*) \\ \neg A \leftarrow B_m, not\ C_m, \mathbf{u} & (4*) \\ A \leftarrow B_j, not\ C_j & (5) \\ \neg A \leftarrow B_k, not\ C_k & (6) \end{array}$$

which is semantically equivalent to $P_U^*(M_\neg, P, U)$ when both A^U and $\neg A^U$ are undefined. Indeed, note that in $P_U^*(M_\neg, P, U)$, instead of (3*) (resp.(4*)), we would have $A \leftarrow B_i, not\ C_i$ (resp. $\neg A \leftarrow B_m, not\ C_m$) if $B_i, not\ C_i$ (resp $B_m, not\ C_m$) is false or undefined, and $A \leftarrow \mathbf{u}$ (resp. $\neg A \leftarrow \mathbf{u}$) if $B_i, not\ C_i$ (resp $B_m, not\ C_m$) is true, but the effect is exactly the same and hence their partial stable models are the same in that case.

For the case where $\neg A^U$ is undefined and A^U is false, we can replace in (3) (resp. (4)) A' (resp.$\neg\_A'$) by the body of (1) (resp. (2)) and replace $not\ \neg A^U$ by $\mathbf{u}$ and remove $not\ A^U$, to obtain (3*) and (4*): $A \leftarrow B_i, not\ C_i, \mathbf{u}$ and $\neg A \leftarrow B_m, not\ C_m$ respectively; now we no longer need rules (7) and (8). Since we don't care about primed literals in the updated models we can remove rules (1) and (2). The so mutilated $P_U^{'}$ preserves the semantics of $P_U^{'}$ when $\neg A^U$ is undefined and A^U is false, apart primed and U literals, and looks like this:

$$\begin{array}{ll} A \leftarrow B_i, not\ C_i, \mathbf{u} & (3*) \\ \neg A \leftarrow B_m, not\ C_m & (4*) \\ A \leftarrow B_j, not\ C_j & (5) \\ \neg A \leftarrow B_k, not\ C_k & (6) \end{array}$$

which, for reasons presented before, is semantically equivalent to $P_U^*(M_\neg, P, U)$.

For the case where A^U is undefined and $\neg A^U$ is false, the proof is similar to the previous case. ■

The next theorem establishes the relationship between program update and interpretation update. To this end we begin by defining a transformation from a 3-valued interpretation into the arguably simplest normal logic program that encodes it.

Definition 6.1.4 (3-valued factual LP) *Let $I = \langle T, F \rangle$ be an interpretation over a language $\mathcal{L}$. We define the normal logic program associated with I, $P^*(I)$, as:*

$$P^*(I) = \{L \leftarrow : L \in T\} \cup \{L \leftarrow \mathbf{u} : L \notin T \cup F\} \ \diamondsuit$$

Definition 6.1.5 (Closeness Relationship for 3-valued Interpretations) *[1]Given three partial interpretations $I_1 = \langle T_1, F_1 \rangle$, $I_2 = \langle T_2, F_2 \rangle$ and $I_3 = \langle T_3, F_3 \rangle$, we say that I_3 is closer to I_1 than I_2 if*

$$(T_2 \setminus T_1 \cup T_1 \setminus T_2) \subset (T_3 \setminus T_1 \cup T_1 \setminus T_3)$$

and

$$(F_2 \setminus F_1 \cup F_1 \setminus F_2) \subset (F_3 \setminus F_1 \cup F_1 \setminus F_3) \ \diamondsuit$$

Theorem 6.1.3 (Generalization of 3-valued Updates) *Let U be an update program and $I = \langle T, F \rangle$ a 3-valued interpretation. Then:*

1. *Every 3-valued U-justified update of I is a $<P^*(I), U>$-justified update.*
2. *A $<P^*(I), U>$-justified update I_u is a U-justified update of I iff*

$$\not\exists I_u^{'} \text{ closer to } I \text{ than } I_u$$

where $I_u^{'}$ is a $<P^*(I), U>$-justified update. $\diamondsuit$

Proof.

Let U be an update program consisting of rules of the form:

$$A \leftarrow B_j, not\ C_j$$
$$\neg A \leftarrow B_k, not\ C_k$$

where A is an atom and each B and C is some finite set of atoms.

According to [5], an interpretation I_u is a 3-valued U-justified update of I iff it is a PSM (modulo primed and explicitly negated elements) of the corresponding program P_U:

$$\begin{array}{lll} P_U: & A' \leftarrow & \text{for all } A \in T \\ & A' \leftarrow \mathbf{u} & \text{for all } A \notin T \cup F \\ & \left.\begin{array}{l} A \leftarrow A', not\ \neg A \\ \neg A \leftarrow not\ A', not\ A \end{array}\right\} & \text{for all } A \\ & \left.\begin{array}{l} A \leftarrow B_j, \neg C_j \\ \neg A \leftarrow B_k, \neg C_k \end{array}\right\} & \text{for all rules from } U \end{array}$$

According to Def. 4.4.1, an interpretation $I_u^{'}$ is a $<P^*(I), U>$-justified update iff it is the restriction to the language of I of a PSM of the program $P_U^{'}$ (we ommit the inertia rules of the form $\neg A \leftarrow \neg\_A', not\ A^U$ because there are no rules for $\neg\_A'$):

$$\begin{array}{lll} P_U^{'}: & A' \leftarrow & \text{for all } A \in T \\ & A' \leftarrow \mathbf{u} & \text{for all } A \notin T \cup F \\ & \left.\begin{array}{l} A \leftarrow A', not\ \neg A^U \\ A \leftarrow A^U \\ \neg A \leftarrow \neg A^U \end{array}\right\} & \text{for all } A \\ & \left.\begin{array}{l} A^U \leftarrow B_j, not\ C_j \\ \neg A^U \leftarrow B_k, not\ C_k \end{array}\right\} & \text{for all rules from } U \end{array}$$

Again, notice the difference in the translation of update rules in what the kind of negation used in their bodies is concerned. We will show that for every PSM I_u of the program P_U, there is a PSM $I_u^{'}$ of $P_U^{'}$ such that $I_u = I_u^{'}$ restricted to the language of I.

Performing a partial evaluation of A^U and $\neg A^U$ on the rules $A \leftarrow A^U$ and $\neg A \leftarrow \neg A^U$ we obtain:

$$
\begin{array}{rll}
P_U^{'}: & A' \leftarrow & (1) \\
& A' \leftarrow \mathbf{u} & (2) \\
& A \leftarrow A', not\ \neg A^U & (3) \\
& A \leftarrow B_j, not\ C_j & (4) \\
& \neg A \leftarrow B_k, not\ C_k & (5) \\
& A^U \leftarrow B_j, not\ C_j & (6) \\
& \neg A^U \leftarrow B_k, not\ C_k & (7)
\end{array}
$$

We can safely replace $not\ \neg A^U$ by $not\ \neg A$ in rule (3), for the only rules for $\neg A$ and $\neg A^U$ have the same body. Now, and since we don't care about A^U and $\neg A^U$ in the updated models, we can remove rules (6) and (7) to obtain program $P_U^{''}$:

$$
\begin{array}{rllrl}
P_U^{''}: & A' \leftarrow & (1) & P_U: & A' \leftarrow \\
& A' \leftarrow \mathbf{u} & (2) & & A' \leftarrow \mathbf{u} \\
& A \leftarrow A', not\ \neg A & (3) & & A \leftarrow A', not\ \neg A \\
& & (4) & & \neg A \leftarrow not\ A', not\ A \\
& A \leftarrow B_j, not\ C_j & (5) & & A \leftarrow B_j, \neg C_j \\
& \neg A \leftarrow B_k, not\ C_k & (6) & & \neg A \leftarrow B_k, \neg C_k
\end{array}
$$

It is easy to see that the only differences between $P_U^{''}$ and P_U are in the kind of negation used in the body of the rules from the update program, and the extra rule (4) in P_U.

We will now remove all primed atoms of both programs. By taking I into account, and for an arbitrary A we have:

For the case where $A \in T$, we obtain:

$$
\begin{array}{rllrl}
P_U^{''}: & A \leftarrow not\ \neg A & (3*) & P_U: & A \leftarrow not\ \neg A \\
& A \leftarrow B_j, not\ C_j & (5) & & A \leftarrow B_j, \neg C_j \\
& \neg A \leftarrow B_k, not\ C_k & (6) & & \neg A \leftarrow B_k, \neg C_k
\end{array}
$$

In this case, since by coherence if $\neg C_k$ is true in a model of P_U, then so must be $not\ C_k$, every such model is also a model of $P_U^{''}$. Also, every model of $P_U^{''}$ that wasn't a model of P_U would be such that $not\ A$ was true or undefined, and there is also a model of both in which A is true; but these models would be eliminated by the closeness condition since A was initially true.

For the case where $A \in F$, we obtain:

$$\begin{array}{llll} P_U^{''}: & & (4*) & P_U: \neg A \leftarrow not\ A \\ & A \leftarrow B_j, not\ C_j & (5) & \quad A \leftarrow B_j, \neg C_j \\ & \neg A \leftarrow B_k, not\ C_k & (6) & \quad \neg A \leftarrow B_k, \neg C_k \end{array}$$

In this case, because of rule (4*) in P_U, the models coincide.

For the case where $A \notin T \cup F$, we obtain:

$$\begin{array}{llll} P_U^{''}: & A \leftarrow \mathbf{u}, not\ \neg A & (3*) & P_U: A \leftarrow \mathbf{u}, not\ \neg A \\ & & (4*) & \quad \neg A \leftarrow \mathbf{u}, not\ A \\ & A \leftarrow B_j, not\ C_j & (5) & \quad A \leftarrow B_j, \neg C_j \\ & \neg A \leftarrow B_k, not\ C_k & (6) & \quad \neg A \leftarrow B_k, \neg C_k \end{array}$$

If A' is undefined, again every model of P_U is also a model of $P_U^{''}$ (note that we're only concerned about the restrictions of the actual models, i.e., P_U could have a model in which both A and $\neg A$ were undefined and the corresponding model of $P_U^{''}$ has A undefined and $\neg A$ false, but their restrictions coincide). As for models of $P_U^{''}$ that aren't models of P_U, they would have to be such that A was false ($\neg A$ was true) but there is also a model of both in which A was undefined. Again, the closeness condition eliminates such models. ■

As for the 2-valued case, the closeness condition is needed to eliminate one of those pairs of models that arise in $P_U^{''}$ due to the translation of the original out's into not's in the body of update rules.

Again, the notion of 3-valued program update presented here is a generalization of the 3-valued updates carried out on a model basis. Consequently, the program transformation above is a generalization of the program transformation in [5].

Chapter 7

Extended Logic Program Updating

In recent years, much work has been done on representing knowledge by using extended logic programs, and so we want to be able to update such programs. To do so, we need to extend the notions presented before, for the case where both the initial program and the update program are defined over a language that contains explicit negation from the start. We will detail the case of a single update under a 3-valued semantics. These results can be extended, as for normal logic programs, for the iterated case. The iterated update transformation will be presented in the last section of this chapter.

The translation of extended update programs into extended logic programs is per Def. 3.3.3. Since this transformation extends the language with new literals, we start by relating the interpretations over this extended language with those over the original one.

Definition 7.0.6 (3-valued Interpretation Restriction) *Given a language $\mathcal{K}$ with explicit negation $\neg$. Let $M_{np} = \langle T_{M_{np}}, F_{M_{np}} \rangle$ be a 3-valued interpretation, over the language $\mathcal{K}_{np}$, obtained by augmenting $\mathcal{K}$ with the set $E = \{L^n, L^p : L \in K\}$(where L^n, L^p and L are objective literals).*

We define the corresponding 3-valued restricted interpretation *M, over $\mathcal{K}$, as:*

$$M = \langle T_M, F_M \rangle$$

where:

$$T_M = \{L : L \in T_{M_{np}} \text{ and } L \in \mathcal{K}\}$$
$$F_M = \{L : L \in F_{M_{np}} \text{ and } L \in \mathcal{K}\} \quad \diamondsuit$$

For simplicity, we will also represent 3-valued interpretations as sets of literals.

7.1 Extended <P,U>-Justified Updates

Like for normal logic programs, we start by defining the rules of the initial program that are rejected due to the rules of the update program.

