Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Evolving Reactive Logic Programs

Federico Banti

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

Lisboa
(2007)

This dissertation was prepared under the supervision of
Professor José Júlio Alferes,
of the Faculdade de Ciências e Tecnologia,
Universidade Nova de Lisboa,
and of Professor Antonio Brogi, of the Dipartimento di Informatica,
Università di Pisa.

To Sara

Acknowledgements

I would like to acknowledge Fundação para a Ciência e Tecnologia that, through POS_Conhecimento Programme, via FEDER structural funds for the European Union, gave me the PhD grant (no. SFRH/BD/ 13716/2003) that financially supported the work from October 2003 to September 2007. I thank the project REWERSE (number 506779), funded by the European Commission within the 6th Framework Programme, for supporting the last month of my work.

For giving me working conditions, I acknowledge the Departamento de Informática of Faculdade de Ciências e Tecnologia of Universidade Nova de Lisboa, and the Centre for Artificial Intelligence – Centria.

Sumário

Um dos principais objectivos da Inteligência Artificial (IA) é o de fornecer capacidade de raciocínio a máquinas. Em muitas aplicações, em especial aquelas que se desenvolvem em ambientes dinâmicos, as capacidades de representação do conhecimento e de raciocínio desenvolvidas na IA devem lidar com conhecimento que possa ser constantemente actualizado.

Além disso, na maioria dos casos, uma aplicação de IA, não deve apenas ter capacidades de representação do conhecimento e raciocínio, mas deve também poder reagir a acontecimentos externos, função do conhecimento que tem representado. Para se adaptar às mudanças externas, uma aplicação de IA pode ter que, por vezes, executar auto-actualizações tanto quer da sua base de conhecimentos como do seu comportamento reactivo. A reacção adequada a uma solicitação externa exige, em muitos casos, não só conhecimento estático sobre o meio ambiente, mas também conhecimento sobre os resultados das acções que podem ser executadas.

A Programação em Lógica está entre os mais estudados paradigmas para representar o conhecimento, baseando-se numa linguagem baseada em regras, simples e intuitiva, e com uma semântica declarativa rigorosa bem como metodologias de dedução automática eficazes.

O objetivo desta tese é desenvolver uma linguagem baseada na programação em lógica, para o desenvolvimento de aplicações de IA, capaz de efectuar raciocínio sobre uma base de conhecimento, de manipular actualizações deste conhecimento básico, de reagir aos acontecimentos externos e de evoluir.

Para alcançar este resultado começamos o nosso trabalho, tendo como ponto de partida a linguagem já existente de programas em lógica dinâmicos (do inglês "Dynamic Logic Programs") isto é, de sequências de programas em lógica em que o primeiro programa representa o conhecimento inicial e os restantes as actualizações subsequentes. A ideia principal desta linguagem é que uma regra de um programa pode ser desconsiderada sempre que surge uma potencial contradição com o que é afirmado por uma regra de um programa posterior. Com base em programação em lógica dinâmica foram desenvolvidas linguagens de actualização de programas em lógica que permitem que programas possam evoluir, actualizando-se.

O nosso primeiro contributo é o de desenvolver uma mais sólida base teórica para programas em lógica dinâmicos. Começamos por refinar o actual modelo baseado

na semântica de modelos estáveis de programas em lógica dinâmicos, definindo assim uma nova semântica que resolve alguns comportamentos contra-intuitívos exibidos pelas anteriores semânticas e que satisfazem propriedades importantes. Esta nova semântica é alargada ao caso mais geral de programas dinâmicos "multi-dimensionais", em que as sequências de programas em lógica são substituídas por uma ordem parcial de multi-conjuntos de programas. O próximo passo foi o de definir uma semântica bem-fundada (do inglês "well founded semantics") para programas em lógica dinâmicos que aproxima a semântica baseada em modelos estáveis, e que tem uma menor complexidade computacional.

Tendo estabelecido esta firme base teórica, estudamos como a linguagem de actualizações de programas em lógica Evolp pode ser utilizada para descrever e raciocinar sobre efeitos de acções, e qual é a sua expressividade e vantagens relativamente a outras linguagens de descrição de acções (do inglês "action description languages"). Tendo resolvido os problemas básicos do raciocínio e actualizações, definimos uma linguagem de Evento-Condição-Acção, chamada ERA (vindo a abreviatura do inglês "Evolving Reactive Algebraic logic programs"), baseada em programas em lógica dinâmicos, e que estende o Evolp através do reforço das sua capacidades reactivas: com eventos, regras reactivas e com acções. Contrastando com as actuais linguagens de raciocínio sobre acções, desenvolvemos na tese uma abordagem integrada para programação de comportamento reactivo, totalmente declarativa, capaz de lidar com raciocínio e execução de uma só vez, juntamente com características auto-evolutivas inéditas.

Summary

One of the main goals of the Artificial Intelligence (AI) field is to provide reasoning capabilities to computers. Within many applications, in particular those settled in a dynamic environment, the knowledge representation and reasoning capabilities developed within AI must deal with knowledge that is constantly updated.

Moreover, in most of the cases, an AI application is not only bound to reasoning activities, but also to react to external events on the the basis of its knowledge. To adapt itself to external changes, an AI application must sometimes perform self updates both to its knowledge base and to its reactive behavior. A proper reaction to an external solicitation requires, in many cases, not only static knowledge about the environment, but also about the outcomes of actions that can be undertaken.

Logic programming is among the most studied paradigms for representing knowledge, relying on a simple and intuitive rule-based logic language and being provided with rigorous and declarative semantics, as well as effective methodologies for automatic deduction.

The goal of this thesis is to develop a logic programming based framework for developing AI applications capable of reasoning on the basis of a knowledge base, of handling updates to this knowledge base, of reacting to external events and of evolving.

To achieve this result we start our investigation from the existing framework of dynamic logic programs, i.e. sequences of logic programs where the first program represents the initial knowledge base and the remaining ones the subsequent updates. The main idea of this framework is that a rule in a program can be rejected whenever a potential contradiction arises with what was asserted by a rule in a subsequent program. On the base of dynamic logic programs, the so called logic programs updates languages, allowing programs to evolve by updating themselves, have been developed.

Our first contribution is to settle the dynamic logic programming framework on more solid theoretical bases. We start by refining the existing stable model based semantics for dynamic logic programs, obtaining a new refined semantics that solves some counterintuitive behavior shown by the previous semantics and that satisfies important properties. The refined semantics is also extended from dynamic logic programs to the more general case of multidimensional dynamic logic programs, where

sequences of logic programs are replaced by partially ordered multisets of programs. The next step is to define the well founded semantics of dynamic logic programs that approximates the refined semantics and has a lower computational complexity.

Having established firmer theoretical bases, we investigate how the existing logic programming updates language Evolp can be used for describing and reasoning about the effects of actions, and what is its expressivity and advantages w.r.t. other action description languages. Having solved the basic problems of reasoning and updates, we define an Event-Condition-Action language called ERA (after Evolving Reactive Algebraic logic programs), based on the semantics of dynamic logic programs, that extends Evolp by enhancing its reactive capabilities with events, reactive rules and actions. Contrasting with the existing frameworks for reasoning about actions, for programming reactive behavior, we develop an integrated, fully declarative, framework for dealing with the above issues of reasoning and execution at once, together with inedit self-evolving features.

Contents

1 Introduction and outline **1**

1.1 Reasoning, reactivity and evolution for Artificial Intelligence applications 2

1.2 An evolving logic programming Event Condition Action language . . . 3

1.3 Outline of the thesis . . . 5

1.3.1 A refined well-supported semantics for DyLPs . . . 5

1.3.2 A well founded semantics for DyLPs . . . 8

1.3.3 Updates languages and reasoning about actions . . . 9

1.3.4 Evolving Reactive Algebraic Programs . . . 10

1.3.5 Appendixes . . . 12

1.4 Comparisons to other approaches . . . 12

1.5 Application areas . . . 14

1.6 List of original contributions . . . 15

2 Background: Logic programs syntax and semantics **17**

2.1 Basic concepts in logic programming . . . 18

2.2 Syntax of logic programs . . . 21

2.3 Interpretations and satisfaction . . . 22

2.4 Classes of logic programs and their semantics . . . 24

2.4.1 Two-valued semantics for normal logic programs . . . 25

2.4.2 The well founded semantics for normal logic programs . . . 27

2.4.3 Stable model semantics for generalized logic programs . . . 29

2.4.4 Well founded semantics for GLPs and ELPs . . . 29

I On the Semantics of Dynamic Logic Programs **33**

3 A refined, well-supported semantics for logic programs updates **35**

3.1 Introduction and motivation . . . 37

3.1.1 Open issues on semantics for DyLPs bases on causal rejections . 39

3.1.2 A principled based approach . . . 41

3.1.3 Beyond linear dynamic logic programs . . . 43

3.1.4 Structure of the chapter . . . 47

3.2 Preliminaries and previous works on updates . . . 48

3.2.1 Well-supported models . . . 50
3.2.2 Multidimensional dynamic logic programs . . . 51
3.3 Refined extensions . . . 52
3.3.1 Refined extensions of generalized logic programs . . . 52
3.3.2 Refined extensions of dynamic logic programs . . . 54
3.4 Refined semantics for dynamic logic programs . . . 56
3.5 Relationship with other semantics for dynamic logic programs . . . 60
3.5.1 Semantics for DyLPs not based on causal rejection . . . 61
3.6 Well-supported models for DyLPs . . . 62
3.7 Further properties of the refined semantics . . . 69
3.8 Refined well-supported semantics for multidimensional dynamic logic programs . . . 73
3.9 Fixpoint characterization for well-supported models of MDyLPs . . . 74
3.10 Relationship to other semantics for MDyLPs . . . 78
3.11 A program transformation for the refined semantics of DyLPs . . . 81
3.12 Transformational semantics for MDyLPs . . . 86
3.13 Concluding remarks . . . 88

4 A well founded semantics for dynamic logic programs 91
4.1 Introduction . . . 93
4.2 The notion of causal rejection for three-valued semantics . . . 95
4.3 The Well founded semantics for DyLPs . . . 97
4.4 Illustrative examples . . . 102
4.5 Properties of the well founded semantics of DyLPs . . . 107
4.5.1 Properties of the fixpoints of $\Gamma\Gamma^R$ . . . 108
4.5.2 Characterization of consistent DyLPs . . . 110
4.6 Relations with the well founded semantics of GLPs . . . 111
4.7 Transformational well founded semantics . . . 111
4.8 Related works . . . 113
4.9 Concluding remarks . . . 115

II Reasoning about and executing actions 117

5 Reasoning about actions 119
5.1 Introduction . . . 120
5.2 An overview of LP updates and action description languages . . . 122
5.2.1 LP updates languages . . . 122
5.2.2 The language Evolp . . . 124
5.2.3 Action languages . . . 125
5.3 Evolving action programs . . . 127
5.3.1 An elaboration of the Yale shooting problem . . . 132

5.4 Relationship to existing action languages . . . 134
5.5 Updates of action domains . . . 136
5.6 Conclusions and future work . . . 140

6 Evolving Reactive Algebraic Programs 143
6.1 Introduction . . . 144
6.1.1 Notation . . . 147
6.2 Outline and syntax of the language . . . 147
6.3 The Semantics of ERA . . . 152
6.3.1 Inferring conclusions on DyLPs . . . 153
6.3.2 Inferring conclusions on ERA systems . . . 154
6.3.3 Execution of actions . . . 156
6.4 Remarks . . . 161
6.4.1 Failure of actions . . . 161
6.4.2 Concurrent execution of actions . . . 161
6.4.3 Examples of frequently used complex actions . . . 163
6.5 Comparisons to other updates languages . . . 164
6.5.1 Relationship with Evolp . . . 165
6.5.2 ERA as an action description language . . . 171
6.5.3 Other updates languages . . . 172
6.5.4 Updates language LUPS . . . 173
6.5.5 Updates language *KUL* . . . 176
6.5.6 Updates language *EPI* . . . 176
6.5.7 Updates language *KABUL* . . . 177
6.6 Comparisons to ECA formalisms and event languages . . . 181
6.6.1 The agent-oriented language DALI . . . 182
6.6.2 Process algebras . . . 186
6.6.3 Snoop and other event languages . . . 187
6.7 Conclusions and open issues . . . 188

7 Conclusions 191
7.1 Current and future work . . . 194

Bibliography 199

A Proofs of theorems from Chapter 3 209
A.1 Properties of the least model . . . 209
A.2 The principle of partial evaluation . . . 211
A.3 Proofs of theorems and propositions . . . 212

B Proofs of theorems from Chapter 4 235

C Proof of theorems from Chapters 5 and 6 **255**

Chapter 1

Introduction and outline

Contents

1.1 Reasoning, reactivity and evolution for Artificial Intelligence applications **2**

1.2 An evolving logic programming Event Condition Action language . **3**

1.3 Outline of the thesis **5**

1.4 Comparisons to other approaches **12**

1.5 Application areas **14**

1.6 List of original contributions **15**

This chapter introduces the main subjects of the thesis and its goals. We begin by illustrating desirable features of a framework for developing AI applications, i.e., from one side, reasoning, reactivity and evolution and, from the other side, the capability of defining and executing complex actions. We proceed by illustrating the framework developed along the thesis for achieving these requirements. An outline of the thesis is then presented detailing the content of the various chapters. The motivation for the thesis is then enforced by first briefly comparing our proposal with existing approaches, in order to show that the latter do not completely match the requirements illustrated at the beginning of the chapter, and then by sketching some potential application areas where our framework could be profitably applied. We end up with a summary list of the contributions of the thesis.

1.1 Reasoning, reactivity and evolution for Artificial Intelligence applications

Research in Artificial Intelligence (AI) is concerned with producing machines to automate tasks requiring intelligent behavior. A distinctive feature of an intelligent behavior is that of being based on the possessed *knowledge*. A first problem to face when implementing AI applications is then how to represent knowledge and how to extract information from such knowledge. In other words, a first brick of an AI application is a *knowledge representation (KR) and reasoning (or inference) system*. The dominant approach in knowledge representation is to define symbolic paradigm based on some form of logic. A symbolic knowledge representation framework usually consists of crude facts (or data) and more sophisticated expressions describing the relations between facts (like, for instance logical implication). Together facts and formulas form the *knowledge base* (KB) of the AI application. Usually facts are referred as the *extensional part* of the KB while formulas represent its *intentional part*.

Many tasks for AI applications also demand to perform some kind of *actions*; hence actions and possibly *the effects of actions* should be representable in the KR framework and the mechanism specifying when an action must be performed must be defined. Moreover, usually the application continually receives external inputs in the form of messages, perceptions, commands and so on. Such inputs can be considered as *events* to which the AI application is supposed to *react* in an intelligent way.

Reactivity is a key feature in dynamic domains where changes frequently occur. Dynamic domains also demand the AI application for taking into account such changes and consequently *update* the KB. The required updates surely involve the extensional part of the knowledge base, but occasionally it may be necessary to update also the intentional part to represent the fact that the very rules of the domain are changed. Moreover, for adapting to the new situation, besides knowledge updates, it might be necessary to update the *behavior* of the AI applications i.e. to update the reactive mechanisms themselves. These updates may be the result of external inputs but it might be necessary for the application to perform actions leading to self-updates. In other words, the AI application is required to have *evolving capabilities*.

A Knowledge representation paradigm for programming reactive and evolving AI applications should hence be able to:

- represent knowledge (facts and formulas) and infer information,
- specify actions that must be executed in response to external events (the reactive behavior of the application),
- reason about the effects of such actions,
- incorporate updates to facts and formulas (knowledge), and behavior, and

- perform self updates as actions.

Despite of the challenging complexity of the capabilities required above, we believe that, for practical applications, we still need to require some additional expressive features. Besides what could be called *basic actions* like, for instance, insertion and deletion of facts and formulas or atomic external actions, a developer may want to specify more sophisticated actions obtained by the basic ones, like “first do this action and then do this other one" or “do both these actions in parallel" and so on. The framework should provide easy tools for combining actions in an easy and modular way allowing to define complex actions in terms of basic ones and then to use such complex actions to define even more complex ones and so on.

1.2 An evolving logic programming Event Condition Action language

Among the existing formalisms for KR, Logic Programming (LP) has a simple logic based syntax, formal declarative semantics and implemented inference systems. The formulas of LP are logic implications (also called rules or clauses) of the form

$$A \leftarrow L_1, \ldots, L_n.$$

and combine an intuitive interpretation with a precise semantical meaning. A *logic program* is any set of such rules. The most well known class of logic programs is that of *normal logic programs* [DPP97, Rei87], i.e. sets of rules of the form above where A (called the *head* of the rule) is an atom and $L_1, \ldots, L_n$ (also known as the *body* of the rule) is a (possibly empty) conjunction of (positive or negative) literals. For the reasons mentioned above, we choose LP as the basic KR language of our framework.

With this choice of basic KR language, and aiming at the general and expressive paradigm for reactive and evolving AI applications, the first step of our research is **the definition of a theory of updates for logic programs**. In this preliminary part, described in Chapter 3 and 4 **we refine and extend previous works on logic programs updates, specifically the framework known as *dynamic logic programs* (DyLPs) [ALP$^+$00a, BFL99, EFST02a, Lei03, LP97, SI99, ZF98, Sef00] leading to a refined stable model-like semantics and to a well founded semantics for DyLPs**. The main idea of logic programs updates is to update a logic program by another (or several) logic programs. In this perspective an old rule can be rejected by a more recent one, the latter encoding an exception to the former.

Contextually with the theory of LP updates, languages for developing LP-like programs capable of performing self-updates, and with an interface for external inputs, have been defined (like, for instance, *LUPS* [APPP02], *EPI* [EFST01], *KUL* [Lei03], and *KABUL* [Lei03]). Such languages possessed “in nuce" many of the ideas for a KR-

paradigm for evolution and reactivity. Among these languages, a very expressive and particularly simple one is *Evolp* [ABLP02]. We focus our investigation on this language. **A first result is to show how Evolp can be used to reason about the effects of actions** and to encode existing paradigms for reasoning about actions.

However, when facing the problem of specifying the execution of actions, rather then simply reasoning about such execution, we find the expressivity of Evolp paradigm less than adequate for the purpose we have in mind.

First of all, the ideas of events and reactions to events are not clearly outlined in the Evolp framework. It is also problematic to express behavior like "react if this event occurs followed by this other event". In other words it is problematic to express *complex events*. Even more problematic, if at all impossible, is to specify complex actions like "execute this task and when you finish execute this other task" or "if this condition holds do this action, otherwise do this other action" and, moreover, to incrementally define complex actions on the basis of simpler ones.

Our conclusion is that to reach the needed expressive capabilities the natural way is to **extend the framework including concepts beyond the usual boundaries of Logic Programming**.

In our opinion, the most natural setting to encode reactivity is that of *Event Condition Action (ECA) languages*. The main construct of an ECA language is that of reactive (also called ECA) rules of the form:

On *Event* **If** *Condition* **Do** *Action*

That literally means: when *Event* occurs and if the *Condition* is satisfied then do the *Action*. The intuitive reading of these rules and their clear reactive nature makes ECA-languages the natural candidate to express reactivity. Moreover, being a rule-based framework, ECA rules can merged with any LP-framework in a comparatively easy way. Syntactically, all we have to do is to allow both inference rules and ECA rules in the same program. We find that it is quite natural to use the mechanism of (self) update of DyLPs and Evolp to allow an ECA program to specify updates to inference rules (encoding knowledge) and to ECA rules (encoding reactive behavior). For defining complex actions and events in an incremental an modular way, the best solution around is, in our opinion, to rely on *algebras of operators*. An algebra allows to combine basic elements (events or actions) to form more complex ones, by operators expressing concepts like "first this occurs and then this other occurs" or "do this and then do that" and so on. Operators can be applied also to complex elements to define new, more complex elements (incremental and modular definition). It is also possible then to *define* a new action or event as an arbitrary algebraic expression.

By specifying two distinct algebras, for actions and for events, we obtain the language ERA (after Evolving Reactive Algebraic programs). An ERA program is

a collection of either inference or ECA rules and both events and action definitions. Although adopting a rather different syntax, we will show that ERA can be considered as a direct extension of Evolp, hence there is a substantial continuity between this and the previous steps of the work.

1.3 Outline of the thesis

As it emerges from the discussion above, our work is ideally divided in two phases, a first phase devoted to utterly develop the existing framework on LP updates and a second one devoted to develop concepts from other areas of research and integrate them in the existing framework. This ideal division is reflected in the structure of the thesis. Chapter 2 is an overview of classes of logic programs and their semantics that are mostly significant for this work. Chapters 3 to 4 are LP updates focused, while Chapters 5 and 6 develop and illustrate the actions language EAPs, and the ECA language ERA. Let us consider in a more detailed way the contribution of the various chapters.

PART I

1.3.1 Chapter 3: A refined well-supported semantics for DyLPs

This chapter illustrates the existing stable model-like semantics for LP updates [ALP+00b, BFL99, EFST02a, Lei97, Lei03, LP98, SI99, ZF98], and some known limitations of these frameworks. It then proposes new theoretical foundations and a refined stable model semantics for logic programs updates. The common idea to all the analyzed approaches is to update a logic program P_1 by another logic program P_2 or by a sequence of logic programs $P_2, \ldots, P_n$ where P_1 represents the initial knowledge and P_2 (resp. $P_2, \ldots, P_n$) is the update (resp. are the various updates). Together, the sequence $P_1, \ldots, P_n$ is known as *dynamic logic program* (DyLP).

Another common idea is that of relying on more general forms of programs than normal logic programs allowing negative literals also in the head of rules rather than simply in their bodies (as in normal LPs) [GL90, LW92]. In particular, in our approach, the programs in the sequence are *generalized logic programs* [LW92] i.e. sets of rules $L \leftarrow B$ where L is a (positive or negative) literal (not just an atom as in normal LPs) and B is a conjunction of literals.

Having the possibility to express both positive and negative conclusions in the head of rules, conflicts of consistency may emerge among rules in case the head of one rule is the opposite of the head of another rule. Semantics of LP updates constructively use these conflicts to encode changes to the initial situation. More precisely, an old rule $L \leftarrow B_1$ may be invalidated by a more recent one (i.e. a rule in a more recent update) $not\, L \leftarrow B_2$. The approaches analyzed are based on the *causal rejection principle* [EFST02a, Lei03] that states: "a rule is rejected if and only if there is a rule whose head

is in conflict with the head of the previous one and whose body is true".

Let us provide an intuitive example to clarify the concept.

Example 1.3.1 *The initial program*

$$P_1: \quad wet \leftarrow rain.$$
$$rain.$$

simply states that if it rains then it is wet and that currently it rains. We update this program with the following:

$$P_2: \quad not\, rain \leftarrow sun.$$
$$sun.$$

stating that it does not rain if there is the sun, and indeed there is the sun. According to the known semantics for updates based on causal rejection, the (stable) model associated to P_1 is $\{wet, rain\}$ stating that it rains and hence is it is wet. However, if we consider the DyLP P_1, P_2 we have a rule in P_2 which is in conflict with the rule (in this case it is simply a fact) rain and whose body is true and hence the fact rain is rejected.

According to this idea, the stable model of the DyLP above, after the update, is

$$\{sun, not\, rain, not\, wet\}$$

An intuitive way to consider DyLPs is that later rules encode *exceptions* to the previous ones.

At the moment we started our research there were several very similar semantics for DyLPs (as show in [Hom04]) based on a stable models approach and the causal rejection principle. These semantics differ in some cases involving DyLPs [EFST01, Lei03]. More precisely, some semantics admit less stable models than others. A problematic aspect is that all these semantics provide counterintuitive results on some programs. More precisely, *none of the cited semantics is immune to tautologies* [Lei03] i.e. the addition of rules of the form $L \leftarrow L$ sometimes changes the semantics of a DyLP by introducing new models. Among those semantics the *dynamic stable model semantics* [ALP+98a] is the one with less problems with tautologies and also the one admitting less models. Nevertheless even this semantics is not completely immune to tautologies. Let us provide an example

Example 1.3.2 *Consider the program P_1 describing some knowledge about the sky. At each moment it is either day time or night time; we can see the stars whenever it is night time and there are no clouds; and currently it is not possible to see the stars.*

$$P_1: \quad day \leftarrow not\, night.$$
$$night \leftarrow not\, day.$$
$$stars \leftarrow night, not\, cloudy.$$
$$not\, stars.$$

The only dynamic stable model of this program is $\{day\}$. Suppose now the program is updated with the following tautology:

$$P_2: \quad stars \leftarrow stars.$$

This tautological update introduces a new dynamic stable model. Indeed, besides the intuitively correct dynamic stable model $\{day\}$, the dynamic stable model semantics also admits the model $\{night, stars\}$. Furthermore, these results are shared by all other existing semantics for updates based on causal rejection [ALP$^+$98b, BFL99, EFST02a, Lei03, LP97, LP98]. We argue that this behavior is counterintuitive as the addition of the tautology in P_2 should not add new models.

Hence, several very similar semantics coexist, and it is not clear which should be the *right* semantics for DyLPs since the only known criterium of choice is immunity to tautologies. But first, not even the best candidate is completely immune to tautologies and second, tautologies are just an example, an alarm that some more *general principle* is violated. However, it is not clear what is the general principle whose violation leads to counterintuitive behavior with tautologies.

Our first contribution is to find such a general principle and then to provide a semantics satisfying this principle. As a first step we define the *refined extension principle* which specifies cases when the addition of a set of rules to a DyLP should not lead to more models. The counterintuitive examples involving tautologies are just special cases of the violation of the principle. Then we define a refinement of the dynamic stable model semantics, called the *refined semantics for DyLPs* that complies with the principle and, consequently, is immune to tautologies.

This first work still leaves some important open issues. First of all, the formulation of the refined extension principle, as shall be seen, is too technical and linked to the formal definition of the semantics. Moreover, it does not characterize completely the semantics and indeed it is easy to define fake semantics for DyLPs that nevertheless satisfies the principle. The second aspect is that the definition provided for DyLPs does not extend to the case of *Multidimensional dynamic logic programs* (MDyLPs) a generalization of DyLPs allowing updates not only as a sequence but more in general as acyclic diagraphs. Moreover, also the refined extension principle is not extendable to cover the new case. Although this is not directly linked to the rest of the work, it would be odd to have a proper refinement for DyLPs but with the question left unsolved in the multidimensional case, especially because the other semantics had extensions to the multidimensional case [LAP00, BFL99].

We solve both the problems by extending the concept of *well supported models* [Rei87] to DyLP and MDyLPS. Well supported models are a non-fixpoint semantics for normal logic programs based on functions from literals to natural numbers, known as *level mappings* . It turns out that supported models are an alternative characterization of the stable model semantics of normal logic programs. We present an intuitive extension of well supported model to the case of DyLPs and then we show that, as it happens in the case of normal logic programs, *well supported models for DyLPs are*

an alternative characterization of the refined semantics. This provides a strong hint that the refined semantics is indeed the proper extension of the stable model semantics to DyLPs. Moreover, well supported models can be immediately extended to MDyLPs thus obtaining the desired extension of the refined semantics to MDyLPs. Given the equivalence between the two characterizations we also refer to the refined semantics as the well supported semantics for DyLPs (and MDyLPs).

The chapter is completed by a program transformation that allows to find for any DyLP (or MDyLP) a normal logic programs whose stable models correspond to the well supported models of the original program, thus allowing to use the existing software for computing the stable model semantics also for computing the well supported semantics of DyLPs and MDyLPs.

1.3.2 Chapter 4: A well founded semantics for DyLPs

The refined semantics for DyLPs extends the stable models semantics for DyLPs. However, the requirements of some applications domains demand a different choice of a basic semantics such as the well founded semantics [GRS91]. The well founded semantics is a skeptical 3-valued logic approximation of the stable model semantics. Every program has a well founded model where every atomic conclusion is either true, false or *undefined*.

One of such requirements is that of computational complexity. As it is well known, the computation of stable models is NP-hard, whereas that of the well founded model is polynomial. Another requirement is that of being able to answer queries about a given part of the knowledge without the need to, in general, consult the whole knowledge base. The well founded semantics complies with the property of relevance [Dix95b], making it possible to implement query driven proof procedures that, for any given query, only need to explore a part of the knowledge base. Moreover, a stable model approach does not guarantee that a program always has a semantics, partially for the presence of contradictions among rules and partially because even logic programs without contradictions do not always have a stable model. Lack of a semantics is a serious drawback for inferring conclusions, and even more for executing actions, being problematic to execute actions on the basis of a KB without a meaning.

A well founded *paraconsistent* [DP98] semantics would solve these problems by assigning a semantics to any program (even programs with unsolved contradictions). For the reasons mentioned above, we felt the need for a well founded semantics for DyLPs.

A few attempts to define a well founded semantics for DyLPs can be found [ALP+00b, APPP02, Lei97]. However, these approaches lack of a theoretical foundation and a declarative semantics, being mainly a spin-off of the research on stable-models like semantics for DyLPs. The common idea of these approaches is indeed to consider the logic program transformation defined as operational equivalent for stable

models-like semantics of DyLPs and to define the well founded semantics of a DyLP $P_1, \ldots, P_n$ as the well founded semantics of the logic program P resulting from the program transformation above applied to $P_1, \ldots, P_n$. Moreover, these semantics are unappropriate since they inherit the counterintuitive behavior of their stable model-like equivalent. For instance they are not immune to tautologies.

Our contribution, in this chapter, is to provide a declarative paraconsistent well founded semantics for DyLPs. This semantics is a generalization for sequences of programs of the well founded semantics of normal [GRS91] and generalized LPs [DP96]. Moreover it is sound w.r.t. the refined semantics for DyLPs as defined in Chapter 3. As for the refined semantics, the approach herein is also based on the causal rejection principle [EFST02a, Lei03] that we extended from a 2-valued to a 3-valued setting. While the causal rejection principle for 2-valued logics is "a rule is rejected if and only if there is a rule whose head is in conflict with the head of the previous one and whose body is true" its extension to a 3-valued setting becomes: "a rule is rejected if and only if there is a rule whose head is in conflict with the head of the previous one and whose body is *non false* (i.e. true or undefined)". The well founded semantics complies with this principle. Finally, as for the well supported semantics, we provide a sound and complete operational semantics based on a program transformation of a DyLP into a normal LP whose well founded semantics coincides with the well founded semantics of the initial DyLP.

PART II

1.3.3 Chapter 5: Logic programs updates languages and reasoning about actions

After establishing the theoretical foundations of semantics for DyLPs, the next step is how to use this framework for programming evolving and reactive AI applications. As anticipated, there are already some proposals based on DyLPs that contain the core of a KR paradigm capable of self evolution. They are known as *Logic programming update languages* [ABLP02, APPP02, EFST01, Lei03]. These programs integrate usual LP inference rules with *commands* specifying how and when to update the current program with new rules. We briefly describe these languages and focus on Evolp [ABLP02], that we regarded as as the most promising proposal around. Besides the usual LP rules, Evolp has a special kind of rules of the form

$$assert(\tau) \leftarrow B$$

where B is a conjunction of literals (as the body of inference rules) and $assert(\tau)$ is a special atom where τ is a rule. The founding idea of Evolp is that "the rule τ updates the current program if the atom $assert(\tau)$ is true". The evolution of a program is also influenced by external information on the form of LPs called *input programs*. The lan-

guage Evolp has *assert*/1 as its unique command. The original idea of its developers, as expressed in [ABLP02], was to use combinations of rules for defining other commands as *macros* (i.e. specific sub-programs). The simplicity of the syntax and semantics of Evolp is the main feature of this language. Another important feature is that the rule τ can be itself of the form $assert(\tau_2) \leftarrow B_2$ thus allowing a program to update its inference rules (the knowledge) and its command rules (the behavior) at once. Finally the semantics of Evolp is parametric w.r.t the underlying semantics for DyLPs (at least for stable-model like semantics). It costs nothing then to use the refined semantics as the basic update framework for Evolp.

The successive first step in our path, is to use Evolp for *reasoning about the effect of actions* (an argument extensively treated, for instance in [BGP97, GL93, GLL$^+$97, GLL$^+$03]). We develop specific commands (as macros) for specifying the effects of complex commands obtaining the macro language, named EAPs (after *evolving action programs*). The first class of macro commands is that of *dynamic rules* of the form

$$\textbf{effect}(r) \leftarrow C$$

where C is a conjunction of literals with usually at least one atom representing a performed action, while r is a LP rule. The intuitive meaning of the command above is: "the effect of the occurrence of C is r". The other class of macro commands is that of *inertial declarations* of the form

$$\textbf{inertial}(a)$$

where a is an atom. The command above states that the truth value of the atom a (in this setting known as a *fluent*) only changes as (direct or indirect) effect of an action.

The expressivity of EAPs is compared to existing *action description languages*. In particular, we develope translations of the action languages $\mathcal{A}$, [GL98a] $\mathcal{B}$, [GL98a] $\mathcal{C}$ [GLL$^+$03, GL98b] into EAPs, thus showing that EAPs are at least as expressive as the illustrated action languages. Moreover, we show how EAPs are particulary suitable for expressing updates of the action description, allowing to specify changes in the described action domain.

1.3.4 Chapter 6: Evolving Reactive Algebraic Programs

Although Evolp proved to be suitable for reasoning about actions, the situation is different when programming reactive behavior. Evolp does not have a clear formalization of the concepts of events, nor the tools for combining basic events into complex ones. Although the command $assert(\tau)$ is an example of *internal action* (an action producing effects only on the internal state of the considered application) Evolp does not consider *external actions* (actions producing effects on the external environment). Moreover, despite of being developed as a basic language requiring macros for defining complex commands, Evolp itself does not have a clear way for defining macros nor for reusing

macros for defining other macros. To overcome these limitations, we decided to extend other commands (or actions) and constructs thus obtaining the ECA language ERA. Besides inference rules, the basic construct of ERA are reactive rules of the form:

On *Event* **If** *Condition* **Do** *Action*

where *Event* is a special literal representing an event, *Condition* is a conjunction of literals (like the body of inference rules) and *Action* is a special atom representing an action. *Event* can be a *basic event* (an event that is considered somehow as atomic) or a *complex event* obtained from basic events by an algebra of operators (as done, for instance, in [CL03, AC03]). The algebra of ERA is taken by the classical event algebra Snoop [AC03]. For instance, if e_1, e_2, e_3 are events, then $e_1 \triangle e_2$ is an event that occurs iff e_1 and e_2 occur simultaneously and $A(e_1, e_2, e_3)$ is an event that occurs if e_1 occurs, followed by e_2 and e_3 does not occur between the occurrence of e_1 and e_2. It is also possible to provide a name e_{nam} to a complex event e by the *event definition* construct

e_{nam} **is** e

As for events, also actions can be basic ($assert(\tau)$ is an example of basic action) or complex. Complex actions are obtained from basic actions by an algebra of operators. For instance, if a_1 and a_2 are actions, the action $a_1 \triangleright a_2$ is the action obtained by first executing a_1 and then (when the execution of a_1 is complete) the action a_2, while $a_1 \| a_2$ specifies that a_1 and a_2 must be concurrently executed. The algebra of *ERA* is inspired by the work on process algebras [Milb, Mil89, Mila, BW90, Hoa85, SJG96]. As for events, it is possible to provide a name a_{nam} to a complex action a by the *action definition* construct

a_{nam} **is** a

A peculiar construct of ERA are *inhibition rules* i.e. rules of the form

When *B* **Do** *not Action*

where *B* is a conjunction of literals and *Action* is any action. Such rules specify actions that *must not be executed* and, when asserted, can inhibit a previous reactive rule. ECA and inhibition rules, as well as event and action definitions, can be asserted in a program with the $assert/1$ command. The semantic or ERA is given by transforming an ERA program into a DyLP. A reactive rule

On *Event* **If** *Condition* **Do** *Action*

becomes the LP rule

$$Action \leftarrow Condition, Event$$

while the inhibition rule

$$\textbf{When } B \textbf{ Do } not\,Action$$

becomes the rule

$$not\,Action \leftarrow B$$

Thus the update mechanism of DyLPs allows to update the reactive behavior of ERA. The effects of actions is given (as is usually done in process algebras [Milb, Mil89, Mila, BW90, Hoa85, SJG96]) by providing a transition system describing the effect of basic actions and how to combine these effects by the operators of the algebra. The main difference of our approach w.r.t. the process algebras cited above lies in the concurrent execution of actions. For instance, the concurrent execution of two processes a_1 and a_2 in most of the existing process algebras is equivalent to interleaving the execution of the basic actions of either a_1 or a_2 and so on on until both the processes are completely executed. In ERA (as well as in other *truly concurrent* paradigms such as, for instance, the one described in [GM99]) instead, the concurrent execution of two a_1 and a_2 is done by *simultaneously executing* the first basic action of a_1 and a_2 then the second ones and so on. This is so because in ERA it is possible to execute actions *simultaneously* and the simultaneous execution of actions is, in general, not equivalent to any sequentialized execution of the same actions.

We also prove in this chapter how Evolp can be encoded in ERA in a straightforward way, thus unifying the language of EAPs conceived for reasoning about actions with the ECA language ERA conceived for executing actions.

1.3.5 Appendixes

Along the thesis, we present various formal statements in the form of theorems, propositions, lemmas and corollaries. The proofs of these statements are not always interesting by themselves, either because they are too long and technical or because they do not provide especially interesting insights on the treated subject. In these cases we postpone the formal proofs to Appendixes A, (containing the proofs of the statements in Chapter 3), B (containing the proofs of the statements in Chapter 4), and C (containing the proofs of the statements in Chapters 5 and 6).

1.4 Comparisons to other approaches

There is a vast literature on the subjects touched by the thesis, i.e. update of knowledge bases, reasoning about the effect of actions, reactive behavior and Event-Condition-Action languages. Extended comparisons among the logic-programming update framework proposed in the thesis and related approaches are spread among the various chapters of the thesis. Hereafter, we present some of those related approaches and underline the relative advantages of our framework.

We already motivated, in Sections 1.3.1 and 1.3.2, for the need of further research on basic semantics for dynamic logic programs. We refer the reader to Chapters 3 and 4 for detailed comparisons to existing semantics for DyLPs.

The issue of reasoning about the effect of actions is extensively treated and many proposals have been formulated in the literature (see, for instance, [BGP97, GL93, GLL$^+$97, GLL$^+$03, GL98a, GLL$^+$03, GL98b]) allowing to describe the effects of actions on the environment by specifying how facts describing the current state of the environment are affected by the execution of actions. These facts are known as *fluents*. However, in a dynamic environment, not only the facts but also the "rules of the game" can change, in particular *the rules describing the changes*. The capability of describing such kind of *meta level changes* is, in our opinion, an important feature of an action description language. This capability can be seen as an instance of *elaboration tolerance*, i.e. *"the ability to accept changes to a person's or a computer's representation of facts about a subject without having to start all over"* [McC88]. The framework of evolving action programs naturally allows to describe changes in the rules describing an environment by updating such rule by other rules as shown in Chapter 5. This adaptive capability is not shared by other known action description languages.

Reactive behavior have been extensively studied, and many Event-Condition-Action formalisms have been proposed, particularly in relation to active database systems (see, for instance, [Rul, ABM$^+$02, AC03, BPS04, CL03, CT04, GNF98, LML98, SJG04, WC96a]). However, none of these languages has the evolving capabilities of ERA (see Chapter 6 for more details). Moreover, most of the approaches have not a clear declarative semantics, relying instead on an operational semantics or even on implementations not supported by any formal semantics (this is the case, for instance, of most of the active database related approaches like [WC96a, SJG04, Rul, ABM$^+$02]). A formal declarative semantics greatly simplifies the understanding of the framework by the user and the discovery and proof (included automated proofs) of properties, such as termination and correctness or to establish translations from one language to a different one in order to compare their expressivity. For instance, a declarative semantics is among the features demanded in [Sem07] as a basic requirement of a Semantic Web services language.

Since the action description framework of EAPs can be embedded in ERA, it is possible to use ERA both for programming an agent and for reasoning about the effect of the actions of the agent in the environment. This allows, for instance, to write simulations of the agent-environment systems in ERA.

1.5 Application areas

On the level of basic semantics of dynamic logic programs, the most promising application areas are KR domains where knowledge is suitably represented by logic rules and, moreover, not only the facts but also the rules of the domain must be frequently updated.

One of these domains is that of *legal reasoning*, where the laws can be described by logic programming rules. Changes in the laws are represented by updates. Thus, a codex of laws is represented by a DyLP $P_1, \ldots, P_n$ where the changes introduced at some time t are encoded by an update program P_t and the laws in force at time t are represented by the DyLP $P_1, \ldots, P_t$. Thus, the DyLP approach to legal reasoning provides a natural way to query the KB about the laws in force at different time points. Moreover, by switching from linear to multidimensional DyLPs, it is also possible to represent hierarchies of laws from different codex, and to allow laws from a higher hierarchy codex to reject laws from a less important one. For instance, within the corp of laws of an European country, communitarian laws may be allowed to reject national ones. Some application of DyLPs to legal reasoning have been already explored, (see, for instance, [QR99]). The results on semantics of DyLPs developed in the thesis may help to correct counterintuitive behavior of programs representing law systems avoiding undesired results in case of mutual dependencies among literals in the program. Moreover, resorting to the well founded semantics for DyLPs would allow a more efficient implementation.

Regarding the action description framework, the elaboration tolerance feature of EAPs may simplify the writing of action description programs by allowing the user to first realize a simpler version of the program, and then to update its rules instead of rewriting them. For instance, while simulating a physical system, it is common to start by making some simplifying assumptions and elaborate a basic simulation starting from this assumptions. Later on, the initial assumptions may reveal themselves too restrictive and a refined, more general, simulation of the environment is required. Moreover, when simulating a highly dynamic environment, it might be necessary to modify the very rules of the environment to encode changes in the environment itself. For instance, the agent can use a mechanic tool that is initially and correctly assumed to always work correctly. Later on, the prolonged usage makes the tool unreliable, and the possibility that the it may work incorrectly must be taken into account.

As already mentioned, the evolving action program framework can be embedded in the ECA language ERA, thus allowing to implement both an agent and the simulation of the environment where the agent acts in the same framework, greatly simplifying the writing of agent-environment simulations. Many agent applications may gain benefit of this possibility, particularly those where an agent must act in a real physical environment. Moreover, ERA naturally allows both external and internal updates to

the knowledge and behavior of the agent by updating its inference and reactive rules. Hence, the programmer may refine the KB and the behavior of the reactive system (as in case of EAPs) to improve the internal representation of the world. A reactive system written in ERA can also autonomously update its internal representation of the world. This last possibility is particularly useful when the reactive system must act according to a given policy. When the external conditions changes the policy of the system must be accordingly updated (and, possibly, *self* updated).

Possibly the most promising application area for the developed framework is the Semantics Web. In [Sem07], the authors list the desired requirement of a Semantic Web services language. Among these requirements there are, for instance, knowledge representation and automated reasoning capabilities, a declarative semantics, the capability to handle updates and queries of data, the capability to execute and reasoning about, actions, their preconditions and post-conditions (i.e. consequences). All these issues are subjects of the presented work.

1.6 List of original contributions

Hereafter, we provide a list of the principal original contributions of the thesis.

- the definition of the refined extensions principle as a general principle that all the semantics for dynamic logic programs based on the causal rejection principle and extending the stable model semantics for normal logic programs should obey;
- the definition of the refined semantics that complies with the refined extension principle and further investigations on its properties;
- the extension of the well supported model semantics to dynamic logic programs and the proof that it coincides with the refined semantics;
- the extension of the well supported semantics to the case of multidimensional dynamic logic programs and the definition of an alternative, although equivalent, fixpoint characterization of the semantics;
- the definition of a well founded semantics for dynamic logic programs and the study of its properties, particularly the relationship with the refined semantics;
- the definition, on the basis of program transformations, of operational equivalent semantics for both the refined and the well founded semantics;
- the definition the action description language EAPs on the the basis of the logic program updates language Evolp whose semantics is based on DyLPs;
- formal proof of the possibility to modularly translate existing action description languages $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ into EAPs, thus showing that EAPs are at least as expressive as these paradigms;

- discussion of the expressiveness of EAPs related to updates of the action description, and the specification of changes in the described action domain;
- the definition of the Event-Condition-Action language ERA, whose semantics is based on DyLPs, extending Evolp with reactive and inhibition rules, and with algebras of events and actions. The language ERA enriches the logic inference and evolving capabilities of Evolp with more reactive features and the capability to define and executed complex actions.

Chapter 2

Background: Logic programs syntax and semantics

Contents

2.1 Basic concepts in logic programming 18

2.2 Syntax of logic programs 21

2.3 Interpretations and satisfaction 22

2.4 Classes of logic programs and their semantics 24

In this chapter we present a panoramic view of the logic programming paradigm which is the background of the framework developed in the thesis. We present various classes of logic programs, in particular generalized logic programs that are the class of programs which forms the base of this work. We also describe some of the most used semantics of logic programs, in particular the stable models and the well founded semantics, and some other semantics that are relevant in the development of the thesis. The results illustrated in this chapter are established in the LP community. For further details we refer the reader to [AB94, Llo87, Rei87].

2.1 Basic concepts in logic programming

Logic Programming is one of the most popular knowledge representation formalisms. Some of the reasons for its success are a simple logic based syntax and the existence of formal declarative semantics. The formulas of LPs are logic implications (also called rules or clauses) of the form $H \leftarrow B$ where H, called the *head* and the rule is a logic formula (usually a literal) and and B, called the *body* of the rule is a conjunction of literals. A *logic program* is any set of logic programming rules. Classes of logic programs are usually classified depending to the kind of formulas allowed as heads and bodies of clauses. The first class of logic programs ever studied is that of *definite logic programs* (see [EK76]) i.e. sets of rules of the form: $A_0 \leftarrow A_1, \ldots, A_n$ where all the $A_i s$ are atoms. Another important feature of LPs is the existence of a Definite logic programs has a simple, rigorous and universally accepted semantics called *Least Herbrand model* correspond to the two-valued model of the program with less positive conclusions.

Starting from the solid ground of definite programs, one of the most important aspects of research was to study more general classes of logic programs in order to improve the expressivity of the formalism and to provide them with suitable semantics. The first and most known of such classes are *normal logic programs* that introduce negative premises in the body of rules. Within normal LPs, rules have the form

$$A \leftarrow L_1, \ldots L_n.$$

where A is an atom and the various $L_i s$ are positive or *negative* literals. If the conjunction $L_1, \ldots L_n$ is empty we simply write A.

The syntax of logic programs is line with that of the popular programming language *Prolog* [Apt96]. Given an atom A its negation is denoted by $not\, A$. Since, within normal LPs, rules have always a positive head, only positive conclusions can be directly derived by rules. In order to also derive negative conclusion, logic programming assumes a *negation by default* rule which can be explained with the sentence " *a conclusion is false unless there is some evidence that it is true.*" In other words if no rule entails A we assume $not\, A$. The concept of default negation naturally entails some form of *non monotonic reasoning*. Non monotonic reasoning differs from the reasoning of classical propositional logic. Given two theories T_1, T_2 in classical logic, if T_1 entails a formula F (denoted $T_1 \models F$) also the union of the two theories $T_1 \cup T_2$ entails F. This property of propositional and first order logic is known as *monotonicity* and formalizes the intuition that adding more information brings to more conclusions without rejecting conclusions already drawn. Non monotonic formalisms do not obey monotonicity. In then the addition of information may invalidate previous conclusions.

It is immediate to verify that normal logic programs with any form of default negation do not obey monotonicity. For instance, the single rule program $P_1 : \; a \leftarrow not\, b$, entails $not\, b$, since there is no rule that may infer b. Hence, from the unique rule in P_1

we also derive a. Considering also the program $P_2 : \ b.$, the program $P_1 \cup P_2$ clearly implies b and hence it does not imply *not* b. As a consequence there is no reason to conclude b. Hence, using $\vdash$ to denote any form of non monotonic inference mechanism, we have $P_1 \vdash a$ but $P_1 \cup P_2 \nvdash a$ clearly violating monotonicity. For a more detailed explanation of default reasoning and cross connections between logic programming and non monotonic reasoning we refer to [DPP97, Rei87]

The problem of determining the most suitable semantics for normal LPs has involved the logic programming community for years and several solutions were proposed and analyzed. Nowadays there is a stable agreement on the choice of two distinct semantics for normal LPs, namely the *stable model* (firstly defined in [GL88]) and the *well founded* (firstly defined in [GRS91]) semantics. The stable models (also called answer sets) semantics provides a meaning to program by specifying a subset of its classical models called *stable models*. The *stable model conclusions* are those literals belonging to the intersection of all the stable models.

The well founded semantics, instead, always assign one three valued model to a programs, called the well founded model of the program. The *well founded conclusions* are the elements of the well founded model.

The well founded and the stable model are closely related. The well founded model of a program is a subset of any of its stable models [GRS91] and so, the well founded conclusions are a subset of the stable model conclusions. Hence, we say that the well founded semantics is a skeptical approximation of the stable model semantics. Both the well founded and the stable model semantics coincide with the minimal Herbrand model in the special case of definite logic programs (see [EK76]).

Others models of a normal LP that plays a significant role in our exposition are the *supported models* and *minimal supported models*, both subsets of the set of classical models associated to a logic program (see [AB94]). The set of supported models of a program is a super set of the set of its minimal supported models which is in turn a superset of set of its stable models of. In this sense we can consider the well-supported model as a refinement of the other two semantics.

The stable model semantics has particularly simple and intuitive definitions. For this reason, whenever further generalizations of logic programs were studied, it was natural to extend the stable models semantics to the new class of programs as first priority.

As illustrated above, in normal LPs rules have always positive heads. This implies that only positive conclusions can be directly derived by rules. To overcome this limitation, normal logic programs were extended to more general classes of programs allowing negation in the head of clauses. One of these classes are the so called *generalized logic programs* (GLPs), first introduced in [LW92]. Within generalized LPs, (default) negative literals may occur both in the body and in the head of rules. Hence, clauses

of generalized LPs have the form:

$$L_0 \leftarrow L_1 \ldots L_n.$$

where all the $L_i s$ are either positive or negative literals. An extension of the stable model semantics for this class of programs was proposed. A distinct proposal for allowing negative conclusions in the head of rules is that of *extended logic programs* (ELPs) (see [Prz90]). Unlike normal logic programs, and GLPs , extended logic programs have *two forms of negation:* the default negation, and *explicit negation*. Given an atom A, $not\, A$ denotes the default negation of A , $\neg A$ denotes the explicit negation of A and $not\, \neg A$ denotes the default negation of the explicit negation of A. Intuitively, while $not\, A$ means: " there is no reason for believing that A is true", $\neg A$ means "there is evidence that A is false". Finally $not\, \neg A$ means " there is no reason for believing that $\neg A$ is true" From this approach it follows that $\neg A$ is a stronger form of negation than $not\, A$ (for this reason some works refers to explicit negation as *strong negation*). Hence it follows that $\neg A$ entails $not\, A$. On the other side, if A is true, it is clearly not possible to infer $\neg A$, unless the considered program is inconsistent. Hence A entails $not\, \neg A$.

Like normal logic programs, ELPs do not allow default negative conclusions in the head of rules, but they admit explicit negative conclusions. A rule in ELPs has the form.

$$H \leftarrow L_1, \ldots L_n.$$

where H is either an atom A or the negation of atom $\neg A$ and any of the L_is is a literal of any of the forms A, $\neg A$, $not\, A$, $not\, \neg A$. Hence, like generalized programs, also ELPs allow to directly derive negative conclusions. Like GLPs, also ELPs have an extension of the stable model semantics, which is called *answer set semantics*. For further details on ELPs see [Prz90].

Generalized and extended logic programs solve the same problem of expressivity by allowing some form of negation (either default or explicit) in the head of rules. It is proved in [DP96] that the two formalisms has the same expressivity, every GLP can be modularly translated into an ELP and viceversa. The choice among one of the two formalisms is hence more a matter of taste and convenience than a choice with theoretical foundations. In the following we opted for GLPs as preferred formalism for deriving negative conclusions because we found it simpler to understand and generalize a formalism with only one kind of negation w.r.t. a formalism with two forms of negation. Other argumentations may be found for preferring ELPs instead. Within this work we will explore a further degree of non monotonicity in logic programming. Within Dynamic Logic Programs, not only the conclusions but also the rules themselves can be rejected by new incoming rules. This perspective poses new theoretical challenges to the field of logic programming.

2.2 Syntax of logic programs

By an *alphabet* Λ we mean a countable disjoint set of constants, predicate and function symbols with associated arity and an infinite countable set of variable symbols. The sets of constants and predicate symbols are assumed to be non empty. A *term* over Λ is defined recursively in the following way:

- A variable in Λ is a term over Λ.
- A constant in Λ is a term over Λ.
- An expression of the form $f(t_1,\ldots,t_n)$ where f is a function symbol with arity n in Λ and $t_1,\ldots,t_n$ are terms over Λ is a term over Λ.
- Nothing else is a term over Λ.

A *ground term* is a term that does not contain variables. The *Herbrand universe* of Λ is the set of ground terms over Λ. An *atom* over Λ is an expression of the form $p(t_1,\ldots,t_n)$ where p is a predicate symbol of Λ with arity n and $t_1,\ldots,t_n$ are terms over Λ. An atom is *ground* iff all its terms are ground. The *Herbrand base* $\mathcal{H}$ of Λ is the set of ground atoms of Λ. An element of the *propositional language* $\mathcal{L}$ over Λ is either an atom A over Λ or the (default) negation *not* A of an atom over $\mathcal{A}$. We say that A is the *default complement* of *not* A and vice versa. For any literal L, by *not* L we denote its default complement. The elements of $\mathcal{L}$ are called the *literals of* $\mathcal{L}$. Let $L_1,\ldots,L_n$ be a conjunction of literal, by *not* $(L_1 \wedge \ldots \wedge L_n)$ we mean the classical negation of such conjunction i.e. the disjunction *not* $L_1 \vee \ldots \vee$ *not* L_n.

Literals compose the rules which are the fundamental brick of programs. Below we provide the definition of *generalized logic programs* (GLPs). Other classes of logic programs are subclasses of GLPs.

Definition 1 *Generalized logic program*

A (generalized) logic program [1] *is a countable set of rules of the form*

$$L_0 \leftarrow L_1,\ldots,L_n.$$

where each L_i *is a literal.*

The notation $L_1,\ldots,L_n$ stands for an (eventually empty) conjunction of literals. Since the order of the literals is not relevant, we do not distinguish the rule $L \leftarrow B_1$ from the rule $L \leftarrow B_2$ where conjunctions B_1 and B_2 only differ in the order of their literals. If B is

[1] The class of *extended logic programs* is informally presented in the previous section. Other, more general classes of logic programs such as disjunctive logic programs (see [RT88]) are outside the scope of this work.

a conjunction of literals, by an abuse of notation, we also refer to B as the *set* of literals appearing in B.