Definition 7.1.1 (Extended Inertial Sub-Program) *Let P be an extended logic program over $\mathcal{K}$, U an update program over $\mathcal{K}_{np}$ and $M_{np} = \langle T_{M_{np}}, F_{M_{np}} \rangle$ a 3-valued interpretation over $\mathcal{K}_{np}$. Let:*

$$\begin{aligned} Rejected(M_{np}, P, U) = \; & \{A \leftarrow body \in P : \hat{M}_{np}(body) = 1 \\ & and\ \exists \neg A^p \leftarrow body' \in U : \hat{M}_{np}(body') \neq 0\ \} \cup \\ & \cup \{\neg A \leftarrow body \in P : \hat{M}_{np}(body) = 1 \\ & and\ \exists \neg A^n \leftarrow body' \in U : \hat{M}_{np}(body') \neq 0\} \end{aligned}$$

where A is an atom. We define Extended Inertial Sub-Program $P_{inertial}(M_{np}, P, U)$ *as:*

$$P_{inertial}(M_{np}, P, U) = P - Rejected(M_{np}, P, U) \ \diamondsuit$$

Again, the rules for some objective literal L that belong to $Rejected(M_{np}, P, U)$ are those that belong to the initial program but, although their body is still verified by the model, there is an update rule that overrides them, by contravening their conclusion. Note that a rule of P for atom A, with true body, is also countervened by a rule of U with true or undefined body for A^n (i.e. one translated from $in(\neg A)$). Since every U also contains the rules $\neg A^p \leftarrow \neg A$ and $\neg A \leftarrow A^n$, then $\neg A$ in $\neg A^p \leftarrow \neg A$ is also true or undefined, and so that rule of P is rejected in this case too. Similarly for a rule of P with head $\neg A$, but now with respect to A^p.

Now, similarly to the case of normal logic programs, we have to define the set of literals that must be explicitly undefined:

Definition 7.1.2 (Extended Undefining Rules) *Let P be an extended logic program over $\mathcal{K}$, U an update program over $\mathcal{K}_{np}$ and $M_{np} = \langle T_{M_{np}}, F_{M_{np}} \rangle$ a 3-valued interpretation over $\mathcal{K}_{np}$. Then, the* Extended Undefining Rules*, $Undefining(M_{np}, P, U)$ is defined as:*

$$\begin{aligned} Undefining(M_{np}, P, U) = \; & \{A \leftarrow \mathbf{u} : A \leftarrow body \in Rejected(M_{np}, P, U) \\ & and\ \nexists \neg A^p \leftarrow body' \in U : \hat{M}_{np}(body') = 1\ \} \cup \\ & \{\neg A \leftarrow \mathbf{u} : \neg A \leftarrow body \in Rejected(M_{np}, P, U) \\ & and\ \nexists \neg A^n \leftarrow body' \in U : \hat{M}_{np}(body') = 1\ \} \ \diamondsuit \end{aligned}$$

And the extended <P,U>-justified updates are characterized as follows:

Definition 7.1.3 (Extended <P,U>-Justified Updates) *Let P be an extended logic program, U an update program, and M a 3-valued interpretation over the language of P. M is a* Extended <P,U>-Justified Update *of P updated by U iff M_{np} is a partial stable model of $P^*(M_{np}, P, U)$, where*

$$P^*(M_{np}, P, U) = U + P_{inertial}(M_{np}, P, U) + Undefining(M_{np}, P, U) \ \diamondsuit$$

Once again we should point out that the extended <P,U>-Justified Update doesn't depend on any initial interpretation. As in the case of normal logic programs, it is the rules that suffer the effect of inertia and not the model literals per se.

Example 7.1.1 *Consider a recoding of the alarm example but using explicit negation, where P and UP are:*

$$\begin{array}{llll} P: & sleep \leftarrow \neg alarm & UP: & in(\neg alarm) \leftarrow \\ & panic \leftarrow alarm & & \\ & alarm \leftarrow & & \end{array}$$

the update program U obtained from UP is:

$$\begin{array}{l} alarm^n \leftarrow \\ \neg alarm^p \leftarrow \neg alarm \\ alarm \leftarrow alarm^p \\ \neg alarm \leftarrow alarm^n \end{array}$$

Intuitively, when performing the update of P by U, we should obtain a single model, namely

$$M = \{\neg alarm, sleep, not\ panic, not\ alarm, not\ \neg sleep, not\ \neg panic\}$$

Let's check whether M is an extended <P, U>-justified update. M is M_{np} restricted to the language of P:

$$M_{np} = \{\neg alarm, sleep, alarm^n, \neg alarm^p, not\ panic, not\ alarm, not\ \neg sleep, not\ \neg panic\}$$

Since

$$Rejected(M_{np}, P, U) = \{alarm \leftarrow\}$$

$$Undefining(M_{np}, P, U) = \{\}$$

$$P^*(M_{np}, P, U) = U + (P - \{alarm \leftarrow\}) + \{\}$$

M_{np} is a partial stable model of $P^*(M_{np}, P, U)$, and so M is an extended <P, U>-justified update.

The previous example had only one 2-valued extended <P,U>-justified update. The following example will show the rôle of $Undefining(M_{np}, P, U)$ when the extended <P,U>-justified updates aren't 2-valued:

Example 7.1.2 *Consider a recoding of the pacifist example but using explicit negation, where P and UP are:*

$$\begin{array}{llll} P: & reasonable \leftarrow pacifist & UP: & in(\neg pacifist) \leftarrow in(war) \\ & pacifist \leftarrow & & in(war) \leftarrow out(peace) \\ & & & in(pacifist) \leftarrow out(war) \end{array}$$

the update program U obtained from UP is (where A is an atom):

$$\begin{array}{ll} pacifist^n \leftarrow war & A \leftarrow A^p \\ war^p \leftarrow not\ pacifist & \neg A \leftarrow A^n \\ pacifist^p \leftarrow not\ war & \neg A^n \leftarrow A \\ & \neg A^p \leftarrow \neg A \end{array}$$

Intuitively, when performing the update of P by U, we should obtain the following three models (where literals not explicitly appearing in P or U are ommited):

$$\begin{array}{lcl} M_1 & = & \{reasonable, pacifist, not\ war, peace, not\ \neg pacifist\} \\ M_2 & = & \{not\ reasonable, not\ pacifist, war, not\ peace, \neg pacifist\} \\ M_3 & = & \{\} \end{array}$$

Let's check whether M_3 is an extended <P, U>-justified update. M_3 is M_{np3} restricted to the language of P:

$$M_{np3} = \{\}$$

Since

$$Rejected(M_{np3}, P, U) = \{pacifist \leftarrow\}$$

$$Undefining(M_{np3}, P, U) = \{pacifist \leftarrow \mathbf{u}\}$$

$$P^*(M_{np}, P, U) = U + (P - \{pacifist \leftarrow\}) + \{pacifist \leftarrow \mathbf{u}\}$$

M_{np3} is a partial stable model of $P^*(M_{np}, P, U)$, and so M_3 is an extended <P, U>-justified update.

7.2 Update Transformation of an Extended Logic Program

Definition 7.2.1 (Update transformation of an extended LP) *Given an update program UP, consider its corresponding extended logic program U. For any extended logic program P, its updated program P_U with respect to U is obtained through the operations:*

- *All rules of U and P belong to P_U subject to these changes, where X is a literal:*
  - *in the head of every rule of P_U originated in U, replace X^p (resp. X^n) by a new literal X^{pU} (resp. X^{nU});*
  - *in the head of every rule of P_U originated in P, replace X by a new literal X';*
- *Include in P_U, for every atom A of P or U, the defining rules:*

$$\begin{array}{ll} A^n \leftarrow \neg A', not\ \neg A^{nU} & A^p \leftarrow A', not\ \neg A^{pU} \\ A^n \leftarrow A^{nU} & A^p \leftarrow A^{pU} \\ \neg A^n \leftarrow \neg A^{nU} & \neg A^p \leftarrow \neg A^{pU} \qquad \diamondsuit \end{array}$$

As before, the transformation reflects that we want to preserve, by inertia, the rules for those literals in P not affected by the update program. This is accomplished via the renaming of the literals in the head of rules only, whilst preserving the body, plus the inertia rules.

Theorem 7.2.1 (Correctness of the update transformation) *Let P be an extended logic program and U a coherent update program. Modulo any primed, A^U, A^p and A^n elements and their defaults, the partial stable models of the updated program P_U of P with respect to U are exactly the extended <P,U>-Justified Updates of P updated by U.* $\diamondsuit$

Proof. Let P be an extended logic program consisting of rules of the form:

$$\begin{array}{l} A \leftarrow B_i, not\ C_i \\ \neg A \leftarrow B_j, not\ C_j \end{array}$$

and U an update program consisting of rules of the form:

$$\begin{array}{ll} A^p \leftarrow B_k, not\ C_k & A \leftarrow A^p \\ \neg A^p \leftarrow B_l, not\ C_l & \neg A \leftarrow A^n \\ A^n \leftarrow B_m, not\ C_m & \neg A^n \leftarrow A \\ \neg A^n \leftarrow B_n, not\ C_n & \neg A^p \leftarrow \neg A \end{array}$$

where A is an atom and each B and C is some finite set of objective literals.

Let $P_U^*(M_{np}, P, U)$ be the program obtained according to Def. 4.3.2:

$$P_U^*(M_{np}, P, U) = U + P_{inertial}(M_{np}, P, U) + Undefining(M_{np}, P, U)$$

and note that $P_{inertial}(M_{np}, P, U) \subseteq P$.

Let P_U be the program obtained according to Def. 4.4.1:

$$P_U : \quad \left.\begin{array}{l} A' \leftarrow B_i, not\ C_i \\ \neg A' \leftarrow B_j, not\ C_j \end{array}\right\} \text{for all rules from } P$$

$$\left.\begin{array}{l} A^{pU} \leftarrow B_k, not\ C_k \\ \neg A^{pU} \leftarrow B_l, not\ C_l \\ A^{nU} \leftarrow B_m, not\ C_m \\ \neg A^{nU} \leftarrow B_n, not\ C_n \\ \neg A^{nU} \leftarrow A \\ \neg A^{pU} \leftarrow \neg A \end{array}\right\} \text{rules from } U \quad \left.\begin{array}{l} A^p \leftarrow A', not\ \neg A^{pU} \\ A^n \leftarrow \neg A', not\ \neg A^{nU} \\ A^p \leftarrow A^{pU} \\ \neg A^p \leftarrow \neg A^{pU} \\ A^n \leftarrow A^{nU} \\ \neg A^n \leftarrow \neg A^{nU} \\ A \leftarrow A^p \\ \neg A \leftarrow A^n \end{array}\right\} \text{for all } A$$

We will show that P_U is equivalent to $P_U^*(M_{np}, P, U)$ for our purposes. Performing on P_U a partial evaluation of A^{pU}, $\neg A^{pU}$, A^{nU} and $\neg A^{nU}$ on the rules $A^p \leftarrow A^{pU}$, $\neg A^p \leftarrow \neg A^{pU}$, $A^n \leftarrow A^{nU}$ and $\neg A^n \leftarrow \neg A^{nU}$ we obtain:

$$\begin{array}{llll} P_U' : & A' \leftarrow B_i, not\ C_i & (1) & \neg A^p \leftarrow \neg A & (10) \\ & \neg A' \leftarrow B_j, not\ C_j & (2) & A \leftarrow A^p & (11) \\ & A^p \leftarrow A', not\ \neg A^{pU} & (3) & \neg A \leftarrow A^n & (12) \\ & A^n \leftarrow \neg A', not\ \neg A^{nU} & (4) & A^{pU} \leftarrow B_k, not\ C_k & (13) \\ & A^p \leftarrow B_k, not\ C_k & (5) & \neg A^{pU} \leftarrow B_l, not\ C_l & (14) \\ & \neg A^p \leftarrow B_l, not\ C_l & (6) & A^{nU} \leftarrow B_m, not\ C_m & (15) \\ & A^n \leftarrow B_m, not\ C_m & (7) & \neg A^{nU} \leftarrow B_n, not\ C_n & (16) \\ & \neg A^n \leftarrow B_n, not\ C_n & (8) & \neg A^{nU} \leftarrow A & (17) \\ & \neg A^n \leftarrow A & (9) & \neg A^{pU} \leftarrow \neg A & (18) \end{array}$$

Note that rules (5)-(12) are exactly equal to the rules of the update program.