We assume that the alphabet Λ used to write a program P consists precisely of all the constants predicate and function symbols that explicitly appear in P. By *grounded version of P* we mean the 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. Given a rule τ of the form $L_0 \leftarrow L_1, \ldots, L_n$ by $hd(\tau)$ (read head of τ) we denote the literal L_0 and by $B(\tau)$ (read body of τ) we denote $L_1, \ldots L_n$. If $m = 0$ the rule τ is called a *fact* and the symbol $\leftarrow$ is omitted.

2.3 Interpretations and satisfaction

In this work we will consider both classical forms of reasoning, where conclusions are either true or false, and more general forms of reasoning also admitting that conclusions might be unknown (or undefined) or even contradictory. For this reason, in this work we use a quite general notion of interpretation.

Definition 2

- *An* Interpretation *over a language $\mathcal{L}$ is any subset of $\mathcal{L}$.*
- *A* consistent interpretation *I over $\mathcal{L}$ is any interpretation such that, for each atom A in $\mathcal{L}$,* at most *one of the two literals A, not A belongs to I.*
- *A* complete interpretation *I over $\mathcal{L}$ is any interpretation such that, for each atom A in $\mathcal{L}$,* at least *one of the two literals A, not A belongs to I.*
- *An* inconsistent interpretation *is any interpretation which is* not *consistent.*
- *A* two-valued interpretation *I over $\mathcal{L}$ is any interpretation such that, for each atom A in $\mathcal{L}$,* exactly one *of A or not A belongs to I.*

An interpretation is two valued if and only if it is consistent and complete. When we want to underline that we are dealing with interpretations that may be *not complete we said that we are dealing with* three-valued interpretations.

To simplify the notation, whenever it is clear that we are talking about two-valued interpretations, we omit all its negative literals.

Let $\mathcal{L}$ and $\mathcal{L}'$ be two languages such that $\mathcal{L} \subset \mathcal{L}'$. Let I be an interpretation over $\mathcal{L}'$. We use the notation $I|_{\mathcal{L}}$ for the set of literals of I in $\mathcal{L}$. Given two interpretations I and I^* over $\mathcal{L}'$, we use the notation $I \equiv |_{\mathcal{L}} I^*$ for $I|_{\mathcal{L}} = I^*|_{\mathcal{L}}$.

Definition 3 (Satisfaction) *Given an interpretation I:*

- *A literal L is* true *(resp.* false*) in I iff $L \in I$ (resp. not $L \in I$). If L is true in I we also say that I* satisfies *L (denoted $I \models L$).*

- *A conjunction of literals $L_1,\ldots,L_n$ is true in I iff all the L_i belongs to I. If $L_1,\ldots,L_n$ is true in I we also say that I* satisfies *$L_1,\ldots,L_n$ (denoted $I \models L_1,\ldots,L_n$).*
- *A conjunction of literals $L_1,\ldots,L_n$ is false in I iff* at least *one literal not L_i belongs to I.*
- *A disjunction of literals $L_1,\ldots,L_n$ is false in I iff there exists a literal L_i such that not L_i belongs to I.*
- *I satisfies the rule $\tau : L \leftarrow L_1,\ldots,L_n$. (denoted by $I \models \tau$) iff either I does not satisfy $L_1,\ldots,L_n$ or I satisfies L.*

Note that, unless I is two-valued, it is not true in general that a literal (resp. conjunction of literals) is true in an interpretation iff it is not false. For instance, given the interpretation $I = \{A, not\, A\}$ over the language $\mathcal{L} = \{A, not\, A, B, not\, B\}$,the literals A and *not A* are both false and true in I while B and *not B* are both not true and not false. We use notation $I \not\models L$ (resp. $I \not\models L_1,\ldots,L_n$, or $I \not\models \tau$) for the statement I does not satisfy L (resp. $L_1,\ldots,L_n$, or τ).

The truth value of a literal (resp. conjunction of literals) w.r.t. an interpretation is said to be *a*) false, *b*) true, *c*) undefined or *d*) inconsistent if the literal (resp. conjunction of literals) it is *a*) true but not false, *b*) false but not true, *c*) neither true nor false, or *c*) both true and false in the considered interpretation.

Definition 4 (Model) *A model of a logic program P is any interpretation that satisfies all its rules.*

Given an interpretation I by I^+ we denote the subset of atoms of I and by I^- we denote the subset of default negative literals.

Interpretations are subject to two distinct kinds of ordering. The classical order where interpretations are ordered w.r.t. their atoms and set-inclusion order where interpretations are ordered w.r.t. all their literals.

Definition 5 *Given two interpretations I_1, I_2: We say that I_1 is preferred to I_2 under the* classical order *iff $I_1^+ \subseteq I_2^+$. We said that I_1 is preferred to I_2 under the* set-inclusion order *iff $I_1 \subseteq I_2$.*

The set-inclusion order is non trivial only w.r.t. three-valued interpretations, since distinct two-valued interpretations are always not comparable under this order. The classical order is significative since it privileges interpretations with less positive conclusions.

Given a class of interpretations, I and one of the orders above, an interpretation I in I is said to be minimal under that order iff for any other interpretation I' in I, I is preferred to I'. Let P be a program, the (generally three-valued). When referring to two-valued interpretation, minimality is always considered w.r.t. the classical order.

This notion of minimality w.r.t. the set-inclusion order is important to introduce the concept of *least three-valued model of a program*.

Definition 6 . *Let P, be a logic program. The* least model of *P (denoted $least(P)$) is the minimal interpretation w.r.t. the set-inclusion order which is a model of P.*

2.4 Classes of logic programs and their semantics

The class of logic programs of Definition 1 is that of *Generalized logic programs* (GLPs). This class is the most general class o LPs we will use in this work. Others, previously defined and more restricted classes of programs, are *definite* and *normal* LPs.

Definition 7 (Classes of programs) *Let P be a logic program:*

- *P is* normal logic program *iff the head of any rule in P is an atom.*
- *P is* definite logic program *iff any literal occurring in any rule of P is an atom.*

Given a definite logic Program P its *least Herbrand model* is the least model $least(P)$. Since the head of any rule in P is an atom, $least(P)$ only contains positive literals. It is proved in [EK76] that the least Herbrand model of a definite logic program always exists.

It is possible to characterize the least model of a program as the least (under set-inclusion) fixpoint of the following *immediate consequence operator* (here and in the following an operator is a function over interpretations).

Definition 8 (Immediate consequence operator) *Let P be a definite logic program and I an interpretation. Then:*

$$T_P(I) = \{L | L \leftarrow B \ \in \ P \ \wedge \ B \subseteq I\}$$

The least model of a definite logic program can be found by iterating the immediate consequence operator starting from the empty interpretation. For further details see [EK76].

Theorem 2.4.1 *Let P be a definite logic program, then its minimal model $least(P)$ is the least fixpoint of the immediate consequence operator T_P and it coincides with the ω-iteration of the T_P operator starting from the empty interpretation i.e.*

$$least(P) = T_P \uparrow^{\omega}$$

Given any generalized logic program, if we interpret any negative literal $not\, A$ as a new atom, the result still apply. Hence *any program P has a (generally three-valued and paraconsistent) interpretation $least(P)$.*

Definite programs were the first studied class of logic programs and the least Herbrand model is universally accepted as *the semantics* of this class of programs. Such

a general agreement was much more difficult to achieve in the case of normal logic programs.

As we said in the introduction to this chapter there are two semantics that are commonly accepted as the standard semantics for normal LPs: the *stable model semantics* for two-valued interpretations and the *well founded semantics* for three valued interpretations.

2.4.1 Two-valued semantics for normal logic programs

The stable model semantics, was first presented in [GL88]. A (two-valued) model M is said to be *stable* iff M^+ (i.e. the subset of M consisting of all its positive literals) is equal to the least model obtained from P by deleting all the rules containing false negative literals (w.r.t. M) in their bodies and deleting negative premises from the bodies of the remaining rules.

Definition 9 (Stable Model) *Let P be a normal logic program and I an interpretation. The I-*reduce *of P (denoted $I-P$ or, for simplicity, P^I) is the normal logic program defined as follows:*

- *If a rule*

$$A \leftarrow A_1, \ldots A_i, not\, B_1, \ldots, not\, B_j.$$

belongs to P and I satisfies none of the atoms $B_1, \ldots, B_j$, then the rule

$$A \leftarrow A_1, \ldots A_i$$

belongs to P^I.

- *No other rule belongs to P^I.*

A two-valued interpretation M is a stable model of *P iff*

$$M^+ = \Gamma_P^N(M) = least(P^M)$$

The operator Γ^N is also known as the *Gelfond-Lifschitz operator*.

Note: The equality defining the stable model semantics usually found in literature is

$$M = least(P^M)$$

This depends from the fact that the subset of negative literals in M is usually ignored.

The complexity of the computation of the stable model semantics spans over the class of NP problems as shown by the following Theorem (for proof and further details see [MT91]).

Theorem 2.4.2 *Let P be a finite program with m rules.*

- *Deciding whether P has a stable model is a NP-complete problem w.r.t. m.*
- *Deciding whether an interpretation M is a stable model of P is a linear problem w.r.t m.*
- *Deciding whether a given literal is true in at least one stable model of P is a NP-complete problem w.r.t. m.*
- *Deciding whether a given literal is true in all the stable models of P is a co−NP-complete problem w.r.t. m.*

Other known semantics for normal logic programs are the *supported model semantics* [AB94], and the the *minimal supported model semantics* [AB94].

Definition 10 *Let P be a normal logic program. A two-valued interpretation S of P is a supported iff:*

- *S is a model of P according to Definition 4.*
- *For any atom $A \in S$ there exists a rule $A \leftarrow B$ in P such that $P \models B$.*

An interpretation S is said to be a minimal supported model *of P iff:*

- *S is a supported model of P*
- *There exists no supported model S' such that S' is preferred to S under the classical order.*

Given any program P, the set of all supported models ($SU(P)$), the set of all minimal supported models ($MSU(P)$) and the set of all stable models ($SM(P)$) of P are related by (see [AB94] for proofs)

$$SU(P) \supseteq MSU(P) \supseteq SM(P).$$

Well-supported models

An alternative characterization of stable models is given by resorting to a notion of *level mappings* over the set of atoms of a language $\mathcal{L}$, where a *level mapping* ℓ is a function from $\mathcal{L}^+$ to the set of natural numbers. We also lift ℓ to the negative literals in $\mathcal{L}$ of the form *not* A, by setting $\ell(not\ A) = \ell(A)$. Given a conjunction of literals $C = L_1, \ldots, L_n$ we further extend ℓ by assigning to C the value $\ell(L_i)$, where i is chosen such that the value of $\ell(L_i)$ is maximal, i.e. $\ell(C) = \max(\{\ell(L_i) : L_i \in C\})$. For convenience — and by slight abuse of notation — we assign the value -1 to the empty conjunction of literals. Our approach is stimulated by recent results on uniform characterizations of different semantics for LPs in terms of level mappings as introduced in [HW05] and extended in [Hit03, HS05]. This perspective provides an additional tool and guidelines on how to obtain reasonable new semantics for new classes of programs.

Stable models for normal LPs can be characterized in terms of level mappings, and in this approach they are termed *well-supported models* (WS) [Fag94]. A model is well-supported iff it is possible to define a level mapping over the literals of the language,

such that a literal A belongs to the model iff there is a rule in the program whose head is A, whose body is true in the considered model and the level of A is greater than the level of any atom in the body.

Definition 11 *Let P be a normal logic program over the language $\mathcal{L}$. An interpretation M over $\mathcal{L}$ is a* well-supported model of P *iff i) M is a model of P and ii) there exists a level mapping ℓ defined over $\mathcal{L}$, such that for each atom A in M there exists a rule*

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

with

$$M \models A_1 \ldots, A_n, not\, B_1, \ldots, not\, B_m$$

and $\ell(A) > \ell(Ai)$ for any A_i with $1 \leq i \leq n$

As formalized in the following result of [Fag94], the WS models of a program P coincide with its stable models.

Theorem 2.4.3 *Let P be a normal logic program over the language $\mathcal{L}$. An interpretation M over $\mathcal{L}$ is a* well-supported model of P *iff it is a stable model of P.*

2.4.2 The well founded semantics for normal logic programs

The well founded semantics is three-valued semantics, i.e. a semantics that admits three valued interpretations as models. In the literature, several approaches had been proposed for defining the well founded semantics of normal logic programs and its extensions [AP96, DP96, Gel92, GRS91] One of the most simple is, in our opinion, the one used in [Gel92] defining this semantics in terms of the Γ_P^N operator defined over interpretations used in Definition 9.

In the following a *monotonous operator* Ω is any operator which is monotonous w.r.t. the set-inclusion order, i.e. given two interpretations I, J with $I \subseteq J$ then $\Omega(I) \subseteq \Omega(J)$. Viceversa, Ω is *anti monotonous* iff $\Omega(J) \subseteq \Omega(I)$. The composition of two anti monotonous operators is a monotonous operator.

Given any monotonous operator Ω, we use the notation $lfp(\Omega)$ for the least fixpoint (under set inclusion) of Ω. According to the classical results on the fixpoint theory (see [Tar55] for proofs and further details) any monotonous operator Ω has a least fixpoint W. A *prefixpoint* of Ω is any interpretation F such that $\Omega(F) \subseteq F$.

Lemma 2.4.1 *The least fixpoint of Ω is a* superset *of any prefixpoint of Ω.*

Moreover, there exists a (possibly transfinite) ordinal α such that W is the result of α-iterations of Ω starting from the empty set i.e.

$$F = \Omega \uparrow^{\alpha}$$

Given two ordinal α and β with $\alpha < \beta$ the following inclusion holds

$$\Omega \uparrow^{\alpha} \subseteq \Omega \uparrow^{\beta} \tag{2.1}$$

The Gelfond-Lifschitz operator Γ^N is anti monotonous and the composition of Γ^N with itself (denoted $\Gamma^N\Gamma^N$) is hence monotonous. Hence, it has a least fix point which is the basis of the definition of the well founded model of P. The least fixpoint of the double iteration of an anti monotonous operator Γ is also called the *alternating fixpoint of* Γ.

Definition 12 *Let P be a normal logic program, let Γ_P^N be the operator mapping an interpretation I to the interpretation $\Gamma_P^N(I) = least(P^I)$ and let $\Gamma^N\Gamma_P^N$ be the operator obtained by the composition of Γ_P^N with itself. Then W is the* well founded model of *P iff:*

- $W^+ = lfp(\Gamma^N\Gamma_P^N)$.
- $W^- = \{not\, A : A \notin \Gamma_P^N(W^+)\}$.

The set of well founded atomic conclusions W^+ given a program P of is the alternating fixpoint of Γ_P^N.

The well founded model always exists and it is a subset of any stable model (see [Gel92] for a proof) .

Theorem 2.4.4 *Any normal logic program P has a well founded model W. For any stable model M of P the inclusion $W \subseteq P$ is true. If W is two-valued, then it coincides with the unique stable model of P.*

Given any subset K of the Herbrand base $\mathcal{H}$, let the set *not* K denote the set of negative literals *not* A such that A belongs to K. The well founded model of any program satisfies the following equality (see [Gel92] for a proof):

$$W = W^+ \cup not\,(\mathcal{H} \setminus \Gamma_P^N(W^+)) \tag{2.2}$$

Moreover, the inclusion

$$W^+ \subseteq \Gamma_P^N(W^+)$$

always holds. While the set W^+ is the set of true atomic conclusions in W, by equality 2.2 it follows that $\Gamma_P^N(W^+)$ is the set of *non-false* atomic conclusions in W and hence $\Gamma_P^N(W^+) \setminus W^+$ is the set of *undefined* atomic conclusions according to W.

Given any fixpoint F of $\Gamma^N\Gamma_P^N$ satisfying the inclusion $F \subseteq \Gamma_P^N(F)$, the interpretation I such that

$$I = F \cup not\,(\mathcal{H} \setminus \Gamma_P^N(F))$$

is said to be a *partial stable model of P*. The well founded model W is (alternative definition) *the least partial stable model of P*.

From the results on monotonous operators illustrated above, it also follows that the well founded mode can be computed by iterating the $\Gamma^N\Gamma^N_P$ operator starting from the empty interpretation. Moreover, for any program P there exists a least ordinal α such that W^+ is equal to the α-iteration of $\Gamma^N\Gamma^N_P$ i.e.

$$W^+ = \Gamma^N\Gamma^N_P \uparrow^\alpha$$

Moreover, for any ordinal $\beta < \alpha$ the following inclusions hold:

$$B = \Gamma^N\Gamma^N_P \uparrow^\alpha \subset W \qquad \Gamma^N_P(W^+) \subseteq \Gamma^N_P(B)$$

Hence, every iteration of Γ^N_P approximates W, even iterations (if we limit to finite ordinals) approximate W^+ from the bottom, while odd iterations approximate $\Gamma^N_P(W^+)$ (set of non false atomic conclusions according to W^+) from the top.

2.4.3 Stable model semantics for generalized logic programs

The stable model semantics was extended in [LW92] to generalized logic programs (see Definition 1). Since every normal logic program is also a GLP, the definition of the semantics provided here is a an alternative although equivalent characterization of the semantics of Definition 9 . Basically the idea is that a two-valued interpretation M is a stable model of a program P iff it is the least model of the program obtained by adding the negative literals in M to P.

Definition 13 (Stable Model of a GLP) *Let P be a GLP and M a two-valued interpretation. We say M is a* stable model *of P iff:*

$$M = least(P \cup M^-)$$

2.4.4 The well founded semantics of generalized and explicit logic programs

In [AP96] and extension of the well founded semantics is defined characterizing the atoms of the well founded model of an extended logic program P in terms of the least fixpoint of a monotonous operator $\Gamma^N_P\Gamma^S_P$ obtained by combining the usual Gelfond-Lifschitz operator Γ^N_P with another anti monotonous operator Γ^S_P which is the Gelfond-Lifschitz operator of a logic program P_S obtained from P. Based on this semantics, in [DP96] a well founded semantics for generalized logic programs is defined by transforming a GLP into an extended logic program. We refer to this semantics as to the *transformational well founded semantics for GLPs*.

We briefly summarize the definitions of the two semantics.

Definition 14 *Given any extended logic program P, the semi normal form of P, denoted P_S is defined as follows:*

- *For any clause*

$$A \leftarrow body$$

in P the clause

$$A \leftarrow body,\ not\ \neg A$$

is in P_S.

- *For any clause*

$$\neg A \leftarrow body$$

in P the clause

$$A \leftarrow body,\ notA$$

is in P_S.

- *There are no others clauses in P_S.*

Definition 15 *Let P be any extended logic program and X a set of atoms of the language of P. We define the operators Γ_P^E and Γ_P^{ES} in the following way.*

$$\Gamma_P^E = least(P^X)$$

$$\Gamma_P^{ES}(X) = least(P_S^X)$$

We use Γ^Einstead of Γ_P^E and Γ^{ES}instead of Γ_P^{ES} whenever the program P is clear from the context.

Definition 16 *Let P be an extended logic program and F any fixpoint of $\Gamma^E\Gamma^{ES}$, then*

$$F \cup not\ (\mathcal{H}_P \setminus \Gamma^{ES}(F))$$

is a paraconsistent partial stable model of P

We define the well founded model of P as the least partial stable model of P under set inclusion order.

Definition 17 *Let P be any generalized logic program. For the transformed of P we mean the extended logic program P^{not} where:*

$$\begin{array}{lll} P^{not} & = R1 \cup R2 \cup\ R3 & \\ R1 & = A_p \leftarrow body. & for\ each\ rule\ A \leftarrow body\ in\ P \\ R2 & = A_n \leftarrow body. & for\ each\ rule\ not\ A \leftarrow body\ in\ P \\ R1 & = A \leftarrow A_p.\quad \neg A \leftarrow A_n. & for\ each\ atom\ A\ in\ \mathcal{H}_P \end{array}$$

We will use the notation P^{Snot} for the semi normal for of P^{not}.

Definition 18 *Let P be any generalized logic program in the extended language $\mathcal{L}$ and P^{not} its transformed. Let $W1$ be the well founded model of P^{not}. We define the well founded model of P under the transformational semantics as $W1|\mathcal{L}$.*

Any normal LP is also a generalized and extended logic programs and hence Definitions 16 and 18 hold for normal logic programs as well. It is important to notice that the two semantics above extend the well founded semantics of normal logic programs as formally state by the following results from, respectively, [AP96] and [DP96].

Theorem 2.4.5 *Let P be a normal logic program and let W be the interpretation model computed according to Definition 16. Then W coincides with the well founded model of P according to Definition 12*

Theorem 2.4.6 *Let P be a normal logic program and let W be the interpretation model computed according to Definition 18. Then W coincides with the well founded model of P according to Definition 12*

The transformational semantics for GLPs defined in [DP96] satisfies several desirable properties and, in our opinion, it behaves intuitively. Despite of this, the approach used in the definition is based on a syntactical transformation and it introduces new literals, hence it is nearer to an operational approach then to a declarative one. Currently, to the best of our knowledge, there exists no purely declarative definition of the well founded semantics of GLPs.

Part I

On the Semantics of Dynamic Logic Programs

Chapter 3

A refined, well-supported semantics for logic programs updates

Contents

3.1 Introduction and motivation 37
3.2 Preliminaries and previous works on updates 48
3.3 Refined extensions 52
3.4 Refined semantics for dynamic logic programs 56
3.5 Relationship with other semantics for dynamic logic programs . . . 60
3.6 Well-supported models for DyLPs 62
3.7 Further properties of the refined semantics 69
3.8 Refined well-supported semantics for multidimensional dynamic logic programs 73
3.9 Fixpoint characterization for well-supported models of MDyLPs . . 74
3.10 Relationship to other semantics for MDyLPs 78
3.11 A program transformation for the refined semantics of DyLPs 81
3.12 Transformational semantics for MDyLPs 86
3.13 Concluding remarks 88

This chapter introduces the paradigm of dynamic logic programs which is the common denominator of the whole thesis. We open the chapter by illustrating accomplished results in the field, in particular the existing semantics based on the stable model approach and the fundamental causal rejection principle, and by presenting open issues.

In particular we underline counterintuitive behavior of the existing semantics, especially in presence of tautologies and self-dependencies among literals. To better understand these behavior we enounce the refined extension principle and show how existing semantics do not obey to this principle. Then we define the refined semantics for dynamic logic programs as an extension of the stable model semantics for generalized logic programs and prove that this semantics satisfies the refined extension principle. As expected, the known counterintuitive behavior of the existing semantics disappear in the refined semantics. We proceed our investigation on what is the most suitable semantics for dynamic logic programs by presenting an extension of the well-supported semantics to the dynamic case and show that it is an alternative characterization of the refined semantics. Then we extend the refined well-supported semantics to the more general class of multidimensional dynamic logic programs and provide a fixpoint characterization analogue to the classical definition of the stable model semantics. The last part of the chapter establishes an operational equivalent of the refined semantics in the case of linear and multidimensional dynamic logic programs. Part of the results of this chapter have been published in [ABBL04, ABBL05, FAA03, BAB05].

3.1 Introduction and motivation

Until recently, most of the research in the field of logic programming for representing knowledge that evolves with time has focused on changes in the extensional part of knowledge bases (factual events or observations). This is what happens with the event calculus [KS86], logic programming forms of the situation calculus [LPR98, MH69] and logic programming representations of action languages [GL93]. In all of these, the problem of updating the intensional part of the knowledge base (rules or action descriptions) remains basically unexplored. In recent years considerable effort has been devoted to explore the problem of how to update knowledge bases represented by logic programs (LPs) with new rules. This allows, for instance, to better use LPs for representing and reasoning with knowledge that evolves in time. This issue is usually known as *logic programs updates*.

In this perspective, the object of study is not any longer a single logic program but rather an arbitrary finite *sequence* of logic programs $P_1, \ldots, P_n$ called *dynamic logic programs* (DyLPs), each program in the sequence representing a supervenient state of the world. The different states can be seen as representing different time points, in which case P_1 is an initial knowledge base, and the other $P_i s$ are subsequent updates of the knowledge base. The different states can also be seen as knowledge coming from different sources that are (totally) ordered according to some precedence, or as different hierarchical instances where the subsequent programs represent more specific information. Different semantics have been proposed [ALP$^+$00a, BFL99, EFST02a, Lei03, LP97, SI99, ZF98, Sef00] that assign meaning to DyLPs. The role of these semantics of DyLPs is to employ the mutual relationships among different states to precisely determine the meaning of the combined program comprised of all individual programs at each state. The various programs in the sequence belong to some class of logic programs (usually generalized or extended LPs) allowing negative conclusion in the head of rules. This choice allows the possibility (and the problem) that a newer rule may contradict an older one by negating its conclusions. The choice of generalized or extended logic programs as basic class of programs does not generate a substantial difference. As shown in [Lei03], it is possible to adapt those semantics of dynamic logic programs based on ELPs to dynamic logic programs based on GLPs. In the following we will consider DyLPs based on generalized logic programs.

Intuitively, one can add, at the end of the sequence, newer rules or rules with precedence (arising from newly acquired, more specific or preferred knowledge) leaving to the semantics the task of ensuring that these added rules are in force, and that previous or less specific rules are still valid (by inertia) only as far as possible, i.e. that they are kept as long as they are not rejected. A rule is rejected whenever it is in conflict with a newly added one whose body is true in the considered model. This concept of rejection

was first introduced by by Leite and Pereira in [LP97] and then utterly fixed by Eiter et al. in [EFST02a] that gave it the name of *causal rejection principle*.

Most of the semantics proposed in the past years (namely those appearing in [ALP$^+$00a, BFL99, EFST02b, Lei03, LP97]) accomplished with this principle. We provide here a simple and intuitive example of DyLPs involving causal rejection of rules.

Example 3.1.1 *The initial program*

$$P_1: \quad wet \leftarrow rain.$$
$$rain.$$

Simply states that if it rains then it is wet and that currently it rains. We update this program with the following:

$$P_2: \quad not\, rain \leftarrow sun.$$
$$sun.$$

Stating that it does not rain if there is the sun and indeed there is the sun. According to the known semantics for updates based on causal rejection, the (stable) model associated to P_1 is $\{wet, rain\}$ stating that it rains and hence is it is wet. However, if we consider the DyLP P_1, P_2 we have a rule in P_2 which is in conflict with the rule (in this case it is simply a fact) rain and whose body is true and hence the fact rain is rejected. According to this idea, the stable model of the DyLP above is

$$\{sun, not\, rain, not\, wet\}$$

Indeed, all the semantics based on causal rejection agree in this example.

The causal rejection principle is one of the milestones of this work, although we believe that, in its present form it could lead the reader to wrong conclusions that a DyLP is, in general, equivalent to the union of all the programs in the sequence minus a set of rejected rules. Although this could be the case in some examples, this is generally false. The reason is that, according to the causal rejection principle, the rejected rules are determined *for each model* hence a rule rejected according to a model could be preserved according to another one. As a consequence, a DyLP is in general not equivalent (i.e. does not have the same semantics) to a subset of the union of its rules. We provide an intuitive example of this situation.