As in the proofs of the previous correctness theorems, we have to simplify P_U' eliminating A^{pU}, $\neg A^{pU}$, A^{nU} and $\neg A^{nU}$, considering the following cases for an arbitrary atom A:

$\neg A^{pU}$ and $\neg A^{nU}$ are both false. Then we can remove $not\ \neg A^{pU}$ (resp. $not\ \neg A^{nU}$) from rule (3) (resp. (4)), and perform a partial evaluation of A' and $\neg A'$ in those rules.

Now we no longer need rules (1) and (2) for we don't care about primed literals, nor rules (13)-(18) for we don't care about L^U literals. The so mutilated program looks like this:

$$\begin{array}{llllll} P'_U: & A^p \leftarrow B_i, not\ C_i & (3*) & \neg A^n \leftarrow B_n, not\ C_n & (8) \\ & A^n \leftarrow B_j, not\ C_j & (4*) & \neg A^n \leftarrow A & (9) \\ & A^p \leftarrow B_k, not\ C_k & (5) & \neg A^p \leftarrow \neg A & (10) \\ & \neg A^p \leftarrow B_l, not\ C_l & (6) & A \leftarrow A^p & (11) \\ & A^n \leftarrow B_m, not\ C_m & (7) & \neg A \leftarrow A^n & (12) \end{array}$$

which is semantically equal to $P^*_U(M_{np}, P, U)$ for this case where $P_{inertial}(M_{np}, P, U) = P$ and $Undefining(M_{np}, P, U) = \{\}$ (note that in $P^*_U(M_{np}, P, U)$ the equivalent of rules (3*) and (4*) are instead for A and $\neg A$ respectively, but the effect is the same due to rules (11) and (12)).

$\neg A^{pU}$ is undefined and $\neg A^{nU}$ is false. Then we can remove $not\ \neg A^{nU}$ from rule (4), and perform a partial evaluation of $\neg A'$ in that rule. We can also replace $not\ \neg A^{pU}$ in rule (3) by $\mathbf{u}$, and perform a partial evaluation of A'. Now we no longer need rules (1) and (2) for we don't care about primed literals, nor rules (13)-(18) for we don't care about L^U literals. The so mutilated program looks like this:

$$\begin{array}{llllll} P'_U: & A^p \leftarrow B_i, not\ C_i, \mathbf{u} & (3*) & \neg A^n \leftarrow B_n, not\ C_n & (8) \\ & A^n \leftarrow B_j, not\ C_j & (4*) & \neg A^n \leftarrow A & (9) \\ & A^p \leftarrow B_k, not\ C_k & (5) & \neg A^p \leftarrow \neg A & (10) \\ & \neg A^p \leftarrow B_l, not\ C_l & (6) & A \leftarrow A^p & (11) \\ & A^n \leftarrow B_m, not\ C_m & (7) & \neg A \leftarrow A^n & (12) \end{array}$$

which is semantically equivalent to $P^*_U(M_{np}, P, U)$ for this case. Indeed, note that in $P^*_U(M_{np}, P, U)$, instead of (3*) (or its corresponding rule for A), we would have $A \leftarrow B_i, not\ C_i$ if $B_i, not\ C_i$ is false or undefined, and $A \leftarrow \mathbf{u}$ if $B_i, not\ C_i$ (resp $B_m, not\ C_m$) is true, but the effect is exactly the same and hence their partial stable models are the same too.

$\neg A^{pU}$ and $\neg A^{nU}$ are both undefined. Then we can replace $not\ \neg A^{pU}$ (resp. $not\ \neg A^{nU}$) in rule (3) (resp. (4)) by $\mathbf{u}$, and perform a partial evaluation of A' and $\neg A'$ on those rules. Now we no longer need rules (1) and (2) for we don't care about primed literals, nor rules (13)-(18) for we don't care about L^U literals. The so mutilated program looks like this:

$$\begin{array}{llllll} P'_U: & A^p \leftarrow B_i, not\ C_i, \mathbf{u} & (3*) & \neg A^n \leftarrow B_n, not\ C_n & (8) \\ & A^n \leftarrow B_j, not\ C_j, \mathbf{u} & (4*) & \neg A^n \leftarrow A & (9) \\ & A^p \leftarrow B_k, not\ C_k & (5) & \neg A^p \leftarrow \neg A & (10) \\ & \neg A^p \leftarrow B_l, not\ C_l & (6) & A \leftarrow A^p & (11) \\ & A^n \leftarrow B_m, not\ C_m & (7) & \neg A \leftarrow A^n & (12) \end{array}$$

which again, is semantically equivalent to $P^*_U(M_{np}, P, U)$ for this case.

Due to the symmetric nature of A^p and A^n, the other cases are similar to the ones discussed above. ■

Example 7.2.1 *Applying this transformation to the alarm example (Ex. 7.1.1):*

$$\begin{array}{llll} P: & sleep \leftarrow \neg alarm & U: & in(\neg alarm) \leftarrow \\ & panic \leftarrow alarm & & \\ & alarm \leftarrow & & \end{array}$$

we obtain (where the rules for A and $\neg A$ stand for their ground instances):

$$\begin{array}{lll} P_U: & sleep' \leftarrow \neg alarm & A^p \leftarrow A', not\ \neg A^{pU} \\ & panic' \leftarrow alarm & A^n \leftarrow \neg A', not\ \neg A^{nU} \\ & alarm' \leftarrow & A^p \leftarrow A^{pU} \\ & alarm^{nU} \leftarrow & \neg A^p \leftarrow \neg A^{pU} \\ & \neg alarm^{pU} \leftarrow \neg alarm & A^n \leftarrow A^{nU} \\ & A \leftarrow A^p & \neg A^n \leftarrow \neg A^{nU} \\ & \neg A \leftarrow A^n & \end{array}$$

with single model (modulo L', L^n, L^p, L^U):

$$M_U = \{\neg alarm, sleep, not\ panic, not\ alarm, not\ \neg sleep, not\ \neg panic\}$$

Definition 4.4.2 and Theorem 4.4.2 both now carry over to a language K with explicit negation.

Definition 7.2.2 (Extended factual LP) *Let $I = \langle T, F \rangle$ be an interpretation over a language $\mathcal{K}$ with explicit negation. We define the normal logic program associated with I, $P^*(I)$, as:*

$$P^*(I) = \{L \leftarrow: L \in T\} \cup \{L \leftarrow \mathbf{u} : L \notin T \cup F\} \ \diamondsuit$$

where the L's are objective literals. ◊

As mentioned before, the translation of update programs into extended logic programs making use of default negation to code the out's in the body of update rules, gives rise to some pairs of models, one of which is always closer to the initial interpretation. The closeness condition in Theorems 4.4.2 and 6.1.3 above and Theorem 7.2.2 below exists to eliminate such farther models in order to obtain the U-justified updates only. This is also shared by [5] for the case of updates extended with explicit negation, and so their soundness and completeness theorem should also make use of the closeness relationship.

Theorem 7.2.2 (Generalization of Updates) *Let U be an update program with explicit negation and I an interpretation. Then:*

1. *Every extended U-justified update of I is an extended $<P^*(I), U>$-justified update.*

2. *An extended $<P^*(I), U>$-justified update I_u is an extended U-justified update of I iff*

$$\nexists I_u^{'} \textit{ closer to } I \textit{ than } I_u$$

where $I_u^{'}$ is an extended $<P^*(I), U>$-justified update. $\diamondsuit$

Proof. Let U be an update program consisting of rules of the form:

$$\begin{aligned}
&in(A) \leftarrow in(B_k), out(C_k)\\
&out(A) \leftarrow in(B_l), out(C_l)\\
&in(\neg A) \leftarrow in(B_m), out(C_m)\\
&out(\neg A) \leftarrow in(B_n), out(C_n)
\end{aligned}$$

where A is an atom, each B and C is some finite set of objective literals and in(_) (resp. out(_)) of a set of objective literals $\{L_1, ..., L_p\}$ stands for in(L_1),...,in(L_p). Let $I_i = \langle T_i, F_i \rangle$ and $I_u = \langle T_u, F_u \rangle$ be two partial interpretations with explicit negation.

According to [5], a 3-valued interpretation I_u is an extended U-justified update (or U-justified update for short) of I_i iff it is a U'-justified update according to Definition 3.2.2, where $U' = U \cup \{out(L) \leftarrow in(\neg L) : L \in K\}$, where $\mathcal{K}$ is the set of all objective literals, and all explicitly negated atoms are simply viewed as new atoms. Since atom A and its explicit negation $\neg A$ are now unrelated, we will replace A by A^P and $\neg A$ by A^n both in the update program and the initial interpretation, and map them back into their original form at the end. Since we no longer have explicit negation, we can use Definition 4.4.1 and Theorem 6.1.3 to determine the U'-justified updates. The translation of the update program U' into a logic program is:

$$\begin{aligned}
&A^p \leftarrow B_k, not\ C_k\\
&\neg A^p \leftarrow B_l, not\ C_l\\
&A^n \leftarrow B_m, not\ C_m\\
&\neg A^n \leftarrow B_n, not\ C_n\\
&\neg A^p \leftarrow A^n\\
&\neg A^n \leftarrow A^p
\end{aligned}$$

and the updated program, according to Definition 4.4.1, is:

$$P_U: \quad \begin{array}{ll} \begin{array}{ll} A^{p\prime} \leftarrow & \text{for all } A \in T_i \\ A^{p\prime} \leftarrow \mathbf{u} & \text{for all } A \notin T_i \cup F_i \end{array} & \begin{array}{ll} A^{n\prime} \leftarrow & \text{for all } \neg A \in T_i \\ A^{n\prime} \leftarrow \mathbf{u} & \text{for all } \neg A \notin T_i \cup F_i \end{array} \\ \left.\begin{array}{l} A^{pU} \leftarrow B_k, not\ C_k \\ \neg A^{pU} \leftarrow B_l, not\ C_l \\ A^{nU} \leftarrow B_m, not\ C_m \\ \neg A^{nU} \leftarrow B_n, not\ C_n \\ \neg A^{pU} \leftarrow A^n \\ \neg A^{nU} \leftarrow A^p \end{array}\right\} \text{rules from } U & \left.\begin{array}{l} A^p \leftarrow A^{p\prime}, not\ \neg A^{pU} \\ A^n \leftarrow A^{n\prime}, not\ \neg A^{nU} \\ A^p \leftarrow A^{pU} \\ A^n \leftarrow A^{nU} \\ \neg A^p \leftarrow \neg A^{pU} \\ \neg A^n \leftarrow A^{nU} \end{array}\right\} \text{for all } A \end{array}$$

We would also have to add to P_U the rules $A \leftarrow A^p$ and $\neg A \leftarrow A^n$ to map the newly introduced elements into their original form. A WFSX model of such program (modulo unecessary elements) is a U-justified update iff there isn't another model closer to I_i.

The translation of the update program U into a logic program is:

$$\begin{array}{ll} A^p \leftarrow B_k, not\ C_k & A \leftarrow A^p \\ \neg A^p \leftarrow B_l, not\ C_l & \neg A \leftarrow A^n \\ A^n \leftarrow B_m, not\ C_m & \neg A^n \leftarrow A \\ \neg A^n \leftarrow B_n, not\ C_n & \neg A^p \leftarrow \neg A \end{array}$$

According to Theorem 7.2.1, the extended $<P^*(I), U>$-justified updates are given by the WFSX models of the program:

$$P'_U: \quad \begin{array}{ll} \begin{array}{ll} A' \leftarrow & \text{for all } A \in T_i \\ A' \leftarrow \mathbf{u} & \text{for all } A \notin T_i \cup F_i \end{array} & \begin{array}{ll} \neg A' \leftarrow & \text{for all } \neg A \in T_i \\ \neg A' \leftarrow \mathbf{u} & \text{for all } \neg A \notin T_i \cup F_i \end{array} \\ \left.\begin{array}{l} A^{pU} \leftarrow B_k, not\ C_k \\ \neg A^{pU} \leftarrow B_l, not\ C_l \\ A^{nU} \leftarrow B_m, not\ C_m \\ \neg A^{nU} \leftarrow B_n, not\ C_n \\ \neg A^{pU} \leftarrow A \\ \neg A^{nU} \leftarrow \neg A \\ A \leftarrow A^p \\ \neg A \leftarrow A^n \end{array}\right\} \text{rules from } U & \left.\begin{array}{l} A^p \leftarrow A', not\ \neg A^{pU} \\ A^n \leftarrow \neg A', not\ \neg A^{nU} \\ A^p \leftarrow A^{pU} \\ A^n \leftarrow A^{nU} \\ \neg A^p \leftarrow \neg A^{pU} \\ \neg A^n \leftarrow A^{nU} \end{array}\right\} \text{for all } A \end{array}$$

By performing a partial evaluation of A and $\neg A$ on the rules $\neg A^{pU} \leftarrow A$ and $\neg A^{nU} \leftarrow \neg A$, we obtain:

$$
P'_U: \quad
\begin{array}{ll}
A' \leftarrow & \text{for all } A \in T_i \\
A' \leftarrow \mathbf{u} & \text{for all } A \notin T_i \cup F_i
\end{array}
\qquad
\begin{array}{ll}
\neg A' \leftarrow & \text{for all } \neg A \in T_i \\
\neg A' \leftarrow \mathbf{u} & \text{for all } \neg A \notin T_i \cup F_i
\end{array}
$$

$$
\left.\begin{array}{l}
A^{pU} \leftarrow B_k, not\ C_k \\
\neg A^{pU} \leftarrow B_l, not\ C_l \\
A^{nU} \leftarrow B_m, not\ C_m \\
\neg A^{nU} \leftarrow B_n, not\ C_n \\
\neg A^{pU} \leftarrow A^p \\
\neg A^{nU} \leftarrow A^n \\
A \leftarrow A^p \\
\neg A \leftarrow A^n
\end{array}\right\}\text{rules from } U
\qquad
\left.\begin{array}{l}
A^p \leftarrow A', not\ \neg A^{pU} \\
A^n \leftarrow \neg A', not\ \neg A^{nU} \\
A^p \leftarrow A^{pU} \\
A^n \leftarrow A^{nU} \\
\neg A^p \leftarrow \neg A^{pU} \\
\neg A^n \leftarrow A^{nU}
\end{array}\right\}\text{for all } A
$$

which is semantically equal to $P_U \cup \{A \leftarrow A^p; \neg A \leftarrow A^n\}$.

Since the WFSX models of both programs are the same and the non-closer models are eliminated in the same way as in Theorem 6.1.3, the present theorem follows. ■

7.3 Iterated Extended Update Transformation

Like for normal logic programs, we must be able to update a program more than once. We could apply the transformation of Def.7.2.1 every time we wanted to update a previously updated program. Though possible, this wouldn't be simple, for we would be introducing inertia rules for all atoms newly introduced by a previous update, but which cannot possibly be the subject of an update. To avoid such wasteful rules, we now present a transformation that, given a sequence of update programs, produces another program that can be consulted to determine what is the state after any of those updates. To avoid confusion due to those new atoms introduced by the transformation of update programs into logic programs as presented before, we will re-present the entire process starting from the update programs with in's and out's.

Definition 7.3.1 (Iterated Extended Program Updates) *Let* $S = \{U_t : t \in T\}$*be an ordered sequence of extended update programs* U *over* $\mathcal{K}$. *The* Iterated Extended Update Program P_S*, defined by* S*, written in the extended language*

$$\mathcal{K}^* = \mathcal{K} + \left\{A_t^p, \neg\_A_t^p, A_t^{pU}, \neg\_A_t^{pU}, A_t^n, \neg\_A_t^n, A_t^{nU}, \neg\_A_t^{nU} : A \in \mathcal{K}, t \in T\right\}$$

is obtained via the operations:

- *For each rule r of $U_t \in S$, P_S contains the rule r' such that (where A is an atom, m, $n \geq 0$, and L_i are objective literals):*

  – *if r is of the form*

  $$in(A) \leftarrow in(L_1), ..., in(L_m), out(L_{m+1}), ..., out(L_n)$$

  then r' is

  $$A_t^{pU} \leftarrow L_1, ..., L_m, not\ L_{m+1}, ..., not\ L_n$$

  – *if r is of the form*

  $$in(\neg A) \leftarrow in(L_1), ..., in(L_m), out(L_{m+1}), ..., out(L_n)$$

  then r' is

  $$A_t^{nU} \leftarrow L_1, ..., L_m, not\ L_{m+1}, ..., not\ L_n$$

  – *if r is of the form*

  $$out(A) \leftarrow in(L_1), ..., in(L_m), out(L_{m+1}), ..., out(L_n)$$

  then r' is

  $$\neg\_A_t^{pU} \leftarrow L_1, ..., L_m, not\ L_{m+1}, ..., not\ L_n$$

  – *if r is of the form*

  $$out(\neg A) \leftarrow in(L_1), ..., in(L_m), out(L_{m+1}), ..., out(L_n)$$

  then r' is

  $$\neg\_A_t^{nU} \leftarrow L_1, ..., L_m, not\ L_{m+1}, ..., not\ L_n$$

- *Include in P_S, for every atom A of $\mathcal{K}$, and for any $t > t_0$, the inertia rules:*

$$\begin{array}{ll} A_t^p \leftarrow A_{t-1}^p, not\ \neg\_A_t^{pU} & A_t^n \leftarrow A_{t-1}^n, not\ \neg\_A_t^{nU} \\ \neg\_A_t^p \leftarrow \neg\_A_{t-1}^p, not\ A_t^{pU} & \neg\_A_t^n \leftarrow \neg\_A_{t-1}^n, not\ A_t^{nU} \end{array}$$

- *Include in P_S, for every atom A of $\mathcal{K}$, and for any t, the rules:*

$$\begin{array}{lll} A_t^p \leftarrow A_t^{pU} & A_t^n \leftarrow A_t^{nU} & \neg\_A_t^{nU} \leftarrow A_t^p \\ \neg\_A_t^p \leftarrow \neg\_A_t^{pU} & \neg\_A_t^n \leftarrow \neg\_A_t^{nU} & \neg\_A_t^{pU} \leftarrow A_t^n \ \diamondsuit \end{array}$$

The above definition assumes that in the language $\mathcal{K}$ there are no symbols of the form A_t^p, $\neg\_A_t^p$, A_t^{pU}, $\neg\_A_t^{pU}$, A_t^n, $\neg\_A_t^n$, A_t^{nU} and $\neg\_A_t^{nU}$. Note that P_S does not contain any rules for atoms of $\mathcal{K}$, i.e. neither A nor $\neg A$ appear in the heads of the rules of P_S.

Now, in order to find out what the result of the updates up to a given state t is, we add the current state defining rules to obtain the program at a given state t:

Definition 7.3.2 (Iterated Extended Update Program at a given state t.) *Given a state $t \in T$, and an iterated extended non-contradictory update program P_S, the* Iterated Extended Update Program at a given state t*, $P_S(t)$, is the program P_S augmented with the current state defining rules:*

$$\begin{array}{lll} A^p \leftarrow A_t^p & A^n \leftarrow A_t^n & A \leftarrow A^p \\ \neg A^p \leftarrow \neg\_A_t^p & \neg A^n \leftarrow \neg\_A_t^n & \neg A \leftarrow A^n \end{array}$$

for all $A \in \mathcal{K}$, if $P_S(r)$ is non-contradictory for all $r \in T, r < t$. Otherwise $P_S(t)$ is not defined. $\diamondsuit$

The above definitions generalize the notion of updating programs to the most general case, i.e. when we're dealing with more than one update program, written over a language with explicit negation and considering extended partial stable models. They allow us to obtain in a simpler and more efficient way the same results that would be obtained by sucessiveely updating the initial program U_{t_0} by the subsequent update programs using Def. 7.2.1.

In the next chapter we exhibit the application of these concepts to an illustrative example.

Chapter 8

An Illustrative Example of Iterated Updating

8.1 University Faculty Evaluation

Consider a university where it has been decided to start a periodic evaluation of faculty members based on their teaching and research activities.

At this university, it is accepted that anyone who has published many papers is considered a good researcher, and anyone whose classes are liked by the students is considered a good teacher.

At the begining of the first evaluation period it was agreed that faculty members known to be good researchers and good teachers would receive a positive evaluation. Those not known to be strong researchers would not be positively evaluated, and those recognized as poor teachers would receive a negative evaluation. This leads us to the following update program, U_1, with the obvious abbreviations, where non-ground rules stand for the set of their ground instances:

$$
\begin{aligned}
U_1 : \quad & in(g\_teach(X)) \leftarrow in(students\_like(X)) \\
& in(g\_res(X)) \leftarrow in(many\_papers(X)) \\
& in(g\_eval(X)) \leftarrow in(g\_res(X)), in(g\_teach(X)) \\
& out(g\_eval(X)) \leftarrow out(g\_res(X)) \\
& in(\neg g\_eval(X)) \leftarrow in(\neg g\_teach(X))
\end{aligned}
$$

Over this first evaluation period, Scott, Peter and Lisa published many papers, but not Jack. Also, a survey showed that the students liked Scott, Peter and Jack but didn't like Lisa. This knowledge leads to the following update program:

$$\begin{array}{lll} U_2: & in(many\_papers(scott)) \leftarrow & in(students\_like(scott)) \leftarrow \\ & in(many\_papers(peter)) \leftarrow & in(students\_like(peter)) \leftarrow \\ & in(many\_papers(lisa)) \leftarrow & out(students\_like(lisa)) \leftarrow \\ & out(many\_papers(jack)) \leftarrow & in(students\_like(jack)) \leftarrow \end{array}$$

Applying Def. 7.3.1, with $T_2 = \{1, 2\}$ and $S_2 = \{U_1, U_2\}$ we obtain the following iterated extended update program P_{S_2}:

$$\begin{array}{l} g\_teach(X)_1^{pU} \leftarrow students\_like(X) \\ g\_res(X)_1^{pU} \leftarrow many\_papers(X) \\ g\_eval(X)_1^{pU} \leftarrow g\_res(X), g\_teach(X) \\ \neg\_g\_eval(X)_1^{pU} \leftarrow not\ g\_res(X) \\ g\_eval(X)_1^{nU} \leftarrow \neg g\_teach(X) \end{array}$$

$$\begin{array}{ll} many\_papers(scott)_2^{pU} \leftarrow & students\_like(scott)_2^{pU} \leftarrow \\ many\_papers(peter)_2^{pU} \leftarrow & students\_like(peter)_2^{pU} \leftarrow \\ many\_papers(lisa)_2^{pU} \leftarrow & \neg\_students\_like(lisa)_2^{pU} \leftarrow \\ \neg\_many\_papers(jack)_2^{pU} \leftarrow & students\_like(jack)_2^{pU} \leftarrow \end{array}$$

$$\left.\begin{array}{l} A_t^p \leftarrow A_{t-1}^p, not\ \neg\_A_t^{pU} \\ \neg\_A_t^p \leftarrow \neg\_A_{t-1}^p, not\ A_t^{pU} \\ A_t^n \leftarrow A_{t-1}^n, not\ \neg\_A_t^{nU} \\ \neg\_A_t^n \leftarrow \neg\_A_{t-1}^n, not\ A_t^{nU} \end{array}\right\} \begin{array}{l} \text{for every atom A and} \\ \text{any t} > 1 \text{ (t} \in T_2) \end{array}$$

$$\left.\begin{array}{l} A_t^p \leftarrow A_t^{pU} \\ \neg\_A_t^p \leftarrow \neg\_A_t^{pU} \\ A_t^n \leftarrow A_t^{nU} \\ \neg\_A_t^n \leftarrow \neg\_A_t^{nU} \\ \neg\_A_t^{nU} \leftarrow A_t^p \\ \neg\_A_t^{pU} \leftarrow A_t^n \end{array}\right\} \begin{array}{l} \text{for every atom A and} \\ \text{any t} \in T_2 \end{array}$$

To determine the iterated extended update program at the end of the first evaluation period, i.e. at state 2, $P_{S_2}(2)$, we add the following rules (as per Def.7.3.2) to P_{S_2} (for every atom A):

$$\begin{array}{lll} A^p \leftarrow A_2^p & A^n \leftarrow A_2^n & A \leftarrow A^p \\ \neg A^p \leftarrow \neg\_A_2^p & \neg A^n \leftarrow \neg\_A_2^n & \neg A \leftarrow A^n \end{array}$$

The only partial stable model of $P_{S_2}(2)$ is (modulo irrelevant atoms):

$$\begin{array}{llcll}
M_2 = \{ & many_papers(scott) & , & many_papers(lisa) & , \\
& students_like(scott) & , & not\ students_like(lisa) & , \\
& g_teach(scott) & , & not\ g_teach(lisa) & , \\
& not\ \neg g_teach(scott) & , & not\ \neg g_teach(lisa) & , \\
& g_res(scott) & , & g_res(lisa) & , \\
& g_eval(scott) & , & not\ g_eval(lisa) & , \\
& not\ \neg g_eval(scott) & , & not\ \neg g_eval(lisa) & , \\
& many_papers(peter) & , & not\ many_papers(jack) & , \\
& students_like(peter) & , & students_like(jack) & , \\
& g_teach(peter) & , & g_teach(jack) & , \\
& not\ \neg g_teach(peter) & , & not\ \neg g_teach(jack) & , \\
& g_res(peter) & , & not\ g_res(jack) & , \\
& g_eval(peter) & , & not\ g_eval(jack) & , \\
& not\ \neg g_eval(peter) & , & not\ \neg g_eval(jack) & \}
\end{array}$$

Stating that both Scott and Peter receive a good evaluation; Lisa is not positively evaluated for she is not a good teacher; Jack is also not positively evaluated for he didn't publish any papers and so he is not considered a good researcher.