Example 3.1.2 *Initially a restaurant is open every day. Later on, the owner decides not to open it during holidays. A day can be either a working day or a holiday. As a consequence, a restaurant is open during the working days and not open during the holidays. The example is formalized by the following DyLP.*

$$P_1: \quad open(restaurant).$$
$$P_2: \quad not\, open(restaurant) \leftarrow holiday.$$
$$workday \leftarrow not\, holiday.$$
$$holiday \leftarrow not\, workday.$$

According to all the semantics for DyLPs based on causal rejection, the program: $\mathcal{P} = P_1, P_2$ has two (stable) models, namely:

$$M_1 = \{open(restaurant), workday\} \quad M_2 = \{holyday\}$$

As the reader may notice, there is a potential conflict between the single rule of P_1 and the first rule of P_2. If the predicate holiday is false in the selected model (M_1), the body of the rule not open(restaurant) ← holiday is false and hence the rule open(restaurant) is not rejected. Otherwise, if holiday is true in the selected model (M_2), the body of the rule not open(restaurant) ← holiday is satisfied and hence the rule open(restaurant) is rejected. There exists no subset of the program $P_1 \cup P_2$ with the same semantics. Indeed, the stable models of any subset containing the single rule of P_1 satisfy the predicate open(restaurant), while the stable models of any subset not containing the single rule of P_1 do not satisfy the predicate open(restaurant).

Although, for historical reasons, we maintain the expression causal rejection, the reader could find it more intuitive to think to the principle in it terms of *causal exception* which could be expressed in this way: a rule is fulfilled unless it is in conflict with a newly added one whose body is true in the considered model. Basically the new rules can specify exceptions to the old ones.

In this part of the work we focus our study on semantics generalizing the stable models semantics and that are based on causal rejection of rules. Semantics not based on causal rejection like the ones defined in [SI99, ZF98], or semantics which use more general forms of rejection like the one presented in [Sef00] and their relation with semantics based on causal rejection are subject of study of others works like [Lei03] and are only mentioned in Section 4.8.

3.1.1 Open issues on semantics for DyLPs bases on causal rejections

While most of the existing semantics for DyLPs based on causal rejection coincide on a large class of program updates (Cf. [EFST02b, Hom04]), there are situations in which the set of (dynamic) stable models (SMs) differs from one semantics to the other. Usually such counter-examples show a counterintuitive behavior of the semantics when dealing with particular kinds of recursive dependencies.

The first known and studied examples [EFST02a, Lei03] of different behavior w.r.t. some apparently inoffensive updates involve tautologies[1], as it is illustrated by the following example.

Example 3.1.3 *Let us consider the DyLP P_1, P_2 with:*

$$\begin{aligned} P_1 &= a. \\ P_2 &= not\ a \leftarrow not\ a. \end{aligned}$$

[1] By a tautology we mean a rule of the form $L \leftarrow B$ with $L \in B$.

Intuitively, one would expect the update of P_1 with P_2 not to change the semantics because the only rule of P_2 is a tautology. Hence, the unique model of the program P_1, P_2 should be $\{a\}$ (which is the unique stable model of P_1). This is not the case according to the semantics of justified updates *[LP97] which admits, for the DyLP P_1, P_2, the models $\{a\}$ and $\{\}$[2]. This example is formally discussed in [Lei03].*

The *dynamic stable model semantics* [ALP$^+$98b, ALP$^+$00b] was introduced for properly dealing with this kind of programs and, indeed, it provides the expected result on example 3.1.3.

Unfortunately, there still remain examples involving tautological updates where none of the existing semantics behaves as expected. Let us now show an example to illustrate the problem.

Example 3.1.4 *Let us consider the program P_1 describing some knowledge about the sky. At each moment it is either day time or night time. We can see the stars whenever it is night time and there are no clouds; and currently it is not possible to see the stars.*

$$\begin{aligned} P_1: \quad & day \leftarrow not\ night. \\ & night \leftarrow not\ day. \\ & stars \leftarrow night, not\ cloudy. \\ & not\ stars. \end{aligned}$$

The only dynamic stable model of this program is $\{day\}$. Let us suppose now the program is updated with the following tautology:

$$P_2: \quad stars \leftarrow stars.$$

This tautological update introduces a new dynamic stable model. Indeed, besides the intuitively correct dynamic stable model $\{day\}$, the dynamic stable model semantics also admits the model $\{night, stars\}$. Technically, this happens because the rule in P_2, which has a true body in the latter model, causally rejects the fact not stars of P_1. Furthermore, these results are shared by all other existing semantics for updates based on causal rejection [ALP$^+$98b, BFL99, EFST02a, Lei03, LP97, LP98]. We argue that this behavior is counterintuitive as the addition of the tautology in P_2 should not add new models.

Typically, these tautological updates are just particular instances of more general updates that should be ineffective but, in reality, cause the introduction of new models e.g. those with a rule whose head is *self-dependent* [3] as in the following example.

Example 3.1.5 *Let us consider again program P_1 of Example 3.1.4, and replace P_2 with*

[2] Similar behavior is exhibited by the update answer-set semantics [EFST02a].

[3] A literal L_1 depends from a literal L_2 iff there exists a rule τ whose head is L_1, and either L_2 occurs in the body of τ, or there exists a literal L_3 occurring in the body of τ such that L_3 depends from L_2. A literal is self-dependent iff it depends from itself.

$$P_2: \quad stars \leftarrow venus.$$
$$venus \leftarrow stars.$$

While P_1 has only one model (viz., $\{day\}$), according to all the existing semantics for updates based on causal rejection, the update P_2 adds a second model,

$$\{night, stars, venus\}$$

Intuitively, this new model arises since the update P_2 causally rejects the rule of P_1 which stated that it was not possible to see the stars.

The open issues at the time the present work started were hence the following:

- There were several semantics on DyLPs based on a stable model-like approach and the causal rejection principle that. Although coinciding on large classes of programs, the various semantics differ on examples involving self dependencies among literals. There were no general agreement on what should be *the* stable model semantics for DyLPs based on causal rejection.
- There were examples showing counterintuitive behavior of all the existing semantics
- In particular, none of the existing semantics was immune from tautologies.

Our research focused on finding a common answer to these issues.

3.1.2 A principled based approach

At the time we started our research, there were some specific examples showing that none of the existing semantics was completely satisfactory, but there was not a general idea on what was not working in the existing semantics. Basically the questions were: "what is not working exactly?" "Which *principle* is violated by the counterintuitive examples?". Somehow the answer to these questions could also suggest which was the right semantics for DyLPs. The best answer available at the time was immunity from tautologies (see [Lei03]), i.e. the addition or deletion of tautologies from a DyLP should not modify its semantics. Nevertheless this property alone was less than adequate for at least two reasons.

- The counterintuitive behavior in the existing semantics span beyond the mere examples on tautologies.
- In principle it would have been simple to define a semantics immune from tautologies by syntactically removing all the tautologies from a DyLP and then applying any of the existing semantics based on causal rejection. This would not

have provided any hint on what should have been the proper semantics for DyLPs.

On the basis of these considerations, it is our stance that, besides the principles used to analyze and the semantics for DyLPs, (described in [EFST02a, Lei03] and in the continue of the paper), another important principle was needed to test the adequacy of semantics of logic program updates.

Examples 3.1.3, 3.1.4 and 3.1.5 show that the known counterintuitive behavior were related to the unwanted generation of new dynamic stable models when certain sets of rules are added to a dynamic logic program. It is worth noting, however, that an update with the form of P_2 in Example 3.1.5 may have the effect of eliminating previously existing models, this often being a desired effect, as illustrated by the following example.

Example 3.1.6 *Let us consider program P_1 with the obvious intuitive reading: one is either alone or with friends, and one is either happy or depressed.*

$$\begin{array}{rl} P_1: & friends \leftarrow not\ alone. \\ & alone \leftarrow not\ friends. \\ & happy \leftarrow not\ depressed. \\ & depressed \leftarrow not\ happy. \end{array}$$

This program has four dynamic stable models, namely:

$$\{friends, depressed\} \quad \{friends, happy\} \quad \{alone, happy\} \quad \{alone, depressed\}$$

Let us suppose now that the program is updated with the following program (similar to P_2 used in Example 3.1.5):

$$\begin{array}{rl} P_2: & depressed \leftarrow alone. \\ & alone \leftarrow depressed. \end{array}$$

This update specified by P_2 eliminates two of the dynamic stable models, leaving only $\{friends, happy\}$ and $\{alone, depressed\}$, this being a desirable effect.

Our intuition was that the evidenced counterintuitive behavior originates by the violation of some principle guaranteeing that, under some conditions, the addition of rules in a program does not introduce new models in the semantics (a sort of criterium of weak monotonicity).

We started our investigation with the simple case of a single logic program and set forth the *refined extension principle* which, if complied with by a semantics, specifies conditions under which rules can be safely added without introducing new models according to that semantics. Notably, the stable model semantics [GL88] complies with this principle. Informally, the semantics based on stable models can be obtained by taking the least model of the definite program obtained by adding some assumptions

(default negations) to the initial program. Intuitively, the refined extension principle states that the addition of rules that do not change that least model should not lead to obtaining more (stable) models.

Subsequently, we generalize this principle by lifting it to the case of semantics for dynamic logic programs. This si possible since all the existing semantics for DyLPs are characterizable by specifying a set of *default assumptions* as in the case of single logic programs and a set of *rejected rules* whose choice is consistent with the causal rejection principle.

Not unexpectedly, given the examples above, it turns out that none of the existing semantics for updates based on causal rejection complies with such principle.

This discrepancy demanded the introduction of a new semantics for dynamic logic programs, namely the *refined dynamic stable model semantics* (or simply refined semantics) which complies with the refined extension principle. The *refined semantics* is obtained by refining the dynamic stable model semantics [ALP$^+$98b, ALP$^+$00b, Lei03] which, of all the existing semantics, as shall be seen, is the one that complies with the refined extension principle on the largest class of programs. This semantics is proved to comply with the refined extension principle and to be immune to the addition and deletion of tautologies.

3.1.3 Beyond linear dynamic logic programs and the refined extension principle

Multidimensional dynamic logic programs (MDyLPs) [BFL99, LAP01] generalize DyLPs by considering, instead of sequences, finite *partially ordered* multisets of programs. This generalization allows to combine in a single framework the possibility of having hierarchically ordered knowledge bases, with evolution in time as shown in the next example.

Example 3.1.7 *We present an example of application of MDyLPs to legal reasoning taken from the European legislation on food and dishes.*

The European laws determining if a dish can be sold to the public overwrite the national laws. The Italian law establishes that the selling of a dish is allowed if its ingredients are allowed. All the ingredients in the list pms of ingredients of a pizza margherita *(from now on pm) are allowed. This is codified by the program*

$$P_I \quad sell(D) \leftarrow ingredients(D,Ds), allowed(Ds). \quad ingredients(pm,pms). \quad allowed(pms).$$

The very same law also holds for Portuguese dishes. All the ingredients lis of linguado *(from now on li) are allowed as it results from the following program*

$$P_P \quad sell(D) \leftarrow ingredients(D,Ds), allowed(Ds). \quad ingredients(li,lis). \quad allowed(lis).$$

Later on, in both countries, the concept of traditional dish *is introduced. Both pizza margherita and linguado are classified as traditional dishes. This is codified by updating the programs P_I and P_P , respectively, with the updates*

$$P_{I1} \quad traditional(pm). \qquad P_{P1} \quad traditional(li).$$

We consider now the European normative. According to these laws, it is not possible to sell a dish if it is not cooked according to some standards. This is encoded by the single-rule program

$$P_E \quad not\, sell(D) \leftarrow not\, standard\_cooked(D).$$

The linguado is cooked in a standard way, but the traditional pizza margherita is cooked in a wood-oven which does not satisfies the European standards, hence $standard\_cooked(li)$ is satisfied but $standard\_cooked(pm)$ does not and, as a consequence $sell(li)$ holds but $sell(pm)$ is not satisfied. Later on, as a consequence of the complaints of various countries, the European normative is updated to allow traditional eishes to overcome this restrictive rules. This is encoded by the update

$$P_{E1} \quad sell(D) \leftarrow traditional(D), ingredients(D,Ds), allowed(Ds).$$

As a consequence of this last update $sell(pm)$ is again satisfied.

Note in this example how the partially ordered programs are used to represent precedence among rules coming from different sources, as well as to represent updates of rules.

Another, equivalent, way to define MDyLPs is that of a finite a acyclic diagraph with LPs associated to each node such that the program P_α is associated to the node α. The two representations are equivalent. Given an acyclic diagraph and the associated programs, a partial order $\prec$ is induced on the programs in the following way: for any two programs P_α, P_β associated to the distinct nodes α and β, $P_\alpha \prec P_\beta$ holds iff there exists a directed path from α to β. On the other side, given a MDyLP with a partial order $\prec$, the corresponding acyclic diagraph is defined as it follows: the nodes of the graph are the indexes of the various programs. For any two nodes α, β there exists a directed link from α to β iff $P_\alpha \prec P_\beta$ and there exists no program P_γ such that

$$P_\alpha \prec P_\gamma \prec P_\beta$$

The update answer-set semantics [EFST02a] and the dynamic stable model semantics [ALP$^+$98a] have extensions to the multidimensional case (see [BFL99] and [Lei03]). Since MDyLPs extend the class of (linear) dynamic logic programs, the examples

3.1.3, 3.1.4 and 3.1.5 of counterintuitive behavior involving self dependencies among literals also apply to these semantics. As for DyLPs, the dynamic stable model semantics is the one showing less cases of counter intuitive behavior. Nevertheless, even for this semantics, there are more general classes of examples of such counterintuitive behavior than in the case of *DyLPs*, as we see from the following example.

Example 3.1.8 *Some sensors are located on a system of pipes. The sensor register whether there is water in a specific pipe or not. Two distinct sensors are placed in the pipe number* 5*. The first sensor detects water, while the second detects no water. Moreover we now the pipes from* 1 *to* 10 *are interconnected in a circular way (piper* 1 *is connected with piper* 2 *and so on, while pipe* 10 *is connected with pipe* 1*) and they are all at the see level. Hence, if there is water in one of the pipes there is water in all of them. We represent this situation with the following program*

$$\begin{array}{ll} P_1: & water(p(5)). \\ P_2: & not\, water(p(5)). \\ P_3: & water(p(n+1)) \leftarrow water(p(n)). \quad 1 \leq n \leq 9 \\ & water(p(1)) \leftarrow water(p(10)). \end{array}$$

We assign the same degree of reliability to the information coming from the two sensors, while the rules expressing the interconnection of the pipes are clearly more trustable than the information coming from sensors which could even be broken. Hence we define the following order among the programs.

$$P_1 \prec P_3 \quad P_2 \prec P_3$$

and thus we obtain the multidimensional dynamic logic program $\mathcal{MP}$. According to the semantics defined in [LAP01, BFL99], the program $\mathcal{MP}$ has the model

$$\{water(p(1)) \ldots water(p(10))\}$$

This does not seem a good conclusion, since the information coming from the two sensors is clearly contradictory and there is no way to decide which source of information should be trusted. We would hence expect that the semantics would have signaled the inconsistency by admitting no model. What happened is that the cycle in P_3 removed the contradiction of P_1 and P_3 by rejecting $not\, water(p(5))$.

In general, when we merge information from several sources, without assigning a total order among such sources, if potential contradictions are present these semantics may admit undesired models.

Unfortunately, as exposed in the previous section, it is not possible to directly extend the definition of the refined semantics to the multidimensional case. An innovative approach is needed.

The refined extension principle is enough for classifying the evidence class of counterintuitive behavior of the existing semantics for DyLPs and to prove that the newly

introduced refined semantics is more suitable than the others in the sense that it complies with the principle. However, the principle is too weak for uniquely determining one "right" semantics for DyLPs. For example, the trivial semantics that assigns to each DyLP the empty set of models, satisfies the principle, which is obviously unsatisfactory. Stronger criteria are thus needed for this purpose. Moreover, the principle is valid only for those semantics that can be univocally determined in terms of sets of rejected rules and default assumptions. This last condition is too restrictive if we extend our investigation to more general classes of Multidimensional dynamic logic programs (MDyLPs). For these reasons we fell the need to introduce new theoretical notions for characterizing semantics of DyLPs.

We found an answer to these issues by extending the notion of *well-supported model* (WS model) [AB94, Fag94] to DyLPs and MDyLPs. Fages [Fag94] shows the equivalence between the concept of stable model and WS model, i.e. given a program P, an interpretation M is a stable model of P iff it is a WS model of P. We show that a similar result holds for the refined semantics of DyLPs. For this reason we refer to this semantics also as the *well-supported semantics for DyLPs*.

As expected, none of the existing semantics for MDyLPs coincides with the well-supported model semantics. Well-supported models do already provide a semantics for MDyLPs. Such a descriptive characterization, however, is not completely satisfactory for several reasons, the main one being the problem of finding a reasonable algorithm for the computation of such semantics. So we provide an alternative, though equivalent and more traditional characterization based on a fixpoint operator. Since the defined semantics coincides with the refined semantics in the case of DyLPs and extends such semantics to MDyLPs, we also refer to it as *the refined semantics for MDyLPs*. We then establish relationships between the refined semantics and the existing semantics for MDyLPs, and show that any WS model is also a model in the existing semantics.

The last part of the this chapter is oriented to implementations and establishes an operational semantics that is proved to be equivalent to the refined semantics for DyLPs and MDyLPs. For providing an operational definition for extensions of normal LPs, a widely used technique is that of having a transformation of the original program into a normal logic program and then to prove equivalence results between the two semantics. In logic programs updates this methodology has been successfully used several times (see, for instance, [ALP$^+$00a, EFST02a]). Once such program transformations have been established (and implemented), it is then an easy job to implement the corresponding semantics by applying existing software for computing the semantics of normal LPs, like DLV [DLV00] or smodels [SMO00]. Following this direction, we provide transformations of DyLPs and MDyLPs into normal LPs and provide equivalence results.

The shape of the transformations proposed for the refined semantics for DyLPs is quite different from the ones proposed for the other semantics (see for instance

[ALP+00a, APPP02, Lei97]). These differences are partially related to the different behavior of the considered semantics (none of the existing program transformation is sound and complete w.r.t. the refined semantics) but they are also related to peculiar properties of the presented program transformations. One of such properties is the minimum size of the transformed program. Since the size of the transformed program significantly influences the cost of computing the semantics (especially in case of the stable model semantics), this topic is quite relevant. A drawback of the existing program transformations is that the size of the transformed program linearly depends on the size of the language times the number of performed updates. This means that, when the number of updates grows, the size of the transformed program grows in a way that linearly depends on the size of the language. This happens even when the updates are empty. On the contrary, in our approach the size of the transformed programs has an upper bound that does not depend on the number of updates, but solely (and linearly) on the number of rules and the size of the original language (see Theorem 3.11.2).

A prototypical implementation for the refined semantics in the linear case that uses the theoretical background of this chapter is available at[Ban05a] This implementation take advantage of DLV and smodels systems to compute the semantics of the transformed programs.

Due to its simplicity, the proposed transformation for the linear case is interesting beyond the mere scope of implementation since it gives new insights on how the rejection mechanism works and how it creates new dependencies among rules. The transformed programs provide an alternative description of the behavior of the updated program.

We also proposed a transformational semantics for MDyLPs. Since the refined semantics for DyLPs is a special case of the refined semantics for MDyLPs, the proposed transformation is also sound and complete w.r.t the refined semantics for DyLPs, although more complicated than the specific transformation presented for the linear case.

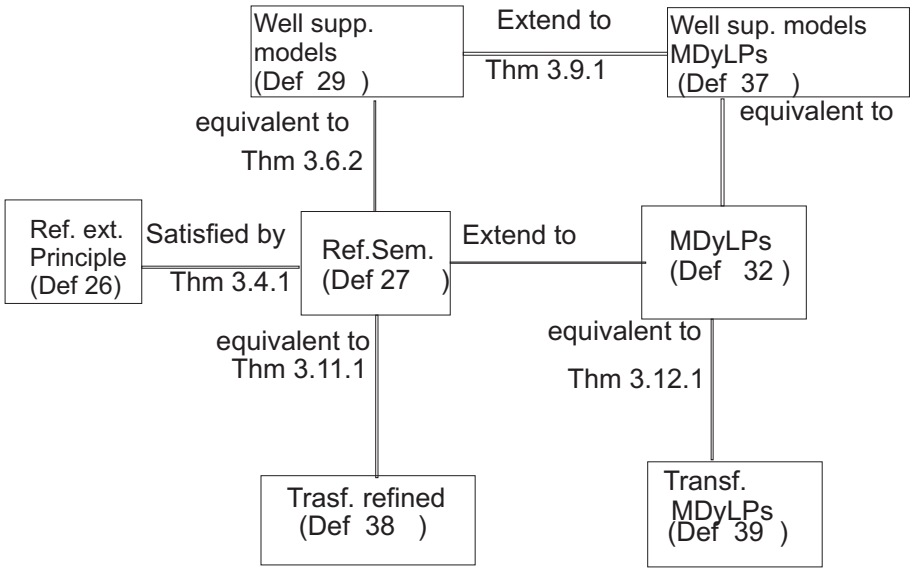
3.1.4 Structure of the chapter

The rest of the chapter is organized as follows. Section 3.2 recalls some preliminary notions and previous works on DyLPs, MDyLPs an well-supported models and establishes notation. Section 3.3 is devoted to motivate and present the refined extension principle, while in Section 3.4 a refined semantics for logic program updates that complies with the principle is presented.

Section 3.5 is devoted to compare the new semantics with other existing semantics for DyLPs, and to analyze these with respect to the refined extension principle. Section 3.6 extends the concept of well-supported model to DyLPs and proves that the refined dynamic stable models of a DyLP coincide with its well-supported models. Section 3.7 investigates further properties of the refined well-supported seman-

tics. Section 3.8 utterly extends the notion of well-supported model to MDyLPs and provides an equivalent fix-point characterization that extends the refined semantics to MDyLPs and Section 3.9 provides an equivalent fixpoint characterization of the well supported semantics for MDyLPs. Section 3.10 compares the new semantics with other existing semantics for MDyLPs. Section 3.11 defines, by a program transformation, the operational equivalent of the refines semantics and proves results of soundness, completeness and complexity of the semantics. Section 3.12 shows analogue results for the multidimensional case. Finally, some concluding remarks are drawn in Section 3.13. In Appendix A, the reader can find all the proofs of the results presented throughout the chapter.

The figure below summarizes the main results of this Chapter pointing out to the main definitions and the theorems relating them.



3.2 Preliminaries and previous works on updates

A *dynamic logic program* $\mathcal{P} : P_1 \ldots P_n$, (*DyLP*) is a sequence of generalized logic programs. Let $\mathcal{P} = (P_1, ..., P_n)$ and $\mathcal{P}' = (P'_1, ..., P'_m)$ be two DyLPs and k an index with $1 \leq k \leq n$. By $\mathcal{P}^k$ we denote the DyLP formed by the first k elements $(P_1, \ldots, P_k)$ of $\mathcal{P}$. We use $\mathcal{P} \oplus \mathcal{P}'$ to denote the sequence of programs obtained by concatenating the elements of the two sequences i.e.

$$\mathcal{P} \oplus \mathcal{P}' = P_1, ..., P_n, P'_1, ..., P'_m$$

If $n = m$ we use $\rho(\mathcal{P})$ to denote the set of all rules appearing in the programs $P_1, ..., P_n$, and $\mathcal{P} \cup \mathcal{P}'$ to denote the DyLP $(P_1 \cup P'_1, ..., P_n \cup P'_n)$. According to all semantics based on causal rejection of rules, an interpretation M models a DyLP, according to a semantic *Sem* iff

$$M = \Gamma^{Sem}(\mathcal{P}, M) \tag{3.1}$$

where

$$\Gamma^{Sem}(\mathcal{P},M) = least\left(\rho(\mathcal{P}) \setminus Rej^{Sem}(\mathcal{P},M) \cup Def^{Sem}(\mathcal{P},M)\right)$$

and where $Rej^{Sem}(\mathcal{P},M)$ stands for the set of rejected rules according to the semantic *Sem* and $Def^{Sem}(\mathcal{P},M)$ for the set of default assumptions according to the semantic *Sem*, both given $\mathcal{P}$ and M. Intuitively, we first determine the set of rules from $\mathcal{P}$ that are not rejected, i.e. $\rho(\mathcal{P}) \setminus Rej^{Sem}(\mathcal{P},M)$, to which we add a set of default assumptions $Def^{Sem}(\mathcal{P},M)$. Note the similarity to the way stable models of generalized logic programs (see Definition 13) are obtained, where default assumptions of the form *not A* are added for every $not\,A \in M^-$.

A DyLP $P = (P_1,...,P_n)$ represents not only the current knowledge but also the *history of the knowledge*. Given any P_k with $k \leq n$, the sequence $P^k = (P_1,...,P_k)$ represents the evolution of $\mathcal{P}$ up to the moment when the update P_k was added to the knowledge base. A straightforward generalization of the notion of semantic $\mathcal{P}$ is that of semantics of $\mathcal{P}$ *at the update* P_k. A two-valued interpretation M is a model of $\mathcal{P}$ at P_k according to the semantic *Sem* iff

$$M = \Gamma^{Sem}\left(\mathcal{P}^k, M\right) \tag{3.2}$$

From [Lei03] it is easy to see that all existing semantics for updates based on causal rejection are parameterizable using different definitions of $Rej^{Sem}(\mathcal{P},M)$ and $Def^{Sem}(\mathcal{P},M)$. The *dynamic stable model* semantics [ALP+98b, Lei03] for DyLP is defined as follows.

Definition 19 *Let $\mathcal{P}$ be a dynamic logic program and M an interpretation. M is a* dynamic stable model *of $\mathcal{P}$ iff*

$$M = \Gamma(\mathcal{P},M)$$

where

$$\Gamma(\mathcal{P},M) = least\left(\rho(\mathcal{P}) \setminus Rej(\mathcal{P},M) \cup Def(\mathcal{P},M)\right) \tag{3.3}$$

and

$$\begin{aligned} Rej(\mathcal{P},M) &= \left\{r \mid r \in P_i, \exists r' \in P_j, i < j, r \bowtie r', M \vdash B(r')\right\} \\ Def(\mathcal{P},M) &= \{not\,A \mid \nexists r \in \rho(\mathcal{P}), H(r) = A, M \vdash B(r)\} \end{aligned}$$

Another semantics based on causal rejection is the justified update ($\mathcal{JU}$) semantics whose definition is as above [LP97].

Definition 20 *Let $\mathcal{P}$ be a dynamic logic program and M an interpretation. M is a* justified update stable model *of $\mathcal{P}$ iff*

$$M = \Gamma^J(\mathcal{P},M)$$

where

$$\Gamma^J(\mathcal{P},M) = least\left(\rho(\mathcal{P}) \setminus Rej(\mathcal{P},M) \cup M^-\right) \tag{3.4}$$

Yet another proposal based on the same principle is the update programs semantics ($\mathcal{UP}$) [EFST02b]. Inheritance programs (see [BFL99]) are defined for disjunctive logic programs, but if we restrict to the non disjunctive case, this semantics coincides with the update answer-set semantics of [EFST02a]. These semantics were originally defined on top of the class of extended logic programs. We provide here a reformulation of these semantics based on GLPs and originally presented in [Lei03].

Definition 21 *Let $\mathcal{P}$ be a dynamic logic program and M an interpretation. M is an* update stable model *of $\mathcal{P}$ iff*

$$M = \Gamma^U(\mathcal{P},M)$$

where

$$\Gamma^U(\mathcal{P},M) = least\left(\rho(\mathcal{P}) \setminus Rej^U(\mathcal{P},M) \cup M^-\right)$$

and

$$Rej^U(\mathcal{P},M) = \left\{ \begin{array}{l} \tau \in P_i : \exists \eta \in P_j \setminus Rej^U(\mathcal{P},M),\ i<j \\ H(\tau) = \neg H(\eta),\ M \models B(\tau),\ M \models B(\eta) \end{array} \right\}$$

where notation $\neg H(\tau)$ stands for the explicit complement of the head of τ (the atom A is the explicit complement of $\neg A$ and viceversa).