After this period, the evaluation board found out that one of the reasons why the students liked some of the teachers was because they missed a lot of classes. This way, they decided that anyone missing a lot of classes shouldn't be considered a good teacher. This rule was to be applied in the next evaluation period and can be represented by the following update program U_3:

$$U_3 : \quad out(g\_teach(X)) \leftarrow in(miss\_classes(X))$$

During this evaluation period, Peter missed a lot of his classes, Jack published many papers, and everything else kept as before. This leads to the following update program U_4:

$$\begin{array}{ll}
U_4 : & in(miss_classes(peter)) \leftarrow \\
& in(miss_classes(jack)) \leftarrow
\end{array}$$

Applying Def. 7.3.1, with

$$\begin{array}{rcl}
T_4 & = & \{1,2,3,4\} \\
S_4 & = & \{U_1, U_2, U_3, U_4\}
\end{array}$$

we obtain the iterated extended update program P_{S_4} which is equal to P_{S_2}, replacing T_2 by T_4, together with the rules:

$$\begin{array}{l}
\neg_g_teach(X)_3^{pU} \leftarrow miss_classes(X) \\
miss_classes(peter)_4^{pU} \leftarrow \\
miss_classes(jack)_4^{pU} \leftarrow
\end{array}$$

To determine the iterated extended update program at the end of the second evaluation period, i.e. at state 4, $P_{S_4}(4)$, we add the following rules (as per Def.7.3.2) to P_{S_4} (for every atom A):

$$\begin{array}{lll}
A^p \leftarrow A_4^p & A^n \leftarrow A_4^n & A \leftarrow A^p \\
\neg A^p \leftarrow \neg_A_4^p & \neg A^n \leftarrow \neg_A_4^n & \neg A \leftarrow A^n
\end{array}$$

The only partial stable model of $P_{S_4}(4)$ is (modulo irrelevant atoms):

$$\begin{array}{lllll}
M_4 = \{ & many_papers(scott) & , & many_papers(lisa) & , \\
& not\ miss_classes(scott) & , & not\ miss_classes(lisa) & , \\
& students_like(scott) & , & not\ students_like(lisa) & , \\
& g_teach(scott) & , & not\ g_teach(lisa) & , \\
& not\ \neg g_teach(scott) & , & not\ \neg g_teach(lisa) & , \\
& g_res(scott) & , & g_res(lisa) & , \\
& g_eval(scott) & , & not\ g_eval(lisa) & , \\
& not\ \neg g_eval(scott) & , & not\ \neg g_eval(lisa) & , \\
& many_papers(peter) & , & not\ many_papers(jack) & , \\
& miss_classes(peter) & , & miss_classes(jack) & , \\
& students_like(peter) & , & students_like(jack) & , \\
& not\ g_teach(peter) & , & not\ g_teach(jack) & , \\
& not\ \neg g_teach(peter) & , & not\ \neg g_teach(jack) & , \\
& g_res(peter) & , & not\ g_res(jack) & , \\
& not\ g_eval(peter) & , & not\ g_eval(jack) & , \\
& not\ \neg g_eval(peter) & , & not\ \neg g_eval(jack) & \}
\end{array}$$

Note that, in what Peter's evaluation is concerned, although the students like him and he has published many papers, he is not considered a good teacher and thus doesn't have a good evaluation because he misses a lot of classes. This is so because missing a lot of classes makes the body of the rule

$$out(g\_teach(peter)) \leftarrow in(miss\_classes(peter))$$

true, and in turn inhibits inertia to be exerted on the rule

$$in(g\_teach(peter)) \leftarrow in(students\_like(peter))$$

After this period, the evaluation board, still not happy with the criteria to evaluate the teaching skills, decided to directly evaluate some of the faculty members by means of special examinations. If a teacher has a good teaching evaluation as a result he/she is considered a good teacher. In the case of a bad teaching evaluation he/she is considered a poor teacher. This can be represented by the following update program U_5:

$$\begin{aligned} U_5: \quad & in(g\_teach(X)) \leftarrow in(g\_teach\_evaluation(X)) \\ & in(\neg g\_teach(X)) \leftarrow in(\neg g\_teach\_evaluation(X)) \end{aligned}$$

During this evaluation period, all but Scott were directly evaluated in what their teaching skills are concerned: Peter and Lisa obtained a good teaching evaluation and Jack a negative one. Also during this period, Scott (who has always been an example to everyone) missed a lot of classes. Every other aspect was the same as in the previous evaluation period. This leads to the next update program U_6:

$$\begin{array}{lll} U_6: & in(g\_teach\_evaluation(peter)) \leftarrow & in(\neg g\_teach\_evaluation(jack)) \leftarrow \\ & in(g\_teach\_evaluation(lisa)) \leftarrow & in(miss\_classes(scott)) \leftarrow \end{array}$$

Applying Def. 7.3.1, with

$$\begin{aligned} T_6 &= \{1, 2, 3, 4, 5, 6\} \\ S_6 &= \{U_1, U_2, U_3, U_4, U_5, U_6\} \end{aligned}$$

we obtain the iterated extended update program P_{S_6} which is equal to P_{S_4}, replacing T_4 by T_6, together with the rules:

$$\begin{array}{ll} g\_teach(X)_5^{pU} \leftarrow g\_teach\_evaluation(X) & \\ g\_teach(X)_5^{nU} \leftarrow \neg g\_teach\_evaluation(X) & \\ g\_teach\_evaluation(peter)_6^{pU} \leftarrow & g\_teach\_evaluation(jack)_6^{nU} \leftarrow \\ g\_teach\_evaluation(lisa)_6^{pU} \leftarrow & miss\_classes(scott)_6^{pU} \leftarrow \end{array}$$

To determine the iterated extended update program at the end of the third evaluation period, i.e. at state 6, $P_{S_6}(6)$, we add the following rules (as per Def.7.3.2) to P_{S_6} (for every atom A):

$$\begin{array}{lll} A^p \leftarrow A_6^p & A^n \leftarrow A_6^n & A \leftarrow A^p \\ \neg A^p \leftarrow \neg\_A_6^p & \neg A^n \leftarrow \neg\_A_6^n & \neg A \leftarrow A^n \end{array}$$

The only partial stable model of $P_{S_6}(6)$ is (modulo irrelevant atoms):

$$
\begin{array}{llcll}
M_6 = \{ & many\_papers(scott) & , & many\_papers(lisa) & , \\
& miss\_classes(scott) & , & not\ miss\_classes(lisa) & , \\
& students\_like(scott) & , & not\ students\_like(lisa) & , \\
& not\ g\_teach\_evaluation(scott) & , & g\_teach\_evaluation(lisa) & , \\
& not\ \neg g\_teach\_evaluation(scott) & , & not\ \neg g\_teach\_evaluation(lisa) & , \\
& not\ g\_teach(scott) & , & g\_teach(lisa) & , \\
& not\ \neg g\_teach(scott) & , & not\ \neg g\_teach(lisa) & , \\
& g\_res(scott) & , & g\_res(lisa) & , \\
& not\ g\_eval(scott) & , & g\_eval(lisa) & , \\
& not\ \neg g\_eval(scott) & , & not\ \neg g\_eval(lisa) & , \\
& many\_papers(peter) & , & not\ many\_papers(jack) & , \\
& miss\_classes(peter) & , & miss\_classes(jack) & , \\
& students\_like(peter) & , & students\_like(jack) & , \\
& g\_teach\_evaluation(peter) & , & not\ g\_teach\_evaluation(jack) & , \\
& not\ \neg g\_teach\_evaluation(peter) & , & \neg g\_teach\_evaluation(jack) & , \\
& g\_teach(peter) & , & not\ g\_teach(jack) & , \\
& not\ \neg g\_teach(peter) & , & \neg g\_teach(jack) & , \\
& g\_res(peter) & , & not\ g\_res(jack) & , \\
& g\_eval(peter) & , & not\ g\_eval(jack) & , \\
& not\ \neg g\_eval(peter) & , & \neg g\_eval(jack) & \}
\end{array}
$$

Note that although Jack's students like him and he doesn't miss many classes, he is now considered a bad teacher due to a direct evaluation of his teaching skills by the board. This leads to a negative overall evaluation. In what Lisa is concerned, her teaching skills were positively evaluated and, although her students don't like her, she is considered a good teacher a receives a good overall evaluation for she is also a good researcher.

After this evaluation period the board decided that, since Scott missed a lot of classes due to an ongoing project that could lead him to win a Nobel Prize, anyone who receives such high award should be considered a good researcher and automatically receive a good evaluation. This can be represented by the following update program U_7:

$$
\begin{array}{ll}
U_7: & in(g\_res(X)) \leftarrow in(nobel\_prize(X)) \\
& in(g\_eval(X)) \leftarrow in(nobel\_prize(X))
\end{array}
$$

It turned out that Scott indeed did receive the Nobel Prize. Also during this evaluation period, Peter didn't publish any papers. A teaching re-evaluation on Jack's teaching skills showed that, although he should not be considered a good teacher, he should not be considered a bad teacher either. Every other aspect remained the same. This leads to the update program U$_8$:

$$\begin{aligned} U_8 : \quad & in(nobel\_prize(scott)) \leftarrow \\ & out(many\_papers(peter)) \leftarrow \\ & out(\neg g\_teach\_evaluation(jack)) \leftarrow \end{aligned}$$

Applying Def. 7.3.1, with

$$\begin{aligned} T_8 &= \{1, 2, 3, 4, 5, 6, 7, 8\} \\ S_8 &= \{U_1, U_2, U_3, U_4, U_5, U_6, U_7, U_8\} \end{aligned}$$

we obtain the following iterated extended update program P_{S_8} which is equal to P_{S_6}, replacing T_6 by T_8, together with the rules:

$$\begin{aligned} & g\_res(X)_7^{pU} \leftarrow nobel\_prize(X) \\ & g\_eval(X)_7^{pU} \leftarrow nobel\_prize(X) \\ & nobel\_prize(scott)_8^{pU} \leftarrow \\ & \neg\_many\_papers(peter)_8^{pU} \leftarrow \\ & \neg\_g\_teach\_evaluation(jack)_8^{nU} \leftarrow \end{aligned}$$

The iterated extended update program P_{S_8} representing the complete scenario since it was decided to start the evaluation is:

$$
\begin{array}{ll}
g\_teach(X)_1^{pU} \leftarrow students\_like(X) & \\
g\_res(X)_1^{pU} \leftarrow many\_papers(X) & \\
g\_eval(X)_1^{pU} \leftarrow g\_res(X), g\_teach(X) & \\
\neg\_g\_eval(X)_1^{pU} \leftarrow not\ g\_res(X) & \\
g\_eval(X)_1^{nU} \leftarrow \neg g\_teach(X) & \\
many\_papers(scott)_2^{pU} \leftarrow & students\_like(scott)_2^{pU} \leftarrow \\
many\_papers(peter)_2^{pU} \leftarrow & students\_like(peter)_2^{pU} \leftarrow \\
many\_papers(lisa)_2^{pU} \leftarrow & \neg\_students\_like(lisa)_2^{pU} \leftarrow \\
\neg\_many\_papers(jack)_2^{pU} \leftarrow & students\_like(jack)_2^{pU} \leftarrow \\
\neg\_g\_teach(X)_3^{pU} \leftarrow miss\_classes(X) & \\
miss\_classes(peter)_4^{pU} \leftarrow & \\
miss\_classes(jack)_4^{pU} \leftarrow & \\
g\_teach(X)_5^{pU} \leftarrow g\_teach\_evaluation(X) & \\
g\_teach(X)_5^{nU} \leftarrow \neg g\_teach\_evaluation(X) & \\
g\_teach\_evaluation(peter)_6^{pU} \leftarrow & g\_teach\_evaluation(jack)_6^{nU} \leftarrow \\
g\_teach\_evaluation(lisa)_6^{pU} \leftarrow & miss\_classes(scott)_6^{pU} \leftarrow \\
g\_res(X)_7^{pU} \leftarrow nobel\_prize(X) & \\
g\_eval(X)_7^{pU} \leftarrow nobel\_prize(X) & \\
nobel\_prize(scott)_8^{pU} \leftarrow & \\
\neg\_many\_papers(peter)_8^{pU} \leftarrow & \\
\neg\_g\_teach\_evaluation(jack)_8^{nU} \leftarrow &
\end{array}
$$

$$
\left.
\begin{array}{l}
A_t^p \leftarrow A_{t-1}^p, not\ \neg\_A_t^{pU} \\
\neg\_A_t^p \leftarrow \neg\_A_{t-1}^p, not\ A_t^{pU} \\
A_t^n \leftarrow A_{t-1}^n, not\ \neg\_A_t^{nU} \\
\neg\_A_t^n \leftarrow \neg\_A_{t-1}^n, not\ A_t^{nU}
\end{array}
\right\}
\begin{array}{l}
\text{for every atom A and} \\
\text{any t} > 1\ (\text{t} \in T_8)
\end{array}
$$

$$
\left.
\begin{array}{l}
A_t^p \leftarrow A_t^{pU} \\
\neg\_A_t^p \leftarrow \neg\_A_t^{pU} \\
A_t^n \leftarrow A_t^{nU} \\
\neg\_A_t^n \leftarrow \neg\_A_t^{nU} \\
\neg\_A_t^{nU} \leftarrow A_t^p \\
\neg\_A_t^{pU} \leftarrow A_t^n
\end{array}
\right\}
\begin{array}{l}
\text{for every atom A and} \\
\text{any t} \in T_8
\end{array}
$$

To determine the iterated extended update program at the end of the fourth evaluation period, i.e. at state 8, $P_{S_8}(8)$, we add the following rules (as per Def.7.3.2) to P_{S_8} (for every atom A):

$$\begin{array}{lll} A^p \leftarrow A_8^p & A^n \leftarrow A_8^n & A \leftarrow A^p \\ \neg A^p \leftarrow \neg\_A_8^p & \neg A^n \leftarrow \neg\_A_8^n & \neg A \leftarrow A^n \end{array}$$

The only partial stable model of $P_{S_8}(8)$ is (modulo irrelevant atoms):

$$\begin{array}{lllll}
M_8 = \{ & many_papers(scott) & , & many_papers(lisa) & , \\
& miss_classes(scott) & , & not\ miss_classes(lisa) & , \\
& students_like(scott) & , & not\ students_like(lisa) & , \\
& nobel_prize(scott) & , & not\ nobel_prize(lisa) & , \\
& not\ g_teach_evaluation(scott) & , & g_teach_evaluation(lisa) & , \\
& not\ \neg g_teach_evaluation(scott) & , & not\ \neg g_teach_evaluation(lisa) & , \\
& not\ g_teach(scott) & , & g_teach(lisa) & , \\
& not\ \neg g_teach(scott) & , & not\ \neg g_teach(lisa) & , \\
& g_res(scott) & , & g_res(lisa) & , \\
& g_eval(scott) & , & g_eval(lisa) & , \\
& not\ \neg g_eval(scott) & , & not\ \neg g_eval(lisa) & , \\
& not\ many_papers(peter) & , & not\ many_papers(jack) & , \\
& miss_classes(peter) & , & miss_classes(jack) & , \\
& students_like(peter) & , & students_like(jack) & , \\
& not\ nobel_prize(peter) & , & not\ nobel_prize(jack) & , \\
& g_teach_evaluation(peter) & , & not\ g_teach_evaluation(jack) & , \\
& not\ \neg g_teach_evaluation(peter) & , & not\ \neg g_teach_evaluation(jack) & , \\
& g_teach(peter) & , & not\ g_teach(jack) & , \\
& not\ \neg g_teach(peter) & , & not\ \neg g_teach(jack) & , \\
& not\ g_res(peter) & , & not\ g_res(jack) & , \\
& not\ g_eval(peter) & , & not\ g_eval(jack) & , \\
& not\ \neg g_eval(peter) & , & not\ \neg g_eval(jack) & \}
\end{array}$$

Note that after a second evaluation of Jack's teaching skills, he was no longer considered a bad teacher. His overall evaluation is still not positive, but at least it is no longer negative.

It is worth pointing out that after each iteration, no information is lost. We could still use P_{S_8} to determine the previous states by adding the current state defining rules, as in Def.7.3.2, for the desired state, and obtain the same models as above.

Note that if we wanted to know the effect of the new evaluation rules, added at the begining of each period, on the evaluation record of the previous period, we could simply determine the state at $t = 3, 5$ and 7.

Chapter 9

Conclusions and Future Work

Throughout this work we have motivated and formalized the generalization of the notion of updates to the case where we want to update programs instead of just their models. We have shown that since a program encodes more information than a set of models, the law of inertia should be applied to rules instead of to model literals, as had been done so far. We presented a transformation which, given an initial program and an update program, generates the desired updated program. This was achieved both for the stable as well as for the partial stable semantics. Our results have been further generalized to allow for programs or update programs extended with explicit negation. Another important result set forth in this work is the extension to the case where we want to update a given program more than once, i.e. the iterated updates[1]. This is important inasmuch as it allows us to conceive what it is to successively update one program by another, and so to define the evolution of knowledge bases by means of updates.

With respect to improvements on the results presented here, a joint paper together with Dr. José Júlio Alferes, Prof. Dr. Luís Moniz Pereira, Prof. Dr. Halina Przymusinska and Prof. Dr. Teodor Przymusinski is well under way, exploring a slightly different update transformation for the case of normal programs under the stable semantics. This new transformation was set forth with the goal of not allowing those non-closer models (see ex.4.4.3). It no longer uses default negation to code the out's in the body of rules; both the in's and the out's give rise to positive atoms that are related through the inertia rules and a special semantics devised for the case.

Program updating is a new and crucial notion deserving future foundational work and

[1] A prolog meta-interpreter for the iterated updates has been develloped. This meta-interpreter is listed in apendix, and available at `http:\\www-ssdi.di.fct.unl.pt\~jleite\` or on request from the author (`jleite@di.fct.unl.pt`).

opening up a whole new range of applications such as:

Software specification: One of the major tasks in the software industry is the updating of software. The release of a new version of a program involves a great deal of work with respect to the integration of new features and improvement of old ones, namely in what the backwards compatibility is concerned. This process would be made much easier if we could just simply specify the update i.e. the parts of the program specific to the new version, and automatically generate the updated version.

Modeling of human reasoning: Human knowledge is in constant evolution. It can be regarded as a continuous and incremental process of acquiring, processing and intergrating information. As one of the widest used frameworks for representing human knowledge is that of logic programming, updates can be viewed as a promising candidate to model and reason about its incremental evolution.

Generalize the scope of update rules: In recent years, much work has been done relating to the issue of disjunctive logic program semantics. It is therefore natural to extend the logic program updates to allow for disjunctive update rules.

Furthermore, just as interpretation updates were extended to the case of updates by means of arbitrary inference rules [34], logic program updates should also be generalized, possibly using super logic programs [9] (this would include the disjunctive case above).

Reasoning about the past: Since no information is lost during the update process, i.e. after any iteration we can still characterize any previous state, we can use this framework to reason about the past in various ways:

- the scope of update rules could be extended to allow for updates to depend on some past state. This would be done by allowing the body of update rules to refer to previous states, thus increasing the expressibility of updates.
- the iterated updated program at state t can be used to directly determine *what might have happened in the past with present knowledge.* This could be done by checking the truth value of atoms $A^p_{t_1}$ and $A^n_{t_1}$ where t_1 refers to the previous state we want reason about.
- also we could backpropagate present knowledge i.e. use reverse inertia to allow present knowledge to be propagated to the past. This would allow for a form of counterfactual reasoning.

Self updates Throughout this work we assumed that updates were externally driven. This need not be so. We could allow for internal self updating. This would be of the form of update rules scheduled to be triggered at some state, under certain conditions. Informally, an example of such kind of self update rules would be:

$$up\,[in(L) \leftarrow in(L_1), out(L_2)] \leftarrow in(L_3), out(L_4)$$

with the intended meaning of: if at state t $in(L_3), out(L_4)$ is verified, perform the update $in(L) \leftarrow in(L_1), out(L_2)$ at state $t+1$.

A simple example would be a clock that could be modeled by the following two self update rules:

$$up(out(s)) \leftarrow in(s)$$
$$up(in(s)) \leftarrow out(s)$$

Improved expressibility could be obtained by combining these self updates with the externally driven ones. A simple application example of this combination would be the modeling of an automata receiving external input by means of an update, changing its state by means of a self update, and performing some output.

This combination of self and externally driven updates would require special care in what their relationship is concerned: should they take place together? Should they alternate? Should they be synchronized?.

Allowing internal self update rules opens up the concept of self-evolving programs, that could be used to model artificial life.

Dealing with contradiction: Contradiction may have its origin at several stages of the update process: both the program to be updated and the update program may be contradictory, and even if they are both non-contradictory, the updated program may be so. One should apply the well known contradiction avoidance/removal techniques [3][4] to updates. Furthermore updates should be generalized to include integrity constraints.

An interesting question arises when we consider the removal of contradiction from the updated program by means of belief revision: should we treat old knowledge (i.e. things that are concluded by means of inertia) in the same way as new knowledge (i.e. things that are derived by means of an update rule)? A rather conservative mind, when faced with an update that is contradictory with its knowledge, will simply discard the update. On the other side, a progressive mind will prefer the new knowledge over the inertial one.

Also updates could be used as a means to remove contradiction. Note that the update of a contradictory program need not necessarily be contradictory.

Instantaneous updates: Besides the updates that persist by inertia, we might want to specify updates that after they take effect no longer continue by inertia. For example, an update may specify some input to a program, and we may wish that it should not remain in effect forever because there will be successive other inputs of the same kind that should supersede one another. This concept of instantaneous updates would also be useful to update a program with the indication that some action has occured now. Intuitively, we want the action to 'go away' afterwards, leaving the way open for next action(s).

Combining instantaneous with persistent updates provide an increased expressibility

Reasoning about actions: Updates pave the way for the dynamics of logic programming as opposed to its statics which has received most attention till now. Areas such as temporal databases, production systems, event calculus and situation calculus can benefit and be fertilized by this updates theory.

The world is in constant evolution and so must be any form of representing it. Human knowledge and reasoning whose modeling is one of the ultimate goals of artificial inteligence, not only is in permanent change, it is so in an incremental way. We constantly receive information about particular aspects of our surroundings and automatically integrate and make it part of the whole of our knowledge. In this process we attempt to keep as much of our previous knowledge as possible although some of it can be abandoned by means of a re-evaluation process. The updating of logic programs, as presented throughout this work, due to its incremental nature and intuitive results, is a good way to model knowledge evolution.

We hope to have convinced the reader of the importance of updating logic programs. In our view, they pave the ground for a new area of research, that of *Logic Programming Updating*, dealing with the evolution of programs.

Bibliography

[1] J. J. Alferes, C. V. Damásio and L. M. Pereira. *A logic programming system for non-monotonic reasoning.* Journal of Automated Reasoning, 14:93-147, 1995.

[2] J. J. Alferes and L. M. Pereira. *On logic program semantics with two kinds of negation.* In K. Apt, editor. Int. Joint Conf. and Symp. on LP, pages 574-588. MIT Press. 1992.

[3] J. J. Alferes and L. M. Pereira. *Contradiction: when avoidance equals removal, I and II.* In Dyckhoff, editor, 4th ELP, volume 798 of LNAI. Springer-Verlag, 1994.

[4] J. J. Alferes and L. M. Pereira *Reasonning with Logic Programming.* LNAI 1111, Berlin. Springer-Verlag, 1996.