The dynamic stable model semantics ($\mathcal{DS}$) the justified update ($\mathcal{JU}$) and the update programs semantics ($\mathcal{UP}$) are all extensions of the stable model semantics for normal and generalized LPs [GL88]; relations among them have been studied in [Hom04, Lei03], and the main result is as follows. Given any DyLP $\mathcal{P}$, let $\mathcal{UP}(\mathcal{P}), \mathcal{JU}(\mathcal{P})$ and $\mathcal{DS}(\mathcal{P})$ be the set of models of $\mathcal{P}$ according to, respectively, the update programs, the justified update, and the dynamic stable model semantics Then, the following (possibly strict) inclusions hold:

$$\mathcal{UP}(\mathcal{P}) \supseteq \mathcal{JU}(\mathcal{P}) \supseteq \mathcal{DS}(\mathcal{P}) \tag{3.5}$$

3.2.1 Well-supported models

The semantic analysis which we will make in this paper rests on the notion of *level mapping* over a set of atoms $\mathcal{L}$, where a *level mapping* ℓ is a function from $\mathcal{L}$ to the set of natural numbers. We also lift ℓ to negative literals of the form *not A*, where A is an element of $\mathcal{L}$, by setting $\ell(not\ A) = \ell(A)$. Given a conjunction of literals $C = L_1, \ldots, L_n$ we further extend ℓ by assigning to C the value $\ell(L_i)$, where i is chosen such that the value of $\ell(L_i)$ is maximal, i.e. $\ell(C) = \max(\{\ell(L_i) : L_i \in C\})$. For convenience — and by

slight abuse of notation — we assign the value -1 to the empty conjunction of literals. Our approach is stimulated by recent results on uniform characterizations of different semantics for LPs in terms of level mappings as introduced in [HW05] and extended in [Hit03, HS05]. This perspective provides an additional tool and guidelines on how to obtain reasonable new semantics for new classes of programs.

A *normal logic program* over a language $\mathcal{L}$ is any (possibly countably infinite) set of rules of the form $A \leftarrow B$, where A is any atom of $\mathcal{L}$ and B is any conjunction of literals of $\mathcal{L}$. Several different (two-valued) semantics are being used for assigning meaning to programs, including the *supported model semantics* [AB94], the *minimal supported model semantics* [AB94] and the *stable model semantics* [GL88].

3.2.2 Multidimensional dynamic logic programs

A *Multidimensional dynamic logic program* $\mathcal{MP}$ is any partially ordered finite multiset of generalized logic programs. Let $\mathcal{M}$ be a set of indexes for the elements of $\mathcal{MP}$, and $\prec$ the partial order defined over $Multi$. For any index i, by P_i we denote the element (also called update) of $\mathcal{MP}$ associated with i. We often use the notation $i \prec j$ instead of $P_i \prec P_j$. Let P_i and P_j be elements of $\mathcal{M}$, we say P_i is *less recent* than P_j (or, alternatively, that P_j is *more recent* than P_i) iff $i \prec j$. Let P_n be an update of $\mathcal{MP}$. The *genealogy of* P_n, denoted $\mathcal{MP}^n$, is the subset of the elements of $\mathcal{MP}$ which are less recent than P_j plus P_n itself. By $\rho(P^n)$ we denote the multiset of all rules of $\mathcal{MP}^n$.

Note that, if the order defined over the MDyLP $\mathcal{MP}$ consisting of m element is a *total order*, $\mathcal{MP}$ is the DyLP $P_1, \ldots, P_m$ where P_i denotes the i^{th} element of $\mathcal{MP}$ such that $P_i \prec P_j$ iff $i < j$.

As we said in the introduction of this chapter, many of the known semantics for DyLPs has been extended to the multidimensional case [BFL99, LAP00]. In [LAP00] MDyLPs are propose with a different though equivalent formulation using directed acyclic graphs (or diagraphs) instead of partially ordered sets.

Indeed, given any finite partially order set S, a graph G is naturally associated to S in the following way.. The nodes of G are exactly the elements of S. If n_1 and n_2 are nodes of G, a directed edge exists from n_1 to n_2 iff n_2 is a successor of n_2. It is easy to prove that G is acyclic i.e., there exists no directed cyclic path in G. On the other way, let G be a diagraph and S the sets of its elements. We define the relation $\prec$ other the elements of S in the following way: $n_1 \prec n_2$ iff there exists a directed path from n_1 to n_2. It is easy to prove that the relation $\prec$ is a partial order.

We present here a definition of *dynamic multi stable models* semantics which is equivalent to the definition given in [LAP00].

Definition 22 *Let $\mathcal{MP}$ be any multidimensional dynamic logic program in the language $\mathcal{L}$, P_n and element of $\mathcal{MP}$ and M any two-valued interpretation over $\mathcal{L}$. The Γ^D operator is defined*

as follows.

$$\Gamma^{D}_{(\mathcal{MP},n)}(M) = least(\rho(\mathcal{MP}^n) \setminus Rej(\mathcal{MP}^n, M) \cup Default(\mathcal{MP}^n, I))$$

Where $Rej(\mathcal{MP}^n, M)$ *is defined as follows*

$$Rej(\mathcal{MP}^n, M) = \{\tau \in P_i : \exists \eta \in P_j, i \prec j : \tau \bowtie \eta,\ M \models B(\eta)\}$$

Definition 23 *Let* $\mathcal{MP}$ *be any multidimensional dynamic logic program in the language* $\mathcal{L}$*,* P_n *and element of* $\mathcal{MP}$ *and* M *any two-valued interpretation over* $\mathcal{L}$*. We say* M *is a* dynamic multi stable model *of* $\mathcal{MP}$ *at* P_n *iff:*

$$M = \Gamma^{D}_{(\mathcal{MP},n)}(M)$$

In [Lei03] and [Hom04] the author proves that any dynamic stable model of a MDyLP $\mathcal{MP}$ at P_n is also a model of $\mathcal{MP}$ at P_n w.r.t. the semantics proposed in [BFL99, LP97].

3.3 Refined extensions

We are interested in exploring conditions guaranteeing that the addition of a set of rules to a (dynamic) logic program does not generate new (dynamic) stable models. In this section, we motivate and introduce the notion of refined extension for both the case of single logic programs and dynamic logic programs, which, together with the results proven, constitute a step in such direction.

3.3.1 Refined extensions of generalized logic programs

Informally, the semantics based on stable models are obtained by taking the least model of the definite program obtained by adding some assumptions (default negations) to the initial program i.e., the stable models of a generalized logic program P are those interpretations M such that M coincides with the least model of the program

$$P \cup \{not\ A \mid A \notin M\}$$

In general, several semantics share the characteristic that the models of a program P can be characterized as the least model of the program

$$P \cup Assumptions(P, M)$$

where $Assumptions(P, M)$ is simply a set of default literals whose definition depends on the semantics in use. Note that all of the stable models [GL88], the well founded [GRS91] and the weakly perfect model semantics [PP88] can be defined in this way. As

mentioned above, the stable model semantics of normal and generalized programs can be obtained by establishing that

$$Assumptions(P,M) = \{not\ A \mid A \notin M\}.$$

In the sequel, if *Sem* is a semantics definable in such a way, by $Sem(P)$ we denote the set of all models of a given program P, according to *Sem*.

With the intention of defining the refined extension principle, we first need to set forth some intermediate notions, namely that of *syntactic extension* of a program. Intuitively we say that $P \cup E$ is a syntactic extension of P iff the rules in E have no effect on the least model of P. Formally:

Definition 24 *Let P and E be generalized logic programs. We say that $P \cup E$ is a* syntactic extension *of P iff $least(P) = least(P \cup E)$.*

Let us consider now a generalized program P, and a set of rules E. A model M of $P \cup E$ according to *Sem* is computed as the least model of the definite logic program obtained by adding the set of default assumptions to $P \cup E$. We can then apply the concept of syntactical extension to verify whether the addition of the rules in E does not influence the computation of M. If this is the case for all models of the program $P \cup E$, according to *Sem*, we say that $P \cup E$ is a *refined extension* of the original program P.

Definition 25 *Let P and E be generalized logic program, M an interpretation, and Sem a semantics for generalized logic programs. We say that $P \cup E$ is an* extension *of P with respect to Sem and M iff*

$$P \cup Assumptions(P \cup E, M) \cup E$$

is a syntactic extension of

$$P \cup Assumptions(P \cup E, M)$$

We say that $P \cup E$ is a refined extension *of P with respect to Sem iff $P \cup E$ is an extension of P with respect to Sem and all models in $Sem(P \cup E)$.*

Example 3.3.1 *Let P_1 and P_2 be the programs of example 3.1.4:*

$$\begin{array}{ll} P_1: & day \leftarrow not\ night. \\ & night \leftarrow not\ day. \\ & stars \leftarrow night, not\ cloudy. \\ & not\ stars. \end{array}$$

$$P_2: \quad stars \leftarrow stars$$

It is clear that irrespective of what the set $Assumptions(P_1 \cup P_2, M)$ of added assumptions is, given any model M, we have that the least model of $P_1 \cup Assumptions(P_1 \cup P_2, M)$ coincides

with the least model of $P_1 \cup Assumptions(P_1 \cup P_2, M) \cup P_2$. *Thus,* $P_1 \cup P_2$ *is a refined extension of* P_1.

We are now ready to formulate the refined extension principle for semantics of generalized logic programs. Intuitively, a semantics complies with the refined extension principle iff a refined extension of a program P does not have more models than P.

Principle [Refined extension – static case] A semantics *Sem* for generalized logic programs complies with the refined extension principle iff for any two generalized logic programs P and E, if $P \cup E$ is a refined extension of P then

$$Sem(P \cup E) \subseteq Sem(P).$$

♢

As one may expect, the principle properly deals with the case of adding tautologies, i.e., for any semantics *Sem* that complies with the principle, the addition of tautologies does not generate new models.

Proposition 3.3.1 *Let Sem be a semantics for generalized programs, P a generalized program, and* τ *a tautology. If Sem complies with the refined extension principle then*

$$Sem(P \cup \{\tau\}) \subseteq Sem(P). \tag{3.6}$$

Most importantly, the stable model semantics complies with the refined extension principle, as stated in the following proposition.

Proposition 3.3.2 *Let P be any generalized logic program,* $P \cup E$ *be a refined extension of P, and M a stable model of* $P \cup E$. *Then M is also a stable model of P.*

As an immediate consequence of these two propositions, we get that the addition of tautologies to a generalized program does not introduce new stable models. The converse is also true i.e. the addition of tautologies to a generalized program does not eliminate existing stable models.

3.3.2 Refined extensions of dynamic logic programs

We now generalize the refined extension principle to the case of dynamic logic programs, so as to guarantee that, according to the semantics that comply with it, certain updates do not generate new models.

Definition 26 *Let $\mathcal{P}$ and $\mathcal{E}$ be two dynamic logic programs*[4]*, Sem a semantics for dynamic logic programs and M an interpretation. We say that $\mathcal{P} \cup \mathcal{E}$ is an* extension *of $\mathcal{P}$ with respect to M iff*

$$\rho(\mathcal{P}) \setminus Rej^{Sem}(\mathcal{P} \cup \mathcal{E}, M) \cup Def^{Sem}(\mathcal{P} \cup \mathcal{E}, M) \cup \rho(\mathcal{E}) \setminus Rej^{Sem}(\mathcal{P} \cup \mathcal{E}, M)$$

is a syntactical extension of

$$\rho(\mathcal{P}) \setminus Rej^{Sem}(\mathcal{P} \cup \mathcal{E}, M) \cup Def^{Sem}(\mathcal{P} \cup \mathcal{E}, M)$$

We say that $\mathcal{P} \cup \mathcal{E}$ is a refined extension *of $\mathcal{P}$ iff $\mathcal{P} \cup \mathcal{E}$ is an extension of $\mathcal{P}$ with respect to all models in $Sem(\mathcal{P} \cup \mathcal{E})$.*

Note that the Definition 26 is the straightforward lifting of Definition 25 to the case of dynamic logic programs. Roughly speaking, we simply replaced P and E with $\rho(\mathcal{P}) \setminus Rej^{Sem}(\mathcal{P} \cup \mathcal{E}, M)$ and $\rho(\mathcal{E}) \setminus Rej^{Sem}(\mathcal{P} \cup \mathcal{E}, M)$, respectively. The refined extension principle is then formulated as follows.

Principle[Refined extension principle]A semantics *Sem* for dynamic logic programs complies with the refined extension principle iff for any dynamic logic programs $\mathcal{P}$ and $\mathcal{P} \cup \mathcal{E}$, if $\mathcal{P} \cup \mathcal{E}$ is a refined extension of $\mathcal{P}$ then

$$Sem(\mathcal{P} \cup \mathcal{E}) \subseteq Sem(\mathcal{P}).$$

$\diamondsuit$

Again, this principle amounts to the straightforward lifting of Principle 3.3.1 Unfortunately, none of the existing semantics for dynamic logic programs based on causal rejection complies with the refined extension principle.

Example 3.3.2 *Let us consider again the program P_1 and P_2 of Example 3.1.5. The presence of the new model contrasts with the refined extension principle. Indeed, if we consider the empty update $P_\emptyset$, then the dynamic logic program $(P_1, P_\emptyset)$ has only one stable model (viz., $\{day\}$). Since, as the reader can check, (P_1, P_2) is a refined extension of $(P_1, P_\emptyset)$ then, according to the principle, all models of (P_1, P_2) should also be models of $(P_1, P_\emptyset)$. This is not the case for existing semantics.*

If we consider infinite logic programs, there is also another class of examples where all the previously existing semantics based on causal rejection fail to satisfy the refined extension principle. This class of examples consists of sequences of updates containing

[4]Here, and elsewhere, we assume that the sequences of GLPs (or DyLPs) $\mathcal{P}$ and $\mathcal{E}$ are of the same length.

an infinite number of conflicting rules whose bodies are satisfied i.e., an infinite number of potential contradictions where each is removed by a later update, of which the following example serves as illustration.

Example 3.3.3 *Let us consider an infinite language consisting of the constant* 0 *and of the unary function successor. The set of terms of such a language has an obvious bijection to the set of natural numbers. In this context, given a term* n*, we will use the expression* $n+1$ *for* $successor(n)$*. Let* q *be an unary predicate and* $\mathcal{H}$ *the set of propositional atoms consisting of all the predicates of the form* $q(n)$*.*

Let us consider the following programs:

$$P_1: \quad q(X). \\ \quad\quad not\ q(X).$$

$$P_2: \quad q(X) \leftarrow q(X+1).$$

Since P_1 *and* P_2 *contain the variable* X*, in order to compute the stable model semantics, we have to consider the ground versions of the two programs i.e., the following two infinite programs:*

$$P_1^{ground}: \quad q(n).\ \forall\, n \in \mathbb{N} \\ \quad\quad not\ q(n).\ \forall\, n \in \mathbb{N}$$

$$P_2^{ground}: \quad q(n) \leftarrow q(n+1).\ \forall\, n \in \mathbb{N}$$

All existing semantics based on causal rejection admit the interpretation $I = \mathcal{A}$*, consisting of all the predicates of the form* $q(n)$*, as a model of the program* $(P_1^{ground}, P_2^{ground})$*. Since the program* $(P_1^{ground}, P_\emptyset)$ *is clearly contradictory, hence having no stable models, and the program* $(P_1^{ground}, P_2^{ground})$ *is a refined extension of* $(P_1^{ground}, P_\emptyset)$*, then, according to the refined extension principle, it should be taken as contradictory and have no models.*

As for the case of generalized programs, if we consider a semantics *Sem* for dynamic logic programs that complies with the principle, the addition of tautologies does not generate new models. This is stated in the following proposition that lifts Proposition 3.3.1 to the case of dynamic logic programs.

Proposition 3.3.3 *Let Sem be a semantics for dynamic logic programs,* $\mathcal{P}$ *a dynamic logic program, and* $\mathcal{E}$ *a sequence of sets of tautologies. If Sem complies with the refined extension principle then*

$$Sem(\mathcal{P} \cup \mathcal{E}) \subseteq Sem(\mathcal{P}).$$

3.4 Refined semantics for dynamic logic programs

We define a new semantics for dynamic logic programs that complies with the refined extension principle.

Before proceeding we take a moment to analyze the reason why the dynamic stable model semantics fails to comply with the refined extension principle in Example 3.1.4. In this example, the extra (counterintuitive) dynamic stable model $\{night, stars\}$ is obtained because the tautology $stars \leftarrow stars$ in P_2 has a true body in that model, hence rejecting the fact *not stars* of P_1. After rejecting this fact, it is possible to consistently conclude *stars*, and thus verify the fixpoint condition (Equation 3.3), via the rule $stars \leftarrow night, not\ cloudy$ of P_1.

Here lies the matrix of the undesired behavior exhibited by the dynamic stable model semantics: One of the two conflicting rules in the same program (P_1) is used to support a later rule (of P_2) that actually removes that same conflict by rejecting the other conflicting rule. Informally, rules that should be irrelevant may become relevant because they can be used by one of the conflicting rules to defeat the other.

A simple way to inhibit this behavior is to let conflicting rules in the same update inhibit each other. This can be obtained with a slight modification to the notion of rejected rules of the dynamic stable model semantics, namely by also allowing rules to reject other rules in the same update. Since, according to the dynamic stable model semantics, rejected rules can reject other rules, two rules in the same update can reject each other, thus avoiding the above described behavior.

Definition 27 *Let $\mathcal{P}$ be a dynamic logic program and M an interpretation. M is a* refined dynamic stable model *(or simply a* refined model*) of $\mathcal{P}$ iff*

$$M = \Gamma^R(\mathcal{P}, M)$$

where

$$\Gamma^R(\mathcal{P}, M) = least\left(\rho(\mathcal{P}) \setminus Rej^R(\mathcal{P}, M) \cup Def(\mathcal{P}, M)\right)$$

and

$$Rej^R(\mathcal{P}, M) = \left\{r \mid r \in P_i, \exists r' \in P_j, i \leq j, r \bowtie r', M \vdash B(r')\right\}$$

At first sight this modification could seem to allow the existence of models in cases where a contradiction is expected (e.g. in a sequence where the last program contains facts for both A and *not* A): if rules in the same update can reject each other then the contradiction is removed, and the program could have undesirable models. Notably, the opposite is actually true (cf. Theorem 3.5.1 below), and the refined dynamic stable models are always dynamic stable models, i.e., allowing the rejection of rules by rules in the same update does not introduce extra models.

To better understand this behavior, consider a DyLP $\mathcal{P}$ with two conflicting rules (with heads A and *not* A) in one of its programs P_i. Take an interpretation M where the bodies of those two rules are both true (as nothing special happens if a rule with false body is rejected) and check if M is a refined dynamic stable model. By Definition 27, these two rules reject each other, and reject all other rules with head A or *not* A

in that or in any previous state. Moreover, *not A* cannot be considered as a default assumption, i.e., does not belong to $Def(\mathcal{P},M)$ because of the rule with head A and true body. This means that all the information about A (a rule with head A or *not A*) with origin in P_i or any previous state is deleted. Since $\overline{M}$ must contain either A or *not _A*, the only possibility for M to be a stable model is that there exists a rule τ, in some later update, whose head is either A or *not A*, and whose body is true in M. This means that a potential inconsistency can only be removed by some later update. If such later update does not exist, then, even though the two rules that cause the contradiction reject each other, we still do not obtain any models.

Finally, as this was the very motivation for introducing the refined semantics, it is worth observing that the refined semantics does comply with the refined extension principle, as stated by the following Theorem.

Theorem 3.4.1 *The refined semantics complies with the refined extension principle.*

By Proposition 3.3.3, it immediately follows from this theorem that the addition of tautologies never adds models in this semantics. Note that the converse is also true: the addition of tautologies does not eliminate existing models in the refined semantics i.e., the refined semantics is immune to tautologies, as stated by the following Theorem.

Theorem 3.4.2 *Let $\mathcal{P}$ be any DyLP and $\mathcal{E}$ a sequence of sets of tautologies. M is a refined stable model of $\mathcal{P}$ iff M is a refined stable model of $\mathcal{P} \cup \mathcal{E}$.*

To give an insight view on the behavior of the refined semantics, we now illustrate how the counterintuitive results of example 3.1.4 are eliminated.

Example 3.4.1 *Let us consider again the DyLP $\mathcal{P} = (P_1, P_2)$ of example 3.1.4*

$$
\begin{array}{ll}
P_1: & day \leftarrow not\ night. \\
& night \leftarrow not\ day. \\
& stars \leftarrow night, not\ cloudy. \\
& not\ stars.
\end{array}
$$

$$
P_2: \quad stars \leftarrow stars
$$

This DyLP has one refined dynamic stable model, $M = \{day\}$. Thus the conclusions of the semantics match with the intuition that it is day and it is not possible to see the stars.

We now show that M is a refined dynamic stable model. First of all we compute the sets $Rej^R(\mathcal{P},M)$ and $Def(\mathcal{P},M)$:

$$
\begin{aligned}
Rej^R(\mathcal{P},M) &= \{stars \leftarrow night, not\ cloudy.\} \\
Def(\mathcal{P},M) &= \{not\ night, not\ stars, not\ cloudy\}
\end{aligned}
$$

Then we check whether M is a refined dynamic stable model according to Definition 27. Indeed:

$$\begin{aligned}\Gamma^R(\mathcal{P},M) &= least\left((P_1\cup P_2)\setminus Rej^R(\mathcal{P},M)\cup Def(\mathcal{P},M)\right)= \\ &= \{day, not\ night, not\ stars, not\ cloudy\} = M\end{aligned}$$

As mentioned before, the dynamic stable model semantics, besides M, also admits the interpretation $N = \{night, stars\}$ as one of its models, thus violating the refined extension principle. We now show that N is not a refined dynamic stable model. As above we compute the sets:

$$\begin{aligned}Rej^R(\mathcal{P},N) &= \{not\ stars;\quad stars \leftarrow night, not\ cloudy.\} \\ Def(\mathcal{P},N) &= \{not\ day, not\ cloudy\}\end{aligned}$$

Hence:

$$\begin{aligned}\Gamma^R(\mathcal{P},N) &= least\left((P_1\cup P_2)\setminus Rej^R(\mathcal{P},N)\cup Def(\mathcal{P},N)\right)= \\ &= \{night, not\ day, not\ cloudy\} \neq N\end{aligned}$$

From where we conclude, according to Definition 27, that N is not a refined dynamic stable model.

On the other hand we have

$$Rej(\mathcal{P},N) = \{not\ stars\}$$

and hence

$$\begin{aligned}\Gamma(\mathcal{P},N) &= least\left((P_1\cup P_2)\setminus Rej(\mathcal{P},N)\cup Def(\mathcal{P},N)\right)= \\ &= \{night, not\ day, stars, not\ cloudy\} = N\end{aligned}$$

Which, according to Definition 19 means that N is a dynamic stable model. This implies that *the dynamic stable model semantics does not coincides with the refined semantics*.

Moreover, if we consider the DyLP $\mathcal{P}' = P_1, P_\emptyset$ i.e. the DyLP obtained by deleting the tautology in P_2, we have:

$$\begin{aligned}Rej\left(\mathcal{P}',N\right) &= \{\} \\ Def\left(\mathcal{P}',N\right) &= \{not\ day, not\ cloudy\}\end{aligned}$$

and hence

$$\begin{aligned}\Gamma\left(\mathcal{P}',N\right) &= least\left((P_1\cup P_\emptyset)\setminus Rej\left(\mathcal{P}',N\right)\cup Def\left(\mathcal{P}',N\right)\right)= \\ &= \{night, not\ day, stars, not\ starsnot\ cloudy\} \neq N\end{aligned}$$

So is a not dynamic stable model of $\mathcal{P}'$ but it is a stable model of $\mathcal{P}$, although the two differs only for the addition of one tautology. By this example and Proposition 3.3.3 we conclude that *the dynamic stable model semantics does not comply with the refined extension principle*. The same conclusions holds for the other semantics based on causal rejection as we shall see in the sext section.

3.5 Relationship with other semantics for dynamic logic programs

As shown from the discussion of example 3.4.1 above, the dynamic stable model semantics does not coincide with the refined semantics and, it does not respect the refined extension principle. It is clear from the definitions that the refined semantics coincides with the dynamic stable model semantics [ALP$^+$98b, ALP$^+$00b, Lei03] for sequences of programs with no conflicting rules in a same program. This means that the dynamic stable model semantics complies with the refined extension principle for such class of sequences of programs, and would have no problems if one restricts its application to that class. However, such limitation would reduce the freedom of the programmer, particulary in the possibility of using conflicting rules to represent integrity constraints. Another limitation would result from the fact that updates also provide a tool to remove inconsistency in programs by rejecting conflicting rules. Such feature would be completely lost in that case.

With respect to the other semantics based on causal rejection, it is not even the case that the principles is satisfied by sequences in that restricted class. We can verify this as in the case of the dynamic stable models semantics by relying on Proposition 3.3.3 since update answer-set semantics [EFST02a] and the justified update semantics [LP97, LP98] fail to be immune to tautologies even when no conflicting rules occur in the same program as shown in the examples below whose detailed discussion can be found in [Lei03].

Example 3.5.1 *Let us consider the DyLP* $\mathcal{P} = (P_1, P_2, P_3)$ *where:*

$$P_1: \quad day.$$

$$P_2: \quad not\ day.$$

$$P_3: \quad day \leftarrow day.$$

stating that initially it is day time, then it is no longer day time, and finally (tautologically) stating that whenever it is day time, it is day time. While the semantics of justified updates [LP97, LP98] and the dynamic stable model semantics [ALP$^+$98b, Lei03] select $\{day\}$ *as the*

only model, the update answer set semantics of [EFST02a] (as well as inheritance programs of [BFL99]) associates two models, $\{day\}$ and $\{\}$, with such sequence of programs[5].

While the semantics of justified updates [LP97, LP98] works properly for the above example, there are classes of programs where its behavior shows its failure to comply with the refined extension principle.

Example 3.5.2 *Let us consider the DyLP $\mathcal{P} = (P_1, P_2)$ where:*

$$P_1: \quad day.$$

$$P_2: \quad not\ day \leftarrow not\ day.$$

According to the semantics of justified updates, (P_1, P_2) has two models, $M_1 = \{day\}$ and $M_2 = \{\}$, whereas $(P_1, P_\emptyset)$ has the single model M_1, thus violating the refined extension principle.

Finally, observe that the refined semantics is more credulous than all the other semantics, in the sense that the set of its models is always a subset of the set of models obtained with any of the others thus making its intersection larger. Comparing first with the dynamic stable model semantics.

Theorem 3.5.1 *Let $\mathcal{P}$ be a DyLP, and M an interpretation. If M is a refined dynamic stable model then M is a dynamic stable model.*

This result generalizes to all the other semantics since the dynamic stable model is the most credulous of the existing semantics. Hence we had the refined stale models of a DyLP $\mathcal{P}$ (denoted $\mathcal{RS}(\mathcal{P})$ as another element at the bottom of the chain of set inclusions 3.5 obtaining the following chain of inclusions.

$$\mathcal{UP}(\mathcal{P}) \supseteq \mathcal{JU}(\mathcal{P}) \supseteq \mathcal{DS}(\mathcal{P}) \supseteq \mathcal{RS}(\mathcal{P}) \tag{3.7}$$

In [EFST02a] the authors define a refinement of the justified updates semantics named *minimal updates semantics*. Since this semantics is strongly linked to a property called *minimality of change* we will discuss its link with the refined semantics in Section 3.7 discussing further properties of the refined semantics.