[5] J. J. Alferes and L. M. Pereira. *Update-programs can update programs.* In J. Dix, L. M. Pereira and T. Przymusinski, editors, Selected papers from the ICLP'96 ws NMELP'96, vol. 1216 of LNAI, pages 110-131. Springer-Verlag, 1997.

[6] J. J. Alferes, L. M. Pereira and T. Przymusinski. *Strong and Explicit Negation in Nonmonotonic Reasoning and Logic Programming.* In J. J. Alferes, L. M. Pereira and E. Orlowska, editors, JELIA'96, volume 1126 of LNAI, pages 143-163. Springer-Verlag, 1996. Extended version to appear in Journal of Automated Reasoning, 1998.

[7] S. Brass and J. Dix. *Disjunctive Semantics based upon Partial and Bottom-Up Evaluation.* In Leon Sterling, editor, Procs. of the 12th Int. Conf. on Logic Programming, Tokyo, pag. 85-98, Berlin, June 1995. MIT Press.

[8] G. Brewka, J. Dix and K. Konolige. *Nonmonotonic Reasoning: An Overview.* CSLI Lecture Notes 73. CSLI Publications, Stanford, CA, 1997.

[9] S. Brass, J. Dix and T. Przymusinski. *Super Logic Programs.* In L. C. Aiello, J. Doyle, and S. C. Shaphiro, editors, Principles of Knowledge Representation and Reasoning, Proc. of the Third Int'l Conf (KR96), pages 529-541. San Francisco, CA, Morgan-Kaufmann, 1996.

[10] J. Dix. *A Classification-Theory of Semantics of Normal Logic Programs: I. Strong Properties.* Fundamenta Informaticae, XXII(3):227-255, 1995.

[11] J. Dix. *A Classification-Theory of Semantics of Normal Logic Programs: I. Strong Properties.* Fundamenta Informaticae, XXII(3):257-288, 1995.

[12] J. Dix. *Semantics of Logic Programs: Their Intuitions and Formal Properties. An Overview.* In Andre Fuhrmann and Hans Rott, editors, Logic, Action and Information - Essays on Logic in Philosophy and Artificial Intelligence, pages 241-327. DeGruyter, 1995.

[13] C. V. Damásio and L. M. Pereira. *Default negated conclusions: why not?* In R. Dyckhoff, H. Herre and P. Schroeder-Heister, editors, Procs. of ELP'96. LNAI. Springer-Verlag, 1996.

[14] J. Dix, L. M. Pereira and T. Przymusinski. *Prolegomena to Logic Programming for Non-monotonic Reasoning.* In J. Dix, L. M. Pereira and T. Przymusinski, editors, Introduction to selected papers from NMELP'96, volume 1216 of LNAI, pages 1-36. Springer-Verlag, 1997.

[15] M. Fitting. *A Kripke-Kleene semantics for logic programs.* Journal of LP, 2(4):295-312, 1985.

[16] M. Van Emden and R. Kowalski. *The semantics of predicate logic as a programming language.* Journal of ACM, 4(23):733-742, 1976.

[17] M. Gelfond and V. Lifschitz. *The stable model semantics for logic programming.* In R. Kowalski and K. A. Bowen. editors. 5th Int. Conf. on LP, pages 1070-1080. MIT Press, 1988.

[18] M. Gelfond and V. Lifschitz. *Logic Programs with classical negation.* In Warren and Szeredi, editors, 7th Int. Conf. on LP, pages 579-597. MIT Press, 1990.

[19] M. Gelfond and V. Lifschitz. *Classical Negation in Logic Programs and Disjunctive Databases.* New Generation Computing, 9:365-387, 1991.

[20] 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.

[21] H. Katsuno and A. Mendelzon. *On the difference between updating a knowledge base and revising it.* In James Allen, Richard Fikes and Erik Sandewall, editors, Principles

of Knowledge Representation and Reasoning: Proc. of the Second Int'l Conf., pages 230-237, Morgan Kaufmann 1991.

[22] A. Keller and M. Winslett Wilkins. *On the use of an extended relational model to handle changing incomplete information.* IEEE Trans. on Software Engineering, SE-11:7, pages 620-633, 1985.

[23] V. Lifschitz and T. Woo. *Answer sets in general nonmonotonic reasoning (preliminary report).* In B. Nebel, C. Rich and W. Swartout, editors, Principles of Knowledge Representation and Reasoning, Proc. of the Third Int'l Conf (KR92), pages 603-614. Morgan-Kaufmann, 1992.

[24] V.Marek and M. Truszczynski. *Autoepistemic Logics.* Journal of the ACM, 38(3):588-619, 1991.

[25] V.Marek and M. Truszczynski. *Revision specifications by means of programs.* In C. MacNish, D. Pearce and L. M. Pereira, editors, JELIA '94, volume 838 of LNAI, pages 122-136. Springer-Verlag, 1994.

[26] V.Marek and M. Truszczynski. *Revision programming.* Research report, University of Kentucky, 1993.

[27] V.Marek and M. Truszczynski. *Revision programming, database updates and integrity constraints.* In Proc. of the 5th International Conference on Database Theory - ICDT 95, pages 368-382. Springer-Verlag, 1995.

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

[29] Isaaco Newtono. *Philosophiæ Naturalis Principia Mathematica.* Editio tertia aucta & emendata. Apud Guil & Joh. Innys, Regiæ Societatis typographos. Londini, MDCCXXVI. Original quotation: *"Corpus omne perseverare in statu suo quiescendi vel movendi uniformiter in directum, nisi quatenus illud a viribus impressis cogitur statum suum mutare."*.

[30] L. M. Pereira and J. J. Alferes. *Well founded semantics for logic programs with explicit negation.* In B. Neumann, editor, European Conf. on AI, pages 102-106. John Wiley & Sons, 1992.

[31] L. M. Pereira, J. J. Alferes and J. N. Aparício. *Counterfactual reasoning based on revising assumptions.* In Ueda and Saraswat, editors, Int LP Symp., pages 566-577. MIT Press, 1991.

[32] L. M. Pereira, J. N. Aparício and J. J. Alferes. *Non-monotonic reasoning with well founded semantics.* In Koichi Furukawa, editor, 8th Int. Conf. on LP, pages 475-489. MIT Press, 1991.

[33] L. M. Pereira and J. J. Alferes. *Nonmonotonic reasoning with logic programming.* Journal of Logic Programming, 17:227-264, 1993.

[34] T. Przymusinski and H. Turner. *Update by means of inference rules.* In V. Marek, A. Nerode, and M. Truszczynski, editors, LPNMR'95, volume 928 of LNAI, pages 156-174. Springer-Verlag, 1995.

[35] M. Winslett. *Reasoning about action using a possible models approach.* In AAAI'88, pages 89-93, 1988.

[36] C. Witteveen and W. Hoek. *Revision by communication.* In V. Marek, A. Nerode and M. Truszczynski, editors, LPNMR'95, volume 928 of LNAI, pages 189-202. Springer-Verlag, 1995.

[37] C. Witteveen and W. Hoek and H. Nivelle. *Revision of non-monotonic theories: some postulates and an application to logic programming.* In C. MacNish, D. Pearce and L. M. Pereira, editors, JELIA '94, volume 838 of LNAI, pages 137-151. Springer-Verlag, 1994.

Appendix A

WFSX^g

In [13], the authors extend the paraconsistent version of the WFSX, namely the WFSX$_p$, to the case where the programs to be dealt with allow for default negated literals in the heads of rules, ie. general programs. They named the semantics WFSX^g_p. Since we are not interested in paraconsistent semantics, we will present here what would be the extension of the WFSX to the general program case. It will be designated as WFSX^g.

Definition A.0.1 (General Logic Program) *Given a language $\mathcal{L}$, a general logic program is a set of rules of the form*

$$G_0 \leftarrow G_1, ..., G_m \qquad (m \geq 0)$$

where each G_i $(0 \leq i \leq m)$ is either an objective or a default literal with respect to the underlying language $\mathcal{L}$.

The intended meaning of a rule *not* $A \leftarrow Body$ is "If the body is true then not A must be true". The motivation of Damásio and Pereira, when dealing with general programs, was to avoid having new fixpoint definitions, but still capture the semantics. They achieved that by presenting the following transformation that converts any general logic program into an extended one, without loss of generality.

Definition A.0.2 (P^{not} transformation) *Let P be a general logic program. The extended logic program P^{not} is constructed as follows, where A is an atom:*

1. *For each literal A(resp. $\neg A$) in $\mathcal{H}(P)$, program P^{not} contains the rule '$A \leftarrow A^p$' (resp. '$\neg A \leftarrow A^n$'), where A^p (resp. A^n) is a new objective literal not in $\mathcal{H}(P)$.*

2. *For each rule of the form '$A \leftarrow Body$' (resp. '$\neg A \leftarrow Body$') program P^{not} contains the rule '$A^p \leftarrow Body$' (resp. '$A^n \leftarrow Body$').*

3. *For each rule of the form 'not $A \leftarrow Body$' (resp. 'not $\neg A \leftarrow Body$') program P^{not} contains the rule '$\neg A^p \leftarrow Body$' (resp. '$\neg A^n \leftarrow Body$').*

Definition A.0.3 (WFSX^g) *Let P be a general logic program. The semantics of P is given by the partial stable models of P^{not} restricted to the language of P. Each such model will be called a general partial stable model (PSM^g). The F-least PSM^g is dubbed the general well founded model (WFM^g).*

Appendix B

A Prolog Interpreter for Program Updates

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                      %
%     Meta-interpreter for Updating programs.          %
%                                                      %
%                    Joao Leite                        %
%                  Vitor Nogueira                      %
%                       1997                           %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/*
 The semantics used is WFSX.

 Syntax:

 Rules are of the form:
        Head <- Body.
 where Head is a literal and Body a conjunction of
 literals separated by ','.

 To write facts, simply ommit the body and the <-

 A literal is either an objective literal 'L', or a
```

```
 default negated objective literal 'not L'.
 An objective literal is an atom 'A' or an explicit
 negated atom '- A'.

 The initial program is allways empty. To consider
 an initial program P simply update empty by P.

 A file may contain several update programs separated
 by 'newProgram.'(see example).

 Warning: this metainterpreter was conceived for
 ground programs; use predicates with variables at your
 own risk.

 Acknowledgements: This meta-interpreter was based on
 another one developped by Dr. Jose Alferes
*/

/*
 Usage:

 clean -> resets the knowledge base (i.e retracts all
          updates made until the present time,
          including the initial program).

 updateF(FileName) -> updates the current knowledge base
                      (that may itself be the result of
                      several updates) with the update
                      program(s) in the file FileName.

 update(List) -> update the current knowledge base with
                 the list of rules List. Here, each
                 rule must be inside parenthesis. One
                 can have t-literals in the head,
                 meaning that that rule is to be added
```

```
                 in the refered update level.

 listUp -> Lists the current state of the knowledge
           base, i.e. all update programs.

 listUp(N) -> Lists the update program of level N.

 saveUp(FileName) -> Saves the current state (including
                     all updates made) into the file
                     FileName. The file is saved with
                     the syntax described above,so that
                     it can be used later for updateF/1.

 current(N) -> Returns in N the level of the latest
              update + 1.

 demo(G/T) -> demo of goal G when current time is T.
              G can be an objective literal 'L', or
              its default 'not L'. T can either be a
              number or 'now' (refering to the number
              of the actual update level). demo(G/now)
              proves G at the latest time level.
*/

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                Operators definition                 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- op(950,fy , not ).          % Negation as failure
:- op(950,fy , '-' ).          % Expicit Negation
:- op(1110,xfy, '<-' ).        % Rule symbol
:- op(1110,yf , '<-' ).        % Rule symbol
:- op(1110,fy , '<-' ).        % Rule symbol

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%                Erasing all updates                %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clean :-
        retractall(current(_)),
        retractall(( _ <- _)).

:- dynamic current/1.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%    Updating from a file with several programs     %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

updateF(FileName) :-
        getNumberOfUpdate(C),
        see(FileName),
        loadClauses(C),
        seen.

loadClauses(CR) :-
        read(C),
   ( C = end_of_file -> true;
   ( processClause(C,CR,NewCR), loadClauses(NewCR) ) ).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                Updating from a list              %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

update(L) :-
        getNumberOfUpdate(C),
        update(L,C).

update([],_).
update([R|T], C) :-
        processClause(R,C,NewC), update(T,NewC).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                Processing one clause              %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

processClause(newProgram,_,C) :- !,
        getNumberOfUpdate(C).
processClause(R,C,C) :-
        makeRule(R,C,Head,Body),
        assert((Head <- Body)).

makeRule( (H <- B), N, NH, NB ) :- !,
        addArgsH(H,N,C,NH),
        makeBody(B,C,NB).
makeRule( Fact, N, NF, true) :-
        addArgsH(Fact,N,_, NF).

makeBody((A,B),C,(NA,NB)) :- !,
        addArgsB(A,C,C,NA), makeBody(B,C,NB).
makeBody(A,C,NA) :-
        addArgsB(A,C,C,NA).

addArgsH(not - P/before,N,C, NP) :- !,
        Bef is N - 1,
        addArgs(P,Bef,C,n,u, neg,NP).
addArgsH(- P/before,N,C,NP) :- !,
        Bef is N - 1,
        addArgs(P,Bef,C,n, u, pos,NP).
addArgsH(not P/before,N,C, NP) :- !,
        Bef is N - 1,
        addArgs(P,Bef,C,p,u, neg, NP).
addArgsH(P/before,N,C,NP) :- !,
        Bef is N - 1,
        addArgs(P,Bef,C,p,u, pos, NP).
```

```
addArgsH(not - P/T,_,C, NP) :- !,
        addArgs(P,T,C,n, u, neg, NP).
addArgsH(- P/T,_,C,NP) :- !,
        addArgs(P,T,C,n, u, pos,NP).
addArgsH(not P/T,_,C, NP) :- !,
        addArgs(P,T,C,p, u, neg, NP).
addArgsH(P/T,_,C,NP) :- !,
        addArgs(P,T,C,p,u, pos, NP).

addArgsH(not - P,N,C, NP) :-
        addArgs(P,N,C,n, u, neg, NP).
addArgsH(- P,N,C,NP) :-
        addArgs(P,N,C,n,u, pos,NP).
addArgsH(not P,N,C, NP) :-
        addArgs(P,N,C,p, u, neg, NP).
addArgsH(P,N,C,NP) :-
        addArgs(P,N,C,p,u, pos, NP).

addArgsB(P/now,_,_,NP) :- !,
        current(N), C is N - 1,
        addArgs(P,C,C,0,0,pos,NP).
addArgsB(P/before,_,_,NP) :- !,
        current(N), Bef is N - 2,
        addArgs(P,Bef,Bef,0,0,pos,NP).
addArgsB(P/T,_,_,NP) :- !,
        addArgs(P,T,T,0,0,pos,NP).
addArgsB(P,N,C,NP) :-
        addArgs(P,N,C,0,0,pos,NP).

addArgs(not -P,N,C,X, Y, Z, not -NP) :- !,
        addArgs(P, N, C, X, Y, Z,NP).
addArgs( -P,N,C,X, Y, Z, -NP) :- !,
        addArgs(P, N, C, X, Y, Z,NP).
addArgs(not P,N,C,X, Y, Z, not NP) :- !,
        addArgs(P, N, C, X, Y, Z,NP).
```

```
addArgs(P, N, C, X, Y, Z, NP) :-
        P =.. [Pred|Arguments],
        NP =.. [Pred,N, C, X,Y,Z|Arguments].

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   Getting the number of the next update program     %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

getNumberOfUpdate(1) :-
        \+ current(_), assert(current(2)).
getNumberOfUpdate(C) :-
   retract(current(C)), C1 is C+1, assert(current(C1)).

/* ------------- demo predicate -------------- */
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%               demo( Goal/Time )                     %
% use Time = now if you want to use the most recent   %
% update program.                                     %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

demo(not -G/now):-
        current(C), T is C-1,
        addArgs(G,T,T,n,0, neg, NG),
        demo(NG, [],[],t),!.
demo(not - G/now) :-
        current(C), T is C-1,
        addArgs(G,T,T,n,0,pos,NG), !,
        demo(not NG,[],[],t).
demo(- G/now) :-
        current(C), T is C-1,
        addArgs(G,T,T,n,0,pos,NG), !,
        demo(NG,[],[],t).
demo(not G/now):-
        current(C), T is C-1,
        addArgs(G,T,T,p,0,neg, NG),
```

```
        demo(NG, [],[],t),!.
demo(not G/now) :-
        current(C), T is C-1,
        addArgs(G,T,T,p,0,pos,NG), !,
        demo(not NG,[],[],t).
demo(G/now) :-
        current(C), T is C-1,
        addArgs(G,T,T,p,0,pos,NG),!,
        demo(NG,[],[],t).

% T stands for any time-stamp
demo(not -G/T):-
        addArgs(G,T,T,n,0, neg, NG),
        demo(NG, [],[],t),!.
demo(not - G/T) :-
        addArgs(G,T,T,n,0,pos,NG), !,
        demo(not NG,[],[],t).
demo(- G/T) :-
        addArgs(G,T,T,n,0,pos,NG), !,
        demo(NG,[],[],t).
demo(not G/T):-
        addArgs(G,T,T,p,0,neg, NG),
        demo(NG, [],[],t),!.
demo(not G/T) :-
        addArgs(G,T,T,p,0,pos,NG), !,
        demo(not NG,[],[],t).
demo(G/T) :-
        addArgs(G,T,T,p,0,pos,NG),
        demo(NG,[],[],t).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%        demo( Goal, Ancestors lists, Mode)           %
%                                                     %
% If Mode=true then tests if Goal is true             %
%     otherwise tests if Goal is true or undef.       %
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

demo( true, _, _, _ ) :- !.

demo( (G,Cont), AnsL, AnsG, M ) :-
        !, demo(G, AnsL, AnsG,M),
        demo(Cont, AnsL, AnsG,M).

demo( not G, _ , AnsG, t ) :-
           !, \+ demo(G,[],AnsG,tu).

demo( not G, _ , AnsG, tu ) :-
           !, \+ demo(G,AnsG,AnsG,t).

demo( G, Ans, _,  _ ) :-
       loop_detect( G, Ans ), !, fail.

demo( G, AnsL, AnsG, t ) :-
        (G <- Body),
        demo( Body, [G|AnsL], [G|AnsG], t ).

demo( G, AnsL, AnsG, tu ) :-
        compl_neg(G, NG),
        (G <- Body),
        demo( (Body,not NG), [G|AnsL], [G|AnsG], tu ).

%%%%%%  rules specific to the update semantics %%%%%%%%

demo( - G, AnsL, AnsG, M ) :-
        checkIndex(G,0,n_p_0), !,
        insertIndex(G,NG,n, n_p_0),
        demo(NG, AnsL, AnsG, M ).

demo( G, AnsL, AnsG, M ) :-
        checkIndex(G,0, n_p_0), !,
```

```
        insertIndex(G,NG,p,n_p_0),
        demo(NG, AnsL, AnsG, M ).

demo(G, AnsL, AnsG, M):-
        checkIndex(G, 0, u_0),
        \+ checkIndex(G, 0, n_p_0),
        checkIndex(G, T, t_undef),
        nonvar(T),
        insertIndex(G, NG, u, u_0),
        demo(NG, AnsL, AnsG, M).

demo(G, AnsL, AnsG, M):-
        checkIndex(G, u, u_0),
        checkIndex(G, neg, pos_neg_0),
        checkIndex(G, T, t_undef),
        nonvar(T),
        checkIndex(G, I, n_p_0),
        inverse(I, InvI),
        insertIndex(G,NG, InvI, n_p_0),
        insertIndex(NG, NNG, pos, pos_neg_0),
        demo(NNG, AnsL, AnsG, M).

%%%%%%%%%%%%%%%%% Defining rules %%%%%%%%%%%%%%%%%%%%%%

demo(G, AnsL, AnsG, M):-
        checkIndex(G, pos, pos_neg_0),
        \+ checkIndex(G, 0, n_p_0),
        checkIndex(G, 0, u_0),
        decreaseTime(G, NG, T),
        T >=0,
        demo(NG, AnsL, AnsG, M),
        insertIndex(G, NG1, u, u_0),
        insertIndex(NG1, NG2, neg, pos_neg_0),
        demo(not NG2, AnsL, AnsG, M).
```

```
demo(G, AnsL, AnsG, M):-
        checkIndex(G, neg, pos_neg_0),
        \+ checkIndex(G, 0, n_p_0),
        checkIndex(G, 0, u_0),
        decreaseTime(G, NG, T),
        T >=0,
        demo(NG, AnsL, AnsG, M),
        insertIndex(G, NG1, u, u_0),
        insertIndex(NG1, NG2, pos, pos_neg_0),
        demo(not NG2, AnsL, AnsG, M).

% Loop Detection
loop_detect(X,[Y|_]) :- X == Y, !.
loop_detect(X,[_|T]) :- loop_detect(X,T).

% Explicit negation
compl_neg( - G, G ) :- !.
compl_neg( G, - G ).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                 Auxiliary predicates                %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

decreaseTime(G,NG,T) :-
        G =.. [Name,TT|Args],
        T is TT - 1,
        NG =.. [Name,T|Args].

inverse(n,p).
inverse(p,n).

checkIndex(G,I,n_p_0) :-
        G =.. [_,_,_,I|_].
checkIndex(G,I,u_0) :-
```

```
        G =.. [_,_,_,_, I|_].
checkIndex(G,I,pos_neg_0) :-
        G =.. [_,_,_,_,_,I|_].
checkIndex(G, T, t_undef):-
        G =.. [_,T|_].

dropIndex(G,NG,n_p_0) :-
     G =.. [Name,A,B,_|Args],
     NG =.. [Name,A,B,0|Args].
dropIndex(G,NG,u_0) :-
     G =.. [Name,A,B,C,_|Args],
     NG =.. [Name,A,B,C,0|Args].
dropIndex(G,NG,pos_neg_0) :-
     G =.. [Name,A,B,C,D,_|Args],
     NG =.. [Name,A,B,C,D,0|Args].

insertIndex(G,NG,I,n_p_0) :-
     G =.. [Name,A,B,_|Args],
     NG =.. [Name,A,B,I|Args].
insertIndex(G,NG,I,u_0) :-
     G =.. [Name,A,B,C,_|Args],
     NG =.. [Name,A,B,C,I|Args].
insertIndex(G,NG,I, pos_neg_0) :-
     G =.. [Name,A,B,C,D,_|Args],
     NG =.. [Name,A,B,C,D,I|Args].

/* ------------- listing and saving states --------- */
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                 listing of update programs          %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

listUp :-
        current(T), C is T - 1,
        write('Current level is '), write(C), nl,
        listUpUntil(C).
```

```
listUpUntil(0) :- !.
listUpUntil(N) :-
        listUp(N),
        N1 is N - 1,
        listUpUntil(N1).

listUp(N) :-
        nl, write('%Program of level '), write(N), nl,
        findall(C,clauseLevel(C,N),L),
        processList(L).

processList([]).
processList([(H <- true)|L]) :- !,
        writeHead(H), write('.'), nl,
        processList(L).
processList([(H <- B)|L]) :-
        writeHead(H), write(' <- '),
        writeBody(B), write('.'), nl,
        processList(L).

writeBody((A,B)) :- !,
        writeElem(A), write(', '), writeBody(B).
writeBody(A) :- writeElem(A).

writeHead(- H) :- !,
        write('not '), writeHead(H).
writeHead(H):-
        checkIndex(H,n,n_p_0),!,
        write('- '),
        writeElem(H).
writeHead(H):-
        writeElem(H).

writeElem(not H) :- !,
```

```
        write('not '), writeElem(H).
writeElem(H):-
        H =.. [Name,_,C,_|Args],
        HH =.. [Name|Args],
        writeTime(HH,C).

writeTime(G,C) :- var(C), !, write(G).
writeTime(G,C) :- write(G/C).

%getting one clause of level N
clauseLevel((H <- B),N) :-
        (H <- B), H =.. [_,N,_|_].
clauseLevel((- H <- B),N) :-
        (- H <- B), H =.. [_,N,_|_].

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%            saving the state into a file             %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

saveUp(File) :-
        tell(File),
        saveLevels,
        told.

saveLevels :-
        current(T), C is T - 1,
        write('% This file was generated by saveUp.'),
        nl, nl, write('% There are '), write(C),
        write(' levels'), nl,
        listUpUntil(1,C).

listUpUntil(C,C) :- !,
        listUp(C).
listUpUntil(N,C) :-
        listUp(N), N1 is N + 1, nl,
```

```
write('newProgram.'), nl,
listUpUntil(N1,C).
```