3.5.1 Semantics for DyLPs not based on causal rejection

For sake of completeness, we briefly discuss two approaches to logic programs updates not based on causal rejection. Namely *Program Updates through induction* from Inoue and Sakama (see [SI99]) and *Program Updates through Priorities* from Zhang and Foo

[5]Notice that, strictly speaking, the semantics [BFL99, EFST02a] are actually defined for extended logic programs, with explicit negation, rather than for generalized programs. However, the example can be easily adapted to extended logic programs.

(see [ZF98]). A more complete analysis and comparisons can be found, for instance, in [EFST02a, Lei03]. Both the frameworks are based on extended logic programs rather than on generalized logic programs.

The approach of Sakama and Inoue to updates is based on abduction. Given two (extended) logic programs P and Q the result of updating P with Q is obtained by making every rule of P defeasible and then eliminating such rules from the program $P \cup Q$ until the resulting program is consistent.

A first consideration that can be done is that, this way *Idempotence* (see Section 3.7 for further details) cannot be guaranteed, i.e. it is not always true that $P \oplus P_\emptyset \equiv P$. Indeed, if the program P is contradictory, the semantics automatically removes the rules in P causing the contradiction. This result comes since the approach of Inoue and Sakama mixes the separate concepts of *theory update* and *theory revision* (see [KM91] for a discussion on the difference between the two concepts.)

A second problem appears when successive updates are considered. If a rule τ is removed to solve a contradiction with an updated rule η with a true body, and later on the body η is falsified by another update, the rule τ is not resumed. For instance, let us consider the following example paraphrased by [Lei03]. Initially there it is a sunny day. Then clouds come with the wind and we know that if there are clouds, there can not be the sun. Finally the clouds are drawn away by the wind. This is formalized by the following DyLP (with explicit negation) $\mathcal{P} : P_1, P_2, P_3$.

$$P_1 : \ sun. \quad P_2 : \neg sun \leftarrow clouds. \quad clouds. \quad P_3 : \ \neg clouds.$$

If we replace explicit negation by default negation in the head we obtain a DyLP (with default negation in the head) whose only refined model at P_2 is $\{clouds\}$ and whose only refined model at P_3 is $\{sun\}$ matching the intuitive meaning of the program. According to the approach of Inoue and Sakama, if we first update P_1 with P_2 and then we update the resulting program with P_3, it is not possible to conclude the fact *sun* since it was rejected from the program after performing the first update P_2.

The approach of Zang and Foo is based on a mixture of models updates and preferences. The result is that, given two programs P_1 and P_2 the result of updating P_1 with P_2 is a *set* of logic programs rather than a single program related to the set of stable models of P_1. It is important to notice that successive updates determine an exponential explosion of the total number of programs. This aspect compromises the efficiency of the approach.

3.6 Well-supported models for DyLPs

The refined semantics of DyLP of Definition 27 complies with the refined extension principle. However, the refined semantics only covers the case of DyLPs while the more general multidimensional case remains uncovered. Moreover, as we will see in

Section 3.9 it is not possible to directly extend Definition 27 to the multidimensional case in order to obtain a refined semantics for multidimensional dynamic logic programs.

A parallel problem is that of establishing whether the refined semantics can be considered a final step in the research of a proper semantics for DyLPs. As we see from the chain of inclusions (3.7), the existing semantics for DyLPs can be ordered in a sequence where each semantics is a refinement of the previous one. Going right in the sequence, the conditions for an interpretation to be accepted as a model become stricter. It seems that research is trying to discard *bad models* from the semantics and to keep only *good models*. Is this really the case? Moreover, can we consider the sequence ended by the refined semantics or do we need to further refine it? To answer these questions we need a formal definition of what we mean by *good model*. Recalling Sections and 2.4.1 and 2.4.1 we can see an interesting historical parallel between the evolution of semantics for DyLPs and the evolution of two-valued semantics for normal LPs, where the supported, minimal supported and stable model semantics are successive refinements.

Given any program P, the set of all supported models ($SU(P)$), the set of all minimal supported models ($MSU(P)$) and the set of all stable models ($SM(P)$) of P are related by $SU(P) \supseteq MSU(P) \supseteq SM(P)$. The similarities are even more notable iff we consider some examples of strict inclusions of the sets of supported, minimal supported and stable models of a logic program.

Example 3.6.1 *Let us consider the program* $P: A \leftarrow A$. *The program* P *has unique SM* $\emptyset$ *which coincides with the unique minimal supported model. It has* $\{A\}$ *as a second supported model. All the cited semantics have* $\emptyset$ *as unique model of the program consisting of the empty set of clauses. Hence, for the supported model semantics adding tautologies of the form* $A \leftarrow A$ *to a program may change its semantics.*

Example 3.6.2 *Let us consider now the program*

$$\begin{aligned} P_1: \quad & A \leftarrow not\, A. \\ & A \leftarrow A. \end{aligned}$$

Program P_1 *has no stable models but has* $\{A\}$ *as unique minimal supported model. The program*

$$P_2: A \leftarrow not\, A$$

has no minimal supported model. Again, the introduction of the tautology $A \leftarrow A$ *has changed the semantics of the program.*

The concepts of supported and minimal supported model are successive approximations towards the concept of well-supported model. Let us consider, for instance, the examples 3.6.1 and 3.6.2. Regarding the first example, $\{A\}$ is a supported model

but not a well-supported model. Indeed, for every level mapping the head of the rule $A \leftarrow A$ is equal to the level of its body, hence this rule cannot be used to derive A under the assumption of well-supportedness. The same reasoning can be applied to the second example. These examples show that the supported and the minimal supported model semantics sometimes allow to derive conclusions that are not well-supported. The reader may notice similarities between examples 3.1.4 and 3.6.1, 3.6.2. In both cases, rules that should not play any semantic role change the behavior in some semantics by introducing more models. In the static case, this behavior is rectified by the introduction of the concept of well-supported model. Guided by this historical perspective, we extend the notion of WS model to DyLPs and MDyLPs. As shown by Theorem 3.6.2 this result will also provides an extension of the refined semantics to MDyLPs.

We first note that in the definition of well-supported models for normal LPs only the level of the positive literals in the body of a rule is considered. This happens because within normal LPs only positive literals are derived by rules, and the negative ones follow by negation by default. For DyLPs and MDyLPs, however, also negative literals can be derived by rules, since we allow negative literals in rule heads. More importantly, in the static case a rule plays a role only for deciding whether or not a given literal should be true or not. In the dynamic case, a rule is also used for *rejecting* other rules. Hence the concept of well-supportedness should be applied not only to the derivation of literals but also to the rejection of rules. More precisely, we require *well-supported rejection*: a rule can reject another rule in a previous update iff the body of the rejecting rule is true and the level of the body of the rejecting rule is less than the level of its head. The following definition formalizes this idea using level mappings.

Definition 28 [6] *Let $\mathcal{P}$ be any DyLP over a language $\mathcal{L}$, ℓ any level mapping over $\mathcal{L}$ and M any interpretation over $\mathcal{L}$, we define the set of rejected rules w.r.t. ℓ as:*

$$Rej_\ell(\mathcal{P},M) = \{\tau \in P_i : \exists\, \eta \in P_j,\ i < j\ \tau \bowtie \eta, M \models B(\eta),\ \ell(hd(\eta)) > \ell(B(\eta))\}$$

In the following we omit the argument $\mathcal{P}$ whenever the considered DyLP is clear from the context.

Given the considerations above and Definition 28, it is now quite simple to extend the concept of well-supported model to the dynamic case. An interpretation is a well-supported model if it is possible to find a level mapping such that the considered interpretation is a model of all the rules that are not rejected w.r.t. the given level mapping and such that, for each atom A which is true in the interpretation a non rejected rule with head A exists, whose body is true and the level of such body is less than the level of A.

[6] Hereafter we use the simplified notation $Rej_\ell(M)$ whenever this causes no ambiguity.

Definition 29 (Well-supported semantics of DyLPs) *Let $\mathcal{P}$ be any D_yLP in the language $\mathcal{L}$ and M any two-valued interpretation over $\mathcal{L}$. We say M is a **well-supported (WS) model** iff there exists a level mapping ℓ over $\mathcal{L}$ such that :*

i) *M is a model of $\rho(P) \setminus Rej_\ell(M)$*

ii) *$\forall A \in M \; \exists \tau \in \rho(P) \setminus Rej_\ell(M)$ such that $hd(\tau) = A$, $\ell(A) > \ell(B(\tau))$ and $M \models B(\tau)$.*

It is quite easy to see that Definition 29 coincides with Definition 11 when the considered DyLP is the single normal logic program P.

Theorem 3.6.1 *Let P be a DyLP consisting of a single update and let P be a normal logic program in the language $\mathcal{L}$. Moreover let and M a two-valued interpretation, over $\mathcal{L}$. M is a WS model according to Definition 29 iff it is a WS model according to Definition 11.*

proof Let M be a WS model of P according to Definition 29 and let ℓ be a level mapping satisfying conditions *i*) and *ii*) of Definition 29. Since P is a single program, the set of rejected rules w.r.t. ℓ is empty, hence $\rho(P) \setminus Rej_\ell(M) = P$. Hence, by condition *i*) of Definition 29, M is model of P, and hence condition *i* of Definition 11 is satisfied. Since ℓ satisfies condition *ii*) of Definition 29, it also trivially satisfies condition *ii*) of Definition 11.

Let now M be a WS model of P according to Definition 11 and let ℓ' be a level mapping satisfying condition *ii*) of Definition 11. Let further ℓ be the level mapping obtained from ℓ' in the following way. If $A \in M$, then $\ell(A) = \ell(A)' + 1$, otherwise $\ell(A) = 0$ The set of rejected rules w.r.t. ℓ is empty and hence $\rho(P) \setminus Rej_\ell(M) = P$. Hence, by condition *i*) of Definition 29, M is model of $\rho(P) \setminus Rej_\ell(M)$, and hence condition *i* of Definition 29 is satisfied. Clearly ℓ satisfies condition *ii*) of Definition 11, i.e. $\forall A \in M \; \exists \tau \in P$ *such that* $hd(\tau) = A$, $\ell(A) > \ell(A_1), \ldots \ell(A_n)$, for each A_i, where the $A_i s$ and are the positive literals belonging to the body of τ and $M \models B(\tau)$. Let the $not\, B_j s$ be the negative literals in the $B(\tau)$. By definition of ℓ, we know $\ell(A) > \ell(not\, B_j)$ for each j. Hence $\ell(A) > \ell(B(\tau))$ and hence ℓ satisfied condition *ii*) of Definition 29. Hence M is a WS model according to Definition 29. ♢

Theorem 2.4.3 states the equivalence between well-supported and stable models, thus providing both an alternative and characterization and a further legitimation of the stable models semantics. In the same way it is possible to prove a similar results in the dynamic case, precisely *The well-supported models of a DyLP are exactly its refined models*.

To prove this result we need two technical lemmata.

Lemma 3.6.1 *Let $\mathcal{P}$ be any D_yLP over the language $\mathcal{L}$, ℓ any level mapping defined on $\mathcal{L}$ and M any interpretation over $\mathcal{L}$. Then the following inclusion holds.*

$$\rho(P) \setminus Rej^R(M) \subseteq \rho(P) \setminus Rej(M) \subseteq \rho(P) \setminus Rej_\ell(M)$$

proof It is trivial to prove that $Rej(M) \subseteq Rej^R(M)$, since the condition for a rule to be rejected in Rej is stronger than in Rej^R. By Definition 28 it is also clear that $Rej_\ell(M) \subseteq Rej(M)$, since the former set of rejected rules is defined as the former one plus an extra condition based on level mappings. Hence we obtain: $Rej_\ell(M) \subseteq Rej(M) \subseteq Rej^R(M)$. The thesis trivially follows from this inclusion. ♢

Lemma 3.6.2 *Let $\mathcal{P}$ be any D_yLP over the language $\mathcal{L}$, ℓ any level mapping defined on $\mathcal{L}$, M a well-supported model of $\mathcal{P}$ and $L \in M$. Then, either $L \in Default(M)$ or there exists a rule $L \leftarrow B$ in $\rho(P) \setminus Rej^R(M)$ such that $M \models B$ and $\ell(L) > \ell(B)$.*

proof We proceed by cases.

- Let $L = not\, A$ for some atom A and there exists no rule $A \leftarrow B$ such that $M \models B$. Then, by definition, $M \in Default(M)$ and the thesis is satisfied.
- Let $L = A$ and there exists no rule with head $not\, A$ whose body is true in M. Then, by *ii*) there exists a rule $\tau : A \leftarrow B$ such that $M \models B$ and $\ell(A) > \ell(B)$. Clearly $\tau \in \rho(P) \setminus Rej^R(M)$ since there is no rule that can reject τ and the thesis is satisfied.
- Finally, let us suppose there exists a rule with head $not\, L$ and true body in M. Let α be the maximum index such that such a rule η^α exists in P_α. Since $L \in M$ then M is not a model of η^α. This means $\eta^\alpha \in Rej_\ell(M)$ i.e. there exists a rule τ^β with head L in P_β : $\alpha < \beta$, such that $M \models B(\tau^\beta)$ and $l(L) > l(B(\tau^\beta))$. Since α is maximal, there exists no rule with head $not\, L$ and true body in P_β or any successive update, hence $\tau^\beta \in \rho(P) \setminus Rej^R(M)$ as desired.

♢

We are now ready to prove the equivalent of Theorem 2.4.3 in the dynamic case.

Theorem 3.6.2 *Let $\mathcal{P}$ be any D_yLP . Any interpretation M is a refined stable model (RSM) of $\mathcal{P}$ iff it is a well supported model of $\mathcal{P}$.*

proof In the following we will use the notation $GS(M)$ for $\rho(P) \setminus Rej^R(M) \cup Default(M)$. We first suppose M is a RSM of $\mathcal{P}$, i.e. $M = least(GS(M))$. We have to prove that M is a well-supported model.

For this we have to prove conditions *i*) and *ii*) of Definition 29. We define the level mapping ℓ as follows: Let T_{GS} be the immediate consequence operator of the *definite* logic program $GS(M)$ Moreover, let T^m_{GS} be the m^{th} application of T_{GS} to the empty set. If *not A* is in $Default(M)$ then $\ell(A) = 1$, otherwise $\ell(A)$ is the minimum m such that either A or *not A* is in T^m_{GS}. Formally:

$$\ell(A) = min(\{m : A \in T^m_{GS} \text{ or not } A \in T^m_{GS}\})$$

- First of all we prove condition *ii*) We proceed by induction on the value of $\ell(L)$

  Basic step. Let A be any atom in M, if $\ell(A) = 1$ then, by lemma 28, A is a fact of $\rho(P) \setminus Rej^R(M)$, and hence, by lemma 3.6.1, A is a fact of $\rho(P) \setminus Rej_\ell(M)$ and hence *ii*) is satisfied.

  Inductive step. If $\ell(A) = m+1$ where m is positive, there exists a rule $\tau : A \leftarrow B$ in $GS(M)$ such that $B \subseteq T^m_{GS}$, then, $\ell(A) > \ell(B)$ and M satisfies B. By lemma 3.6.1, τ is also a rule of $\rho(P) \setminus Rej_\ell(M)$ and condition *ii*) is satisfied

- We prove now condition *i*) i.e. M is a model of $\rho(P) \setminus Rej_\ell(M)$. It is sufficient to prove that if τ^i is a rule in any P_i such that M is not a model of τ^i, then $\tau^i \in Rej_\ell(M)$.

  If τ^i is such a rule and L is its head, then in $M \models B(\tau^i)$ and $L \notin M$. Then, since M is two-valued, $not\ L \in M$. Let us consider $\ell(L)$. By definition of ℓ there have to be a rule $\eta^j : not\ L \leftarrow B$ in $GS(M)$ such that B is true in $last(GS(M)) = M$ and $\ell(L) > \ell(B)$. Since the body of τ^i is true, $not\, L$ does not belong to $Default(M)$. Then η^j belongs to some P_j. Moreover $j > i$, since τ^i rejects any rule whose head is $not\ L$ in any update P_k with $k \leq i$. We conclude, by Definition 28, that $\tau^i \in Rej_\ell(M)$ as desired.

Let now M be a WS model of $\mathcal{P}$. We have to prove that

$$M = least(GS(M))$$

- We start proving $least(GS(M)) \subseteq M$. First of all we notice that $Default(M) \subseteq M$. Indeed, if $not\ A \in Default(M)$ then it does not exist in P a rule $A \leftarrow B$ such that $M \models B$ hence, A is not supported and hence $A \notin M$, which means $not\ A \in M$. Moreover, by hypothesis we know M is a model of $P \setminus Rej_\ell(M)$ and by what previously said and lemma 3.6.1, it also follows that M is a model of $GS(M)$, this implies that $least(GS(M)) \subseteq M$ as desired.

- We have now to prove that, for any $L \in least(GS(M))$ it also holds $L \in M$. We proceed by induction on $\ell(L)$.

  Basic step. Let us suppose $\ell(L)$ is minimal. By lemma 3.6.2 either $L \in Default(M)$ or there exists a rule $\tau : L \leftarrow B$ in $\rho(P) \setminus Rej^R(M) \subseteq GS(M)$ such that $M \models B$ and $\ell(L) > \ell(B)$.

  In the former case, since $Default(M) \subseteq GS(M)$, $L \in least(GS(M))$ as desired. In the latter, by minimality of $\ell(L)$ it follows that τ is the fact L, and hence, since $\tau \in GS(M)$, $L \in least(GS(M))$ as desired.

  Inductive step. We suppose the thesis is proved for $\ell(L) \leq m$ and prove it for $\ell(L) = m+1$. Again by lemma 3.6.2 either $L \in Default(M)$ or there exists a rule $\tau : L \leftarrow B$ in $\rho(P) \setminus Rej^R(M) \subseteq GS(M)$ such that $M \models B$ and $\ell(L) > \ell(B)$.

In the former case, by definition of $GS(M)$, $L \in least(GS(M))$ as desired. In the latter, we know $\tau \in GS(M)$. Since $\ell(L) > \ell(B)$ and every literal in B is true in M, by inductive hypothesis, we conclude $least(GS(M)) \models B$, and hence $L \in GS(M)$. This concludes the proof.

♢

We have hence obtained an alternative characterization of the refined semantics for DyLPs. From Theorems 3.6.2 and 3.6.1, it is trivial to prove that the refined semantics for DyLPs coincides with the stable models semantics for normal logic program whenever the considered program $\mathcal{P}$ is a single normal logic program. We prove a stronger result saying that this is also the case if $\mathcal{P}$ is a generalized logic program.

Theorem 3.6.3 *Let $\mathcal{P}$ be a DyLP consisting of the single generalized logic program P over the language $\mathcal{L}$ and M be an interpretation over $\mathcal{L}$. M is a refined stable model of $\mathcal{P}$ iff M is a stable model of P.*

proof We have to prove that M is a refined stable model of $\mathcal{P}$ iff M is a stable model of P.

Let M be a refined stable model of $\mathcal{P}$. Theorem 3.5.1 states that M is also a dynamic stable model of $\mathcal{P}$. As proved in [Lei03], if $\mathcal{P}$ is the single generalized logic program P, than any dynamic stable model M is also a stable model of P.

Let now M be a stable model of $\mathcal{P}$. We prove that M is a well-supported model of $\mathcal{P}$ and obtain, by Theorem 3.9.1 that M is a refined model of $\mathcal{P}$. Let P' be the normal LP obtained from P by deleting all the rules whose head is a negative literals. It is proved in [Rei87], that M is stable model of P' (and hence it is a WS model of P') and that M is a model of P. Then, there exists a level mapping ℓ' satisfying conditions *ii*) of Definition 11.

Let further ℓ be the level mapping obtained from ℓ' in the following way. If $A \in M$, then $\ell(A) = \ell(A)' + 1$, otherwise $\ell(A) = 0$ The set of rejected rules w.r.t. ℓ is empty and hence $\rho(P) \setminus Rej_\ell(M) = P$. Since M is a model of P, condition *i*) of Definition 29 is satisfied.

regarding condition *ii*) of Definition 29, clearly ℓ satisfies condition *ii*) of Definition 11, i.e. $\forall\, A \in M\ \exists\, \tau \in P$ *such that* $hd(\tau) = A,\ \ell(A) > \ell(A_1), \ldots \ell(A_n)$, for each A_i, where the $A_i s$ and are the positive literals belonging to the body of τ and $M \models B(\tau)$. Let the $not\, B_j s$ be the negative literals in the $B(\tau)$. By definition of ℓ, we know $\ell(A) > \ell(not\, B_j)$ for each j. Hence $\ell(A) > \ell(B(\tau))$ and hence ℓ satisfied condition *ii*) of Definition 29. Hence M is a WS model of $\mathcal{P}$ according to Definition 29. ♢

We proved that the WS models coincide with the refined stable models of a DyLP. Since a refined model of a DyLP is also a model according to any other stable models-like semantics for DyLPs based on causal rejection we obtain the following result.

Corollary 3.6.1 *Let $\mathcal{P}$ be a dynamic stable model. If an interpretation M is a well-supported model of $\mathcal{P}$, it is also a model according to update programs, justified update, and dynamic stable models semantics.*

The notion of well-supported model clarifies the different (and counterintuitive) behavior of the considered semantics. Two distinguished semantics differ for those cases when one semantics is a better approximation of the semantics of well-supported models than the other one. Let us consider for instance example 3.1.4. The unique well-supported model (and hence the unique refined model) is $\{day\}$. The rule *not stars* is rejected by τ : *stars* $\leftarrow$ *stars*. Since, according to any level mapping, the level of the head of τ is equal to the level of its body, and hence τ can never reject any rule, it follows that $\emptyset$ is not a WS model.

3.7 Further properties of the refined semantics

In the following we address relevant notions and properties that have been used in other works (in particular [Lei03, EFST02a]) study other semantics for DyLPs based on causal rejection.

Surely, an important point is to establish a notion of equivalence between two DyLPs. A natural form of equivalence considered, for instance, in [EFST02a] is that of *semantic equivalence*. Two dynamic logic programs are equivalent iff they have the same set of models according to the considered semantics. Another, more strict notion of equivalence is that of *update equivalence* (see [Lei03] for further details) requiring the same set of models (according to the considered semantics) despite of any subsequent updates equal for both programs. This notion of equivalence is useful for studying possible simplifications, like deletion of rules preserving the meaning of a program. Formally:

Definition 30 *Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two DyLPs. Then $\mathcal{P}_1$ and $\mathcal{P}_2$ are* update equivalent *(denoted $\mathcal{P}_1 \equiv^{\oplus} \mathcal{P}_2$) according to the refined semantics iff for any (possibly empty) DyLP $\mathcal{Q}$:*

$$\mathcal{RS}(\mathcal{P}_1 \oplus \mathcal{Q}) = \mathcal{RS}(\mathcal{P}_2 \oplus \mathcal{Q})$$

By considering the special case of $\mathcal{Q} = \emptyset$ (i.e. $\mathcal{Q}$ is the empty sequence of programs) two update equivalent programs have the same set of refined models i.e. they are semantically equivalent according to the refined semantics.

Since relation $\equiv^{\oplus}$ is only based on equality between sets of interpretations, by the properties of equivalence of equality we immediately obtain that also $\equiv^{\oplus}$ is an equivalence relation.

Proposition 3.7.1 *Let $\mathcal{P}_1$,$\mathcal{P}_2$ and $\mathcal{P}_3$ be three DyLPs. Then:*

Reflexivity $\mathcal{P}_1 \equiv^{\oplus} \mathcal{P}_1$;

Symmetry *If* $\mathcal{P}_1 \equiv^{\oplus} \mathcal{P}_2$ *then* $\mathcal{P}_2 \equiv^{\oplus} \mathcal{P}_1$*;*

Transitivity *If* $\mathcal{P}_1 \equiv^{\oplus} \mathcal{P}_2$ *and* $\mathcal{P}_2 \equiv^{\oplus} \mathcal{P}_3$ *then* $\mathcal{P}_1 \equiv^{\oplus} \mathcal{P}_3$.

Another property that directly follows from the definition is that updating update equivalent programs with the same updates we still obtain update equivalent programs.

A first and fundamental property allowing simplification of programs is that is a rule $L \leftarrow B$ belongs to an update P_i, then any rule with either head L or *not* L with body B_2 such that $B_1 \subseteq B_2$ in any previous update can be removed.

Proposition 3.7.2 *Let* $\mathcal{P} : P_1, \ldots, P_n$ *be any DyLP and let* τ *be a rule in any* P_i*. If* γ *is a rule in any* P_j *with* $hd(\tau) = hd(\gamma)$ *or* $\tau \bowtie \gamma$ *and* $B(\tau) \subseteq B(\gamma)$ *then* $\mathcal{P} \equiv^{\oplus} \mathcal{P}'$ *where* $\mathcal{P}'$ *is the DyLP obtained by removing the rule* γ *from* P_i*.*

As a particular case, if a literal (i.e. a rule with empty body) belongs to some P_i we can remove any previous rule whose head is either the literal or its default negation.

Corollary 3.7.1 *Let* $\mathcal{P} : P_1, \ldots, P_n$ *be any DyLP and let* L *be a literal in any* P_i*. If* γ *is a rule in any* P_j *with* $L = hd(\gamma)$ *or* $L = not\,(hd(\gamma))$ *then* $\mathcal{P} \equiv^{\oplus} \mathcal{P}'$ *where* $\mathcal{P}'$ *is the DyLP obtained by removing the rule* γ *from* P_i*.*

A rule with negative head in an update P_i can be removed if there are no conflicting rules in P_i nor in any less recent update.

Proposition 3.7.3 *Let* $\mathcal{P} : P_1, \ldots, P_n$ *be any DyLP and let* $\tau : not\, A \leftarrow B$ *be a rule in any* P_i *with negative head. If there exists no rule* η *any* P_j *such that* $j \leq i$ *such that* $\tau \bowtie \eta$ *then* $\mathcal{P} \equiv^{\oplus} \mathcal{P}'$ *where* $\mathcal{P}'$ *is the DyLP obtained y removing the rule* τ *from* P_i*.*

It is also possible to move a set of rules K from one update P_i to another update P_j with $i \leq j$ or viceversa if there is no rule conflicting with a rule of K in any update P_z with $i \leq z \leq j$.

Proposition 3.7.4 *Let* $\mathcal{P} : P_1, \ldots, P_n$ *be any DyLP,* P_i *and* P_j *two updates of* $\mathcal{P}$ *with* $i < j$ *and let* K *be a subset of* P_j *such that there exist no rule* τ*, not belonging to* K*, in* P_z *with* $i \leq z \leq j$ *conflicting with a rule in* K*. Then* $\mathcal{P} \equiv^{\oplus} \mathcal{P}'$ *where* $\mathcal{P}'$ *is the DyLP obtained by removing the rule* τ *from* P_i*.*

proof It immediately follows from Definition 29 that any interpretation I is a well supported model of $\mathcal{P}$ iff it is a well supported model of $\mathcal{P}'$. ♢

Yet Another possible simplification is the possibility to remove empty updates.

Proposition 3.7.5 *Let $\mathcal{P}$, be a DyLP. Then:*

$$\mathcal{P} \oplus P_{\emptyset} \equiv^{\oplus} \mathcal{P}$$
$$P_{\emptyset} \oplus \mathcal{P} \equiv^{\oplus} \mathcal{P}$$
$$\mathcal{P} \oplus P_{\emptyset} \equiv^{\oplus} \mathcal{P}$$

proof It immediately follows from the Definition 29 since updates with no rules do not concur in any way to the definition of a well-supported model. ♢

The first two equivalences of Proposition 3.7.5 are referred in [EFST02a] as *Idempotence* and *Initialization* although the notion of equivalence used in the text is that of semantic equivalence (which is a weaker notion of equivalence as we have seen above).

Another simplification property is that repeated programs can be deleted, formally:

Proposition 3.7.6 *Let $\mathcal{Q}$ and $\mathcal{P}$ be two DyLPs. Then:*

$$\mathcal{P} \oplus \mathcal{P} \equiv^{\oplus} \mathcal{P}$$
$$\mathcal{Q} \oplus \mathcal{P} \oplus \mathcal{P} \equiv^{\oplus} \mathcal{Q} \oplus \mathcal{P}$$

proof It trivially follows from Definition 29 that any well-supported model of $\mathcal{P} \oplus \mathcal{P}$ ($\mathcal{Q} \oplus \mathcal{P} \oplus \mathcal{P}$) is a well-supported model of $\mathcal{P}$ ($\mathcal{Q} \oplus \mathcal{P}$) and viceversa. ♢

The first equivalence of Proposition 3.7.6 is referred in [EFST02a] as *Absorption*.

In the literature there are some attempts to define properties and postulates that should be satisfied by a semantics for logic programs updates, see for instance [EFST02a, KM91, MC07]. We have seen some properties satisfied by the refined semantics that were illustrated in [EFST02a]. We will come back on this general issue in Section 3.13.

We would like to discuss a further property enounced in [EFST02a] which is *not* satisfied in the refined semantics: that of *minimality of changes*. This property can be generally enounced as follows: the models admitted in the semantics are only those that minimize the set of rejected rules. Informally, if we consider a semantics that satisfies this property, given two models M_1 and M_2 of a DyLP $\mathcal{P}$ the rules of $\mathcal{P}$ that are rejected according to M_1 cannot be a subset of the rules rejected according to M_2 and viceversa. The principle behind this property is that of incorporating a new update P into an existing program $\mathcal{P}$ with as little changes as possible. As a particular case, if $P_1 \cup P_2$ has at least one stable model, then the models of P_1, P_2 should be a subset of the models of $P_1 \cup P_2$, since the models of $P_1 \cup P_2$ do not require any rejection of rules to be allowed.

Let us consider again example 3.1.2 where the situation is that of a restaurant that is initially open every day and later not open during the holidays (a day can be either

a working day or a holiday). The DyLP of this example is:

$$\begin{array}{ll} P_1: & open(restaurant). \\ P_2: & not\, open(restaurant) \leftarrow holiday. \\ & workday \leftarrow not\, holiday. \\ & holiday \leftarrow not\, workday. \end{array}$$

This DyLP has two refined models (and hence two models according to any semantics based on causal rejection).

$$M_1 = \{open(restaurant), workday\} \quad M_2 = \{holiday\}$$

The result formalizes the intuition that the restaurant is open during the working days and not open during the holidays. Indeed, regarding M_1:

$$\begin{array}{rcl} Rej^R(\mathcal{P}, M_1) & = & \{\} \\ Def(\mathcal{P}, M_1) & = & \{not\, holiday\} \\ \Gamma^R(\mathcal{P}, M_1) & = & least\left((P_1 \cup P_2) \setminus Rej^R(\mathcal{P}, M_1) \cup Def(\mathcal{P}, M_1)\right) = \\ & = & \{open(restaurant), workday, not\, holiday\} \end{array}$$

Regarding M_2:

$$\begin{array}{rcl} Rej^R(\mathcal{P}, M_2) & = & \{open(restaurant)\} \\ Def(\mathcal{P}, M_2) & = & \{not\, workday\} \\ \Gamma^R(\mathcal{P}, M_2) & = & least\left((P_1 \cup P_2) \setminus Rej^R(\mathcal{P}, M_2) \cup Def(\mathcal{P}, M_2)\right) = \\ & = & \{not\, open(restaurant), holiday, not\, workday\} \end{array}$$

We see that the set of rejected rules according to M_1 and hence this set is a subset of the rejected rules according to M_2. The property of minimality of changes is then violated in this example. Since every refined model is also a model according to any of the semantics based on causal rejection, the example above also implies that minimality of changes is not satisfied by any of these semantics.

In [EFST02a] the authors define a refinement of the justified updates semantics called *minimal updates* that satisfies the property of minimality of changes. This is done by selecting those models for which the set of rejected rules is minimal. It would be clearly possible to apply this idea also to the refined semantics, According to minimality of changes, the only admitted model should be M_1. The reader may notice that this conclusions is counterintuitive in this example, since it excludes the possibility that a day is a holiday. The reason is that the update P_2 encodes the *exception* to the initial rules given by the changes. In this perspective, in our opinion, minimality of changes is not always required. Moreover, it is proved in [EFST02a] that the computational com-

plexity of determining whether a predicate is true in any minimal update model of a DyLP is a $\Pi_2^P - complete$ problem while the same problem is $co-NP$-complete in the refined semantics (see Theorem 3.11.3). These differences in the results and complexity suggest that the two approaches are targeting different applications.

3.8 Refined well-supported semantics for multidimensional dynamic logic programs

As already said MDyLPs represent an extension of DyLPs. While DyLPs are sequences of logic programs, MDyLPs are partially ordered multisets of programs. The notion of well-supported model we have used for clarifying which semantics for DyLPs can be straightforwardly extended to the multidimensional case.

Definition 31 *Let $\mathcal{MP}$ be any MDyLP over a language $\mathcal{L}$, P_n an update of $\mathcal{MP}$, ℓ any level mapping over $\mathcal{L}$ and M any two-valued interpretation over $\mathcal{L}$, we define the set of rejected rules w.r.t. ℓ as*[7]*:*

$$Rej_\ell(\mathcal{MP}, M, n) = \{\tau \in P_i : \exists \eta \in P_j, i \prec j \preceq n, \tau \bowtie \eta, M \models B(\eta), \ell(hd(\eta)) > \ell(B(\eta))\}$$

Definition 32 (Well-supported semantics for MDyLPs) *Let $\mathcal{MP}$ be any multidimensional dynamic logic program in the language $\mathcal{L}$ and M interpretation over $\mathcal{L}$, P_n an update of $\mathcal{M}$ and M any interpretation over $\mathcal{L}$. We say M say M is a **well supported model of** $\mathcal{MP}$ **at** P_n iff there exists a level mapping ℓ over $\mathcal{L}$ such that*

i) *M is a model of $\rho(P) \setminus Rej_\ell(M, n)$*

ii) *$\forall A \in M \; \exists \tau \in \rho(P) \setminus Rej_\ell(M, n)$ such that $hd(\tau) = A$, $\ell(A) > \ell(B(\tau))$ and τ is supported by M.*

This way, we obtain a descriptive characterization of a semantics for MDyLPs. A first question is whether this characterization coincides with either the update programs or the dynamic stable models semantics for MDyLPs. The answer is clearly negative, since these semantics are generalization of semantics for DyLPs whose models do not always coincide with the well-supported models.

Moreover, it is important to notice that class of programs where these semantics do not coincide with the well-supported model characterization is quite more general then in the case of *DyLPs*.

Let us consider again the MDyLP $\mathcal{MP}$ of example 3.1.8. This program has no well-supported model at P_3 and hence we conclude the program is not consistent. This seems to be a good conclusion, since the information coming from the two sensors is

[7]Hereafter we use the simplified notation $Rej_\ell(M, n)$ instead of $Rej_\ell(\mathcal{MP}, M, n)$ whenever this causes no ambiguity.

clearly contradictory and there is no way to decide which source of information should be trusted. What most likely happened is that one of the two sensors is broken. The well-supported characterization detects this anomaly by saying the program has no well-supported model.

If, instead, we compute the update programs and the dynamic stable models semantics of $\mathcal{MP}$ at P_3 we find that the program has one model, according to these semantics, precisely the program $\{water(p(1)), \ldots, water(p(10))\}$. What happened is that the cycle in P_3 has removed the contradiction of P_1 and P_3 by rejecting $not\, water(p(5))$.

In general, when information from several sources is merged without assigning a total order among such sources, if potential contradictions are present the other semantics for MDyLPs may admit undesired models.

3.9 Fixpoint characterization for well-supported models of MDyLPs

Well-supported models define a semantics for MDyLPs. However, this characterization is purely descriptive, which is obviously not entirely satisfactory for computational purposes. Moreover, to understand, from this definition, whether a given interpretation is a WS model of a given MDyLP, we have to face the problem of finding a corresponding level mapping or show that such a level mapping does not exist. This is not a reasonable approach for computing the semantics, and furthermore may not lead to quick ways of testing whether a given interpretation is a well-supported model or not. For these reasons, we present an alternative characterization based on a fixpoint operator. We characterize our models as the fixpoints of an operator defined from interpretations to interpretations, in the spirit of the Gelfond-Lifschitz operator [GL88].

Before providing a fixpoint characterization for the well-supported semantics we illustrate by examples how the dynamic stable model semantics of Definition 32 differs from well-supported models on a wider class of programs and the failure of naive attempts to generalize the Definition 27 that justifies the necessity of a more sophisticated approach.

Example 3.9.1 *Let $\mathcal{MP}$ be the MDyLP: formed by the programs: P_a, P_b, P_t with the following rules:*

$$P_a : A. \quad P_b : not\, A. \quad P_t : A \leftarrow A.$$

and the following partial order relation:

$$P_a \prec P_t \qquad P_b \prec P_t$$

There exists no well-supported model of $\mathcal{MP}$ at P_t, since there can be no well-supported rejection and the union of P_a and P_b contains a contradiction. Nevertheless $M = \{A\}$ is a multidimen-

sional dynamic stable model of $\mathcal{MP}$ at P_t. Indeed, according to Definition 22 we have:

$$\begin{aligned} Default(\mathcal{MP}^t, M) &= \{\} \\ Rej(\mathcal{MP}^t, M) &= \{not\, A\} \\ \Gamma^D_{(\mathcal{MP},n)}(M) &= least(\rho(\mathcal{MP}^t) \setminus Rej(\mathcal{MP}^t, M) \cup Default(\mathcal{MP}^t, M)) = \\ least(\{A,\ A \leftarrow A\}) &= \{A\} = M \end{aligned}$$

which implies, according to Definition 32 that M is a dynamic stable model of $\mathcal{MP}$ at P_t.

Example 3.9.1 illustrates that examples of MDyLPs for which the dynamic stable model semantics differs from the well-supported span over the class of MDyLPs with conflict among rules from *not comparable programs*, i.e. programs with no mutual precedence order.

This is a generalization to the linear case, where counterexamples belong to the class of DyLPs with conflicting rules in the same program. In the linear case the behavior was corrected by the refined semantics by allowing mutual rejection among rules in the same program. Hence, to correct the illustrated behavior, the naive attempt for extending the refined semantics to MDyLPs would be to also allow mutual rejection among rules belonging to non comparable programs formally:

Definition 33 (Naive refined multi stable model) *Let $\mathcal{MP}$ be any multidimensional dynamic logic program in the language $\mathcal{L}$, P_n and element of $\mathcal{MP}$ and M any two-valued interpretation over $\mathcal{L}$. The Γ^D operator is defined as follows.*

$$\Gamma^{LF}_{(\mathcal{P},n)}(M) = least(\rho(\mathcal{MP}^n) \setminus Rej^F(\mathcal{MP}^n, M) \cup Default(\mathcal{MP}^n, I))$$

Where $Rej(\mathcal{MP}^n, M)$ is defined as follows

$$Rej^F(\mathcal{MP}^n, M) = \{\tau \in P_i : \exists \eta \in P_j, j \not< i\ \ \tau \bowtie \eta,\ M \models B(\eta)\}$$

We say M is a naive-refined multi stable model *of $\mathcal{MP}$ at P_n iff:*

$$M = \Gamma^D_{(\mathcal{MP},n)}(M)$$

However, this semantics (that we called the naive-refined semantics for MDyLPs) in *not equivalent* to the well-supported semantics as shown by the following example [8].

Example 3.9.2 *Let $\mathcal{MP}$ be the MDyLP: formed by the programs: P_a, P_b, P_{a1}, P_{a2},$P_\emptyset$ with the following rules:*

$$P_a : A.\quad P_b : A.\quad P_{a1} : not\, A.\quad P_{b1} : not\, A.\quad P_\emptyset :$$

[8] We acknowledge Joao Leite for this example.

and the following partial order relation:

$$P_a \prec P_{a1} \quad P_b \prec P_{b1} \quad P_{a1} \prec P_\emptyset \quad P_{b1} \prec P_\emptyset$$

There interpretation $M = \{not\, A\}$ is the unique well-supported model of $\mathcal{MP}$ at $P_\emptyset$, since the facts A in P_a and P_b are rejected by the facts $not\, A$ in P_{a1} and P_{b1}

Nevertheless M is not a naive-refined multi stable model $\mathcal{MP}$ at $P_\emptyset$. Indeed, according to Definition 33 we have:

$$\begin{aligned} Default(\mathcal{MP}^\emptyset, M) &= \{\} \\ Rej^F(\mathcal{MP}^\emptyset, M) &= \{A, \quad A, not\, A, \quad not\, A\} \\ \Gamma^D_{(\mathcal{MP},n)}(M) &= least(\rho(\mathcal{MP}^\emptyset) \setminus Rej^R(\mathcal{MP}^\emptyset, M) \cup Default(\mathcal{MP}^\emptyset, M)) = \\ least(\{\}) &= \emptyset \neq M \end{aligned}$$

which implies, that M is a well-supported model but not a naive-refined multi stable of $\mathcal{MP}$ at $P_\emptyset$.

The failure of the naive approach for a fixpoint characterization of well-supported models of MDyLPs turned our research towards a more sophisticated approach. To better understand the relation of the well-supported semantics we also provide an alternative formulation of the dynamic multi stable model semantics of Definition 23.

As in the existing stable models-like semantics for MDyLPs, we define our models as the fixpoints of an operator defined from interpretations to interpretations, somehow similar to the Gelfond-Lifschitz operator [GL88]. Given a program $\mathcal{MP}$ and an update P_n with index n, we associate to the pair $(\mathcal{MP}, n)$, two distinct operators, namely $\Gamma_{(\mathcal{MP},n)}$ and $\Gamma^R_{(\mathcal{MP},n)}$ of Definition 36. We obtain an alternative characterization of well-supported models as the fixpoints of the Γ^R operator of Definition 36. We call such semantic *the refined semantics for MDyLPs*. Regarding the operator of Definition 36, we will in lemma 3.10.1 that this operators coincides with the operator Γ^D of Definition 22. Hence we obtain an alternative definition of the dynamic stable models semantics for MDyLPs as the fixpoints of the Γ operator of Definition 36. This alternative definition will be useful in Section 3.10 for comparing the refined and the dynamic stable models semantics for MDyLPs.

The semantics of a MDyLP $\mathcal{MP}$ is given with respect to an update P_n of $\mathcal{MP}$. To establish the semantics we consider just the genealogy of P_n i.e. the updates P_i such that $P_i \preceq P_n$. To take this into account it is sufficient, as we show in Definition 36, to apply the following definitions to the program $\mathcal{MP}^n$ instead of $\mathcal{MP}$

Let $\mathcal{MP}$ be a multidimensional DyLP over $\mathcal{L}$, P_j, P_k be programs of $\mathcal{MP}$ and $j \prec k$. If there exists a rule γ in P_k with head L, then every rule in P_j with head $not\, L$ could be rejected, depending whether the body of γ is true or not. We formalize this concept with a logic program in an extended language.

Definition 34 *Let $\mathcal{MP}$ be any multidimensional dynamic logic program in the language $\mathcal{L}$. Let the atoms of the form $rej(L,i)$, where L is a literal in $\mathcal{L}$ and i ranges over the indexes of the updates of $\mathcal{MP}$, be new atoms not belonging to $\mathcal{L}$. The set of rejecting rules over an extended language is defined as follows.*

$$Rj(\mathcal{MP}) = \{rej(not\, L, j) \leftarrow B. \mid L \leftarrow B. \in P_k \ \wedge \ j \prec k\}$$

Let $\mathcal{MP}$ be a multidimensional DyLP over $\mathcal{L}$, n an index, L a literal of $\mathcal{L}$, M an interpretation over $\mathcal{L}$, P_i, P_j and P_k be programs of $\mathcal{MP}^n$ and τ^i and η^j rules in, respectively P_i and P_j. We say η^j *is a threat for L in i*, or alternatively that *L is threatened by* η^j iff the head of η^j is *not L*, its body is true in M, and $j \not\prec i$. We say η^j is a threat for another rule τ^i in P_i iff it is a threat for its head in i.

It is clearly possible that two rules threaten each other, this happens whenever these two rules belongs to the same update or to non comparable updates and their bodies are both true in M. A literal (rule) is considered *safe*, if all its threats in more recent updates are rejected. A literal (rule) is considered *strictly safe* in P_i iff all its threats are rejected. Intuitively only safe rules should be allowed to derive conclusions. The notion of *safeness* we adopt determine the semantics for MDyLPs. The main idea behind our definition is that a rule can be used to derive consequences iff it has been already established that such rule is safe. To achieve this result, we first consider the GLP given by the union of all the rules in $\mathcal{MP}$ with a new atom in the body of each rule which is satisfied only if the considered rule is safe. Then we introduce rules specifying which threatens should be rejected in order to consider a literal (rule) as safe. Finally we introduce rules determining when a threat is rejected.

Definition 35 *Let $\mathcal{MP}$ be any multidimensional dynamic logic program in the language $\mathcal{L}$. Let the atoms of the form $safe(L,i)$, where L is a literal in $\mathcal{L}$ and i ranges over the indexes of the updates of $\mathcal{MP}$, be new atoms not belonging to $\mathcal{L}$. We denote by $\Sigma(\mathcal{MP})$ the following set of rules.*

$$\Sigma(\mathcal{MP}) = \{L \leftarrow B, safe(L,i) | L \leftarrow B \in P_i, \}$$

Let M be an interpretation over $\mathcal{L}$. The set of conditions for a literal L to be safe (resp. strictly safe) at P_i are defined as follows.

$$\begin{aligned} cond(\mathcal{MP},M,L,i) &= \{rej(not\, L, j) : \ i \prec j : \exists \eta \in P_j \mid M \models B(\eta), \ \wedge hd(\eta) = not\, L\} \\ cond^R(\mathcal{MP},M,L,i) &= \{rej(not\, L, j) : \ j \not\prec i : \exists \eta \in P_j \mid M \models B(\eta) \ \wedge hd(\eta) = not\, L\} \end{aligned}$$

The set of condition rules *and* strictly condition rules *are defined as follows*

$$\begin{aligned} Safe(\mathcal{MP},M)) &= \{safe(L,i) \leftarrow cond(\mathcal{MP},M,L,i) | \ \exists\, \tau \in P_i : \ hd(\tau) = L\} \\ Safe^R(\mathcal{MP},M)) &= \{safe(L,i) \leftarrow cond^R(\mathcal{MP},M,L,i) | \ \exists\, \tau \in P_i : \ hd(\tau) = L\} \end{aligned}$$

In the following we write $cond^R(L,i)$ and $cond^R(L,i)$ instead of $cond^R(\mathcal{MP},M,L,i)$ and

$cond(\mathcal{MP}, M, L, i)$ whenever $\mathcal{MP}$ and M are clear from the context. Clearly the following inclusion always holds.

$$cond(L,i) \quad \subseteq \quad cond^R(L,i) \tag{3.8}$$

Having specified the conditions for a rule to be allowed to derive literals, both by the introduction of extra literals and rules, we add the set of default assumptions and compute the least model of the obtained program. Note that the safe rules and the rejected ones are determined along with the computation of the model. Finally we discard the auxiliary literals computed, and obtain the resulting interpretation.

Definition 36 *Let $\mathcal{MP}$ be any multidimensional dynamic logic program in the language $\mathcal{L}$, P_n an update of $\mathcal{MP}$ and M any interpretation over $\mathcal{L}$. We We define the operators $\Gamma_{(\mathcal{MP},n)}$ and $\Gamma^R_{(\mathcal{MP},n)}$ from interpretations over $\mathcal{L}$ to interpretations over $\mathcal{L}$ in the following way:*

$$\begin{aligned}
\Gamma_{(\mathcal{MP},n)}(I) &= least(\Sigma(\mathcal{MP}^n) \cup Safe(\mathcal{MP}^n, I) \cup Rj(\mathcal{MP}^n) \cup Default(\mathcal{MP}^n, I))|_{\mathcal{L}} \\
\Gamma^R_{(\mathcal{MP},n)}(I) &= least(\Sigma(\mathcal{MP}^n) \cup Safe^R(\mathcal{MP}^n, I) \cup Rj(\mathcal{MP}^n) \cup Default(\mathcal{MP}^n, I))|_{\mathcal{L}}
\end{aligned}$$

In the remainder we will use the notation Γ and Γ^R whenever $\mathcal{MP}$ and n are clear from the context.

We are finally ready to define the refined semantics for MDyLPs. A *refined multi stable model*(RMSM) of a MDyLP at P_n is any fixpoint of the $\Gamma^R_{(\mathcal{MP},n)}$ operator.

Definition 37 (Refined Semantics for MDyLPs) *Let $\mathcal{MP}$ be any multidimensional dynamic logic program in the language $\mathcal{L}$, P_n and element of $\mathcal{MP}$ and M any interpretation over $\mathcal{L}$. We say M is a refined multi stable model of $\mathcal{MP}$ at P_n iff:*

$$M = \Gamma^R_{(\mathcal{MP},n)}(M)$$

The semantics of Definition 37 corresponds with the the well-supported semantics of Definition 32 as shown by the following Theorem.

Theorem 3.9.1 *Let $\mathcal{MP}$ be any multidimensional dynamic logic program in the language $\mathcal{L}$, n an index and M any two-valued interpretation over $\mathcal{L}$. Then, M is a* refined multi stable model *of $\mathcal{MP}$ at P_n iff M is a* multi update well supported model *of $\mathcal{MP}$ at P_n.*

3.10 Relationship to other semantics for MDyLPs

As an immediate consequence of Theorem 3.9.1 we obtain that the refined multi stable models semantics of Definition 37 is an extension of the refined stable models semantics defined for the linear case.

Proposition 3.10.1 *Let $\mathcal{MP}$ be a MDyLP consisting of m elements and let the partial order $\prec$ defined over $\mathcal{MP}$ be a total order. For each natural number i, $i \leq m$, let P_i be the i^{th} element of $\mathcal{MP}$. Then for each $n \leq m$ an interpretation M is a multi refined stable model of $\mathcal{MP}$ iff it is a refined stable model of $P_1, \ldots, P_n$.*

proof The proof is immediate. By Theorem 3.9.1 M is a multi refined stable model at P_n iff it is a well-supported model of $\mathcal{MP}^n$. Since $\prec$ is a total order, this is equivalent to say that M is a well-supported model of $P_1, \ldots, P_n$. By Theorem 3.6.2 this is equivalent to say that M is a refined stable model of $P_1, \ldots, P_n$. $\diamondsuit$

We want further to establish a relationship between the refined semantics and the existing semantics for MDyLPs. First of all we prove that the dynamic stable models semantics for MDyLPs can be defined in terms of the Γ operator of Definition 36. To prove this, we need to prove first that the Γ^D operator of Definition 22 and the Γ operator of Definition 36 are the same operator defined in two different ways.

Lemma 3.10.1 *Let $\mathcal{MP}$ be any multidimensional dynamic logic program in the language $\mathcal{L}$, P_n an update of $\mathcal{M}ulti$ and I any set of literals over $\mathcal{L}$ Then:*

$$\Gamma_{(\mathcal{MP},n)}(I) = \Gamma^D_{(\mathcal{MP},n)}(I)$$

An alternative definition of the dynamic stable models semantics follows as a corollary of Lemma 3.10.1

Theorem 3.10.1 *Let $\mathcal{MP}$ be any multidimensional dynamic logic program in the language $\mathcal{L}$, P_n an update of $\mathcal{M}ulti$ and M any two-valued interpretation over $\mathcal{L}$. Then M is a multi dynamic stable model as defined in [Lei03] iff*

$$M = \Gamma_{(\mathcal{MP},n)}(M)$$

Since the dynamic stable models and the refined semantics for MDyLPs are extensions, respectively, of the dynamic stable models and the refined semantics for DyLPs, we would expect that, also in the multidimensional case, any refined stable model of a given program is also a dynamic stable model. Indeed, this result holds, but to prove this we first need one further technical result.

Lemma 3.10.2 *Let $\mathcal{MP}$ be any multidimensional dynamic logic program in the language $\mathcal{L}$, P_n an update of $\mathcal{M}ulti$ and I any interpretation over $\mathcal{L}$. Then*

$$\Gamma_{(\mathcal{MP},n)}(I) \subseteq \Gamma^R_{(\mathcal{MP},n)}(I)$$

proof We know that

$$\Gamma_{(\mathcal{MP},n)}(I) = least(\Sigma(\mathcal{MP}^n) \cup Safe(\mathcal{MP}^n, I) \cup Rj(\mathcal{MP}^n) \cup Default(\mathcal{MP}^n, I))|_{\mathcal{L}}$$
$$\Gamma^R_{(\mathcal{MP},n)}(I) = least(\Sigma(\mathcal{MP}^n) \cup Safe^R(\mathcal{MP}^n, I) \cup Rj(\mathcal{MP}^n) \cup Default(\mathcal{MP}^n, I))|_{\mathcal{L}}$$

Hence, it is sufficient to prove

$$R = least(\Sigma(\mathcal{MP}^n) \cup Safe^R(\mathcal{MP}^n, I) \cup Rj(\mathcal{MP}^n) \cup Default(\mathcal{MP}^n, I)) \subseteq$$
$$H = least(\Sigma(\mathcal{MP}^n) \cup Safe(\mathcal{MP}^n, I) \cup Rj(\mathcal{MP}^n) \cup Default(\mathcal{MP}^n, I))$$

By definition, H is a model of $\Sigma(\mathcal{MP}^n)$, $Rj(\mathcal{MP}^n)$, and $Default(\mathcal{MP}^n, I)$. We prove that H is also a model of $Safe^R(\mathcal{MP}^n, I)$. Let $r : safe(L,i) \leftarrow cond^R(L,i,M)$ be a rule of $Safe^R(\mathcal{MP}^n, I)$. Then the rule $r' : safe(L,i) \leftarrow cond(L,i,M)$ belongs to $Safe(\mathcal{MP}^n, I)$. If $safe(L,i)$ belongs to H, then H is a clearly model of r. Otherwise , if $safe(L,i) \notin H$, then, since H is a model of r', $cond(L,i)$ is false. By inclusion 3.8, this implies $cond^R(L,i)$ is also false and hence H is a model of r. Since r can be any rule in $Safe^R(\mathcal{MP}^n, I)$ we conclude $H \models Safe^R(\mathcal{MP}^n, I)$. Hence H is a model of

$$GS(M) = \Sigma(\mathcal{MP}^n) \cup Safe^R(\mathcal{MP}^n, I) \cup Rj(\mathcal{MP}^n) \cup Default(\mathcal{MP}^n, I)$$

Since R is the least model of $GS(M)$ we conclude $R \subseteq H$ and this concludes the proof $\diamondsuit$

We are now ready to prove that the refined stable semantics for MDyLPs is indeed a refinement of the dynamic semantics.

Theorem 3.10.2 *Let $\mathcal{MP}$ be any multidimensional dynamic logic program in the language $\mathcal{L}$, P_n an update of $\mathcal{M}ulti$ and M a refined multi stable model of $\mathcal{MP}$ at P_n. Then*

$$M = \Gamma_{(\mathcal{MP},n)}(M)$$

proof We have to prove the two inclusions.

$$a) \quad M \subseteq \Gamma_{(\mathcal{MP},n)}(M) \qquad\qquad b) \quad M \supseteq \Gamma_{(\mathcal{MP},n)}(M)$$

- The proof of a) is trivial. By definition, $M = \Gamma^R_{(\mathcal{MP},n)}(M)$. By lemma 3.10.2 we obtain $M = \Gamma^R_{(\mathcal{MP},n)}(M) \subseteq \Gamma_{(\mathcal{MP},n)}(M)$
- It remains to prove b). Let $G(M)$ be the program

$$\Sigma(\mathcal{MP}^n) \cup Safe(\mathcal{MP}^n, I) \cup Rj(\mathcal{MP}^n) \cup Default(\mathcal{MP}^n, I)$$

  Let T_G be the immediate consequences operator related to the program $G(M)$ and T_G^m the m^{th} application of T_G to the empty set. Since $\Gamma_{(\mathcal{MP},n)}(M) = \bigcup T_G^m|_{\mathcal{L}} m$, it is sufficient to prove that, for each natural number m, $T_G^m|_{\mathcal{L}} m \subseteq M$. We proceed by induction on m

  Basic step. $T_G^0 = \emptyset$, hence the thesis is trivially satisfied

  Inductive step. Let us suppose $T_G^m|_{\mathcal{L}} \subseteq M$, and prove that $T_G^{m+1}|_{\mathcal{L}} \subseteq M$. If $L \in Default(M)$ then, since M is a refined multi stable model, then $L \in M$.

Otherwise there exists a rule $\tau^i : L \leftarrow B$ in some update P_i, such that $L \leftarrow B, safe(L,i) \in G(M)$, $T_G^m \models B$ (and hence, by inductive hypothesis, $M \models B$) and $safe(L,i) \in T_G^m$. Let us assume that i is the maximum index for which such a rule belongs to P_i.

By contradiction Let us suppose $L \notin M$, then , by Theorem 3.9.1, $\tau^i \in Rej_\ell(M)$. This means there is a rule η^j with head $not\, L$ and true body in M, in some P_j with $i \prec j$. Then η^j threatens τ^i. Since $safe(L,i) \in T_G^m$, then $rej(not\, L, j) \in T_G^m$ i.e. there exists a rule $rej(L,j) \leftarrow B'$ in $G(M)$ which means, there exists a rule $L \leftarrow B', safe(L,k)$ in $G(M)$ for some $k \succ i$. Since $i \prec k$, then $cond(L,j,M) \subseteq cond(L,k,M)$. Hence $safe(L,k) \in T_G^m$. This is inconsistent with the hypothesis of maximality of i.

Hence $L \in M$ as desired. Then $\Gamma_{(\mathcal{MP},n)}(M) \subseteq M$ as desired and this concludes the proof.

$\diamondsuit$

It is clear now where the matrix of the counterintuitive behavior of the dynamic stable models semantics in example 3.1.8 lies. The adopted concept of safeness is too weak. Indeed, the rule $water(p(5))$ in the update P_1 is considered as safe even if it is threatened by the rule $not\, water(p(5))$ and this threat is not defeated.

Since a dynamic multi stable model is also a model according to the semantics define in [BFL99, LP97], by Theorem 3.10.2 it follows that a refined stable model of a given program is also a model according to any of the existing semantics for MDyLPs based on causal rejection.

Theorem 3.10.3 *Let $\mathcal{MP}$ be any MDyLPs in the language $\mathcal{L}$, P_n be an update of $\mathcal{MP}$ and M be a refined multi stable model of $\mathcal{MP}$ at P_n. Then M is also a model of $\mathcal{MP}$ in any of the semantics for MDyLPs defined in [LAP01, BFL99].*

3.11 A program transformation for the refined semantics of DyLPs

The refined transformation defined in this Section turns a DyLP $\mathcal{P}$ in the language $\mathcal{L}$ into a normal logic program $\mathcal{P}^T$ (called the *refined transformational equivalent of* $\mathcal{P}$) in an extended language. We provide herein a formal procedure to obtain the transformational equivalent of a given DyLP.

Let $\mathcal{L}$ be a language. By $\mathcal{L}^T$ we denote the language whose elements are either atoms of $\mathcal{L}$, or atoms of one of the following forms: u, A^-, $rej(A,i)$, $rej(A^-,i)$, where i is a natural number, A is any atom of $\mathcal{L}$ and none of the atoms above belongs to $\mathcal{L}$. Intuitively, A^- stands for "A is false", while $rej(A,i)$ (resp. $rej(A^-,i)$), stands for: "all

the rules with head A (resp. *not* A) in the update P_i are rejected". For every literal L, if L is an atom A, then $\overline{L}$ denotes A itself, while if L is a negative literal *not* A then $\overline{L}$ denotes A^-. For extension, given a set of literals $\mathcal{K}$, by $\overline{\mathcal{K}}$ we denote the following set of atoms:

$$\overline{\mathcal{K}} = \{\overline{L} : L \in \mathcal{K}\}$$

Given conjunction of literals $B = L_1, \ldots L_n$, by $\overline{B}$ we denote the conjunction $\overline{L_1}, \ldots, \overline{L_n}$. Given a rule $\tau : L \leftarrow B$, by $\overline{\tau}$ we denote the rule $\overline{L} \leftarrow \overline{B}$ and given a program P, by $\overline{P}$ we denote the following set of rules:

$$\overline{P} = \{\overline{\tau} : \tau \in P\}$$

Finally, u is a new atom not belonging to $\mathcal{L}$ which is used for expressing integrity constraints of the form $u \leftarrow not\ u, L_1, \ldots, L_k$ (which have the effect of removing all stable models containing $L_1, \ldots, L_k$).

Definition 38 *Let $\mathcal{P}$ be a Dynamic Logic Program whose language is $\mathcal{L}$. By the refined transformational equivalent of $\mathcal{P}$, denoted $\mathcal{P}^T$, we mean the normal program $P_1^T \cup \ldots \cup P_n^T$ in the extended language $\mathcal{L}^T$, where each P_i^R exactly consists of the following rules:*

Default assumptions *For each atom A of $\mathcal{L}$ appearing in P_i, and not appearing in any other P_j, $j \leq i$ a rule:*

$$A^- \leftarrow not\ rej(A^-, 0)$$

Rewritten rules *For each rule $L \leftarrow B$ in P_i, a rule:*

$$\overline{L} \leftarrow \overline{B},\ not\ rej(\overline{L}, i)$$

Rejection rules *For each rule $L \leftarrow B$ in P_i, a rule:*

$$rej(\overline{not\ L}, j) \leftarrow \overline{B}$$

where $j \leq i$ is the largest index such that P_j has a rule with head not L. If no such P_j exists, and L is a positive literals, then $j = 0$, otherwise this rule is not part of P_i^R. Moreover, for each rule $L \leftarrow B_1$ in P_i, a rule:

$$rej(\overline{L}, j) \leftarrow rej(\overline{L}, i)$$

where $j < i$ is the largest index such that P_j also contains a rule $L \leftarrow B_2$. If no such P_j exists, and L is a negative literal, then $j = 0$, otherwise this rule is not part of P_i^R.

Totality constraints *For each atom A in $\mathcal{L}$occurring in a rule of P^i, the constraint:*

$$u \leftarrow not\ u,\ not\ A,\ not\ A^-$$

Unless the constraint above already appears in some other P_k^T with $k < j$.

Let us briefly explain the intuition and the role of each of these rules. The *default assumptions* specify that a literal of the form A^- is true (i.e. A is false) unless this initial assumption is rejected. The *rewritten rules* are basically the original rules of the sequence of programs with an extra condition in their body that specifies that in order to derive conclusions, the considered rule must not be rejected. Note that, both in the head and in the body of a rule, the negative literals of the form *not A* are replaced by the corresponding atoms of the form A^-. The role of *rejection rules* is to specify whether the rules with a given head in a given state are rejected or not. Such a rule may have two possible forms. Let $L \leftarrow B$ be a rule in P_i. The rule of the form $rej(\overline{L}, j) \leftarrow \overline{B}$ specifies that all the rules with head *not L* in the most recent update P_j with $j \leq i$ must be rejected. The rules of the form $rej(\overline{L}, j) \leftarrow rej(\overline{L}, i)$ "propagate" the rejection to the updates below P_j. Finally, *totality constraints* assure that, for each literal A, at least one of the atoms A, A^- belongs to the model. This s done to guarantee that the models of transformed program are indeed two-valued.

The role of the atoms of the extended language $\mathcal{L}^T$ that do not belong to the original language $\mathcal{L}$ is merely auxiliary, as we see from the following Theorem. Let $\mathcal{P}$ be any Dynamic Logic Program and P_k and update of $\mathcal{P}$. Remember that a refined model of $\mathcal{P}$ at P_k is a refined model of $\mathcal{P}^k$ (see Section 3.2 for further details). We use the notation $\mathcal{P}^{Tk}$ for $P_0^R \cup \ldots \cup P_k^R$.

Theorem 3.11.1 *Let $\mathcal{P}$ be any Dynamic Logic Program in the language $\mathcal{L}$, P_i an update of $\mathcal{P}$, and let $\mathcal{P}^{Ti}$ be as above. Let M be any interpretation over $\mathcal{L}$. Then M is a refined stable model of $\mathcal{P}$ at P_i iff there exists a two-valued interpretation M^T such that M^T is a stable model of $\mathcal{P}^{Tk}$ and $M \equiv_{\mathcal{L}} M^T$. Moreover, M and M^T satisfy the following conditions (3.11.1):*

$$\begin{array}{lcl} A \in M \Leftrightarrow A \in M^T & & not\ A \in M \Leftrightarrow A^- \in M^T \\ not\ A \in Default(\mathcal{P}^i, M)) & \Leftrightarrow & rej(A^-, 0) \notin M^T \\ \tau \in rej^R(M, \mathcal{P}^i) \wedge \tau \in P_i & \Leftrightarrow & rej(\overline{hd(\tau)}, i) \in M^T \end{array}$$

For illustration, we present an example of the computation of the refined transformational equivalent of a DyLP.

Example 3.11.1 *Let $\mathcal{P}: P_1, P_2$ be the following DyLP:*

$$\begin{array}{ll} P_1: & a \leftarrow b. \\ P_2: & b. \quad c. \\ P_3: & not\ a \leftarrow c. \end{array}$$

The transformational equivalent of $\mathcal{P}$ is the following sequence P_1^R, P_1^R, P_2^R:

$$
\begin{array}{lll}
P_1^R: & a^- \leftarrow not\ rej(a^-,0). & b^- \leftarrow not\ rej(b^-,0). \\
 & a \leftarrow b,\ not\ rej(a,1). & rej(a^-,0) \leftarrow b. \\
P_2^R & c^- \leftarrow not\ rej(c^-,0). & \\
 & rej(b^-,0). & rej(c^-,0). \\
 & b \leftarrow not\ rej(b,2). & c \leftarrow not\ rej(c,2). \\
P_3^R: & a^- \leftarrow c,\ not\ rej(a,3). & rej(a,1) \leftarrow c.
\end{array}
$$

For computing the refined semantics of $\mathcal{P}$ at P_2 we just have to compute the stable model semantics of the program $P_1^R \cup P_2^R$. This program has a single stable model M^T consisting of the following set[9]*.*

$$M^T = \{a^-,\ a,\ b,\ c,\ rej(a^-,0),\ rej(b^-,0),\ rej(c^-,0)\}$$

We conclude that $\mathcal{P}$ has $M = \{a,\ b,\ c\}$ as the unique refined model. at P_2. To compute the refined semantics of $\mathcal{P}$ we have to compute, instead, the stable model semantics of the program $P^R = P_1^R \cup P_2^R \cup P_3^R$. Let us briefly examine the transformed program P^R and see how it clarifies the meaning of the related DyLP. Since there exist no rules with head $rej(b,2)$ and $rej(c,2)$, we immediately infer b and c. Then we also infer $rej(a,0)$, $rej(b,0)$ and $rej(c,0)$, and so that all the default assumptions are rejected. The last rule of P_3^R implies $rej(a,1)$, thus the rule $a \leftarrow b$ in P_1 is rejected and we do not infer a. Indeed, we infer a^- by the first rule of P_3^R. Hence, the program has the single stable model M^T which means $\mathcal{P}$as the unique refined model $\{b,\ c\}$.

To compute the refined semantics of a given DyLP $P_1, \ldots, P_n$ at a given state, it is sufficient to compute its refined transformational equivalent $P_1^R, \ldots P_n^R$, then to compute the stable model semantics of the normal logic program $\mathcal{P}^{Ti}$ and, finally, to consider only those literals that belong to the original language of the program. A feature of the transformation of Definition 38 is that of being incremental i.e., whenever a new update P_{n+1} is received, the transformational equivalent of the obtained DyLP is equal to the union of P_{n+1}^R and the refined transformational equivalent of the original DyLP. The efficiency of the implementation relies largely on the size of the transformed program compared to the size of the original one. We present here a theoretical result that provides an upper bound for the number of clauses of the refined transformational equivalent of a DyLP.

Theorem 3.11.2 *Let $\mathcal{P} = P_1, \ldots P_n$ be any finite ground DyLP in the language $\mathcal{L}$ and let $\mathcal{P}^{Tn}$ be the set of all the rules appearing in the refined transformational equivalent of $\mathcal{P}$. Moreover, let m be the number of clauses in $\rho(\mathcal{P})$ and l be the cardinality of $\mathcal{L}$*[10]*. Then, the program $\mathcal{P}^{Tn}$ consists of at most $3m + l$ rules.*

[9] As usual in the stable model semantics, hereafter we omit the negative literals

[10] Since $\mathcal{L}$ contains the positive and the negative literals, l is equal to two times the number of predicates appearing in $\mathcal{P}$.

proof It follows directly from Definition 38. The number of *defaults assumptions* is at most the size of the set of atoms in $\mathcal{L}$ i.e. $l/2$. Also the number of *totality constraints* is at most the number of atoms in $\mathcal{L}$ $(l/2)$. The number of *rewritten rules* is exactly the number of all the rules in any update P_i i.e. m. Each rule $L \leftarrow B$ in any P_i generates at most two rejection rules, one of the form $rej(\overline{not\, L}, j) \leftarrow \overline{B}$ and another one of the form $rej(\overline{L}, j) \leftarrow rej(\overline{L}, i)$. Hence there are at most $2m$ rejection rules. There are no other rules in $\mathcal{P}^{Tn}$. Hence there are at most $3m+l$ rules in $\mathcal{P}^{Tn}$ $\diamondsuit$

The size of the refined transformational equivalent of a DyLP depends linearly and solely on the size m of the program and of the language, and it can be computed in polynomial time w.r.t. to the size itself. The size of the transformed program has an upper bound which does not depend on the number of updates performed. Thus, we gain the possibility of performing several updates of our knowledge base without losing too much on efficiency. From Theorem 2.4.2, we obtain the following results on the complexity of the refined semantics.

Theorem 3.11.3 *Let $\mathcal{P}$ be a DyLP over the language $\mathcal{L}$, m the total number of rules appearing in any update of P and l the size of $\mathcal{L}$. Then:*

- *Deciding whether $\mathcal{P}$ has a refined dynamic stable model is a NP-complete problem w.r.t. $m+l$.*
- *Deciding whether an interpretation M refined dynamic stable model of $\mathcal{P}$ is a linear problem w.r.t $m+l$.*
- *Deciding whether a given literal is true in at least one refined dynamic stable model of $\mathcal{P}$ is a NP-complete problem w.r.t. $m+l$.*
- *Deciding whether a given literal is true in all the refined dynamic stable models of $\mathcal{P}$ is a $co-NP$-complete problem w.r.t. $m+l$.*

The refined transformation presents some similarities with the ones presented in [ALP$^+$00a] and [EFST02a]. The three transformations use new atoms to represent rejection of rules. A fundamental difference between these transformations is that they are not semantically equivalent. The transformation in [ALP$^+$00a] is defined for implementing the *dynamic stable model semantics of DyLPs* of [ALP$^+$00a], while the one in [EFST02a] implements the *Update semantics* [EFST02a]. These semantics are not equivalent to the refined one, which was, indeed, introduced for solving some counterintuitive behavior of the previously existing semantics for DyLPs (cf. [ABBL05]). In particular, it is proved in [ABBL05] that every refined stable model is also a dynamic stable model and an update stable model but the opposite is not always true. Moreover, the size of the transformation defined in [ALP$^+$00a] is $2m+l(n+2)$ where l and m are as in Theorem 3.11.2 and n is the number of updates of the considered DyLP. Hence, a single

(even empty or single rule) update adds at least l rules to the transformed program. A similar result also holds when considering the transformation of [EFST02a] (here the size of the transformed program is $2m+nl$). The size of the refined transformational equivalent is instead independent from n. Hence, for DyLPs with many updates, the transformed programs of these transformation become considerably larger then the ones of the refined transformation, especially in cases where each of these updates has few rules.

Moreover, the transformations of [ALP+00a] and [EFST02a] approach the problem of computing the semantics at different updates by introducing an extra index on the body of the transformed program. On the contrary, when using the refined transformation, it is sufficient to ignore the rules of the transformed program that are related to the updates after P_i. Apart from computational aspect, the use of extra indexes and the proliferation of rules make these semantics unsuitable for the purpose of understanding the behavior of the updated program.

3.12 Transformational semantics for MDyLPs

Definition 37 and 32 provides two distinct characterizations of the refined semantics for MDyLPs. Anyway it would not be an easy task to directly implement algorithms to compute the semantics of a MDyLP. We adopt, instead, a different approach to the implementation of the refined semantics for MDyLPs. We provide a syntactical transformation that, starting from an *MDyLP* $\mathcal{MP}$ defined over a language $\mathcal{L}$, and a specific node n, gives a transformed *normal logic program* P^T in an extended language, such that the refined stable models of $\mathcal{MP}$ at node P_n are exactly the stable models of P^T restricted to the original language $\mathcal{L}$. We obtain then a third characterization of the defined semantics. Using this transformational approach, it is possible to use efficient implementations of the stable models semantics for normal logic programs like smodels [SMO00] and DLV [DLV00] for computing the refined semantics of a MDyLP.

Definition 39 *Let $\mathcal{MP}$ be a MDyLP over language $\mathcal{L}$ and n be the index of a node P_n in $\mathcal{MP}$. and let L^-, $inf(L,i)$, $ok(L,i)$, and $safe(L,i)$ for each pair i, where $L \in \mathcal{L}$ and i is a node of $\mathcal{MP}$ with $i \preceq j$, be new atoms not appearing in $\mathcal{L}$. The* transformational equivalent *of $\mathcal{MP}$ at P_n is the normal logic program*

$$Tr(\mathcal{MP}^n) = DefR(\mathcal{MP}^n) \cup Inf(\mathcal{MP}^n) \cup Prog(\mathcal{MP}^n) \cup OK(\mathcal{MP}^n) \cup Cons(\mathcal{MP}^n)$$

over the language $\mathcal{L}^T$, where the sets $DefR(\mathcal{MP}^n)$, $Inf(\mathcal{MP}^n)$, $Prog(\mathcal{MP}^n)$, $OK(\mathcal{MP}^n)$ and $Cons(\mathcal{MP}^n)$ are defined as follows

- *The set $DefR(\mathcal{MP})$ of* default rules *is the set consisting of a rule*

$$A^- \leftarrow not\ \ inf(A,i_1),\ldots,not\, inf(A,i_n).$$

for any atom A in the language $\mathcal{L}$, where $not\, inf(A,i_1),\ldots,not\, inf(A,i_n)$ is the conjunction of all $not\, inf(A,i_s)$ such that $i_s \leq n$.

- *The set $Inf(\mathcal{MP})$ of* local inference rules *is the set consisting of a rule*

$$inf(L,i) \leftarrow \overline{B}$$

for for each rule $L \leftarrow B.$ in any P_i.

- *The set $Prog(\mathcal{MP})$ of* program rules *is the set consisting of a rule*

$$\overline{L} \leftarrow \overline{B}, safe(L,i)$$

for for each rule $L \leftarrow B.$ in any P_i.

- *The set $OK(\mathcal{MP}^n)$ of* OK rules *is the set consisting of a rule*

$$safe(L,i) \leftarrow ok(L,j_1),\ldots ok(L,j_m)$$

for each j_s such that $i \not\prec j_s$ and $j_s \preceq n$. a rule

$$ok(L,i) \leftarrow \overline{B}$$

for each rule $rej(not\,,i) \leftarrow B$ in $Rj(\mathcal{MP}^n)$ and a rule

$$ok(L,i) \leftarrow not\, inf(not\, L,i)$$

for for each pair (L,i) such that L is a literal over $\mathcal{L}$ and i is a node st. $i \preceq n$.

- *The set $Rj(\mathcal{MP})$ of* threat rejection rules *$rj(L,i) \leftarrow inf(not\, L,j), below(i,j)$ for each triple (L,i,j) such that L is a literal over $\mathcal{L}$ and $i,\ j$ are nodes.*

- *An atom $below(i,j)$ for each pair of nodes i,j such that $P_i \prec P_j$.*

- *And finally the set $Cons(\mathcal{MP})$ of* consistency check rules. *$\leftarrow A,\ A^-$ for each atom A in $\mathcal{L}$.*

We prove that the refined stable models of a MDyLP at a given nodes are in one-to-one correspondence with the stable models of its the transformational equivalent.

Theorem 3.12.1 *Let $\mathcal{MP}$ be a MDyLP over the language $\mathcal{L}$, P_i any update of $\mathcal{MP}$, $Tr(\mathcal{MP}^n)$ the transformational equivalent of $\mathcal{MP}^n$ defined over the language $\mathcal{L}^T$ and M an interpretation over $\mathcal{L}$. Then M is a refined stable model of $\mathcal{MP}$ at P_n iff there exists a stable M^* of $Tr(\mathcal{MP}^n)$ such that $M \equiv |_{\mathcal{L}} M^*$. If M^* is a stable model of $Tr(\mathcal{MP}^n)$, there does not exist another stable model M_2^* of $Tr(\mathcal{MP}^n)$ such that $M_2^* \equiv |_{\mathcal{L}}\, M^*$*

3.13 Concluding remarks

The chapter explores the framework of dynamic logic programs focusing on semantics based on the causal rejection principle. We have started by motivating and introducing a new general principle – the *refined extension principle* – that can be used to compare semantics of logic programs that are obtainable from the least model of the original program after the addition of some (default) assumptions. The principle states that the addition of rules that do not change the least model of the program together with the assumptions can never generate new models. A special case of this principle concerns the addition of tautologies. Not surprisingly, the stable model semantics for normal and generalized logic programs complies with this principles.

We have generalized this principle for the case of dynamic logic programs, noticing that none of the existing semantics complies with it. A clear sign of this, already mentioned in the literature [EFST02a, Lei03], was the fact that none of these existing semantics is immune to tautologies. We have illustrated how, among these existing semantics, the dynamic stable model semantics [ALP$^+$98b, ALP$^+$00b, Lei03] is the one that complies with the principle for a wider class, viz., the class of dynamic logic programs without conflicting rules in a same program in the sequence.

To remedy this problem, exhibit by the existing semantics, and guided by the refined extension principle, we have introduced a new semantics for dynamic logic programs – the *refined dynamic stable model semantics* – and shown that it complies with such principle. Furthermore, we have proved that this semantics is immune to tautologies.

We have compared the new semantics with extant ones and have shown that, besides the difference regarding the principle, this semantics is more credulous than any of the others, in the sense of admitting a smaller set of stable models (thus yielding a larger intersection of stable models).

A further objective was to establish whether the refined semantics can be properly considered as *the* stable models-like semantics for DyLPs based on causal rejection. Our idea was to extend the definition of well-supported model to the dynamic case. It turns out that, for DyLPs, our characterization coincides with the refined semantics. Strong of the two equivalent characterizations we have extended our investigation of the new semantics by considering notions and properties form existing works.

Then we extended our investigation to multidimensional dynamic logic programs that generalize DyLPs by replacing the condition of total order among programs with the less restrictive condition of a partial order. We defined a well-supported semantics for MDyLPs which extends the refined well-supported semantics for DyLPs, provided a fixpoint characterization of such semantics and established relationships between the new semantics for MDyLPs and existing ones. Finally we provided operational equivalent of the semantics by program transformations of DyLPs and MDyLPs into normal logic programs.

An aspect not explored by our work is the to define classes of logic programs for which the various semantics based on causal rejection (especially the refined semantics presented herein) coincide. This is the main subject of a paper from Martin Homola [Hom04] that also extends the refined semantics in order to allow both explicit negation and default negation in the head.

We believe that the main question to solve in the future is: is it possible to conclude that the refined semantics is the proper extension of the SMs semantics to DyLPs and MDyLPs? Unfortunately there exists no theoretical result which is equivalent to the statement "*this is the correct semantics*". Nevertheless we claim that the characterizations given herein provides some evidence that further refinements of the semantics will not be necessary at least if we restrict the requirement to the causal rejection principle.

In Section 3.7 we informally illustrated the property of minimality of change and shown, from one side, that the refined semantics and the others semantics based on causal rejection do not satisfy this property and from the other side that there are examples suggesting that this property is not always desirable. The discussion opens a new kind of issues: which are the desirable properties or postulates that a proper semantics for updates should satisfy? There are several works like, for instance, [EFST02a, KM91, MC07] trying to fix properties that should be satisfied by semantics for updates. Nevertheless it is not always clear whether a property should be satisfied and what could be the area of applications of updates that demands for the specific properties. Relevant areas of investigation in the future could then be:

- To individuate a set of relevant properties for semantics of dynamic logic programs.
- To specify to which areas of application the various properties are related.
- To understand which of these properties are satisfied by the refined semantics and in case, which are the eventual needed refinement for obtaining a semantics satisfying the desirable properties.

This investigation on the basic properties should be then extended to the multi-dimensional case. For instance it could be relevant to know whether it is possible to reformulate the refined extension principle in order to apply it to MDyLPs. Another work that is in order for MDyLPs is the implementation of the operational semantics shown in Section 3.12.

Chapter 4

A well founded semantics for dynamic logic programs

Contents

4.1 Introduction **93**

4.2 The notion of causal rejection for three-valued semantics **95**

4.3 The Well founded semantics for DyLPs **97**

4.4 Illustrative examples **102**

4.5 Properties of the well founded semantics of DyLPs **107**

4.6 Relations with the well founded semantics of GLPs **111**

4.7 Transformational well founded semantics **111**

4.8 Related works **113**

4.9 Concluding remarks **115**

As mentioned in chapter 3, before the present work various semantics have been proposed for dealing with updates in the setting of logic programs. Despite of this considerable work, very few attempts had been made to establish a well founded semantics for logic programs updates, since the efforts of researchers had concentrated on the stable model approach. As we motivate in the beginning of the chapter, in some potential areas of applications of Dynamic Logic Programs a well founded approach seems to be more suitable than a stable model one, mainly for computational efficiency reasons. In this chapter we propose a well founded semantics for Dynamic Logic Programs. Various theoretical results show how our proposal is related to the stable model approach and how it extends the well founded semantics of generalized logic programs. Moreover we show strong links between the well founded and the refined semantics proposed, the former being a skeptical approximation of the latter.

As a side effect of our efforts, we define also a declarative characterization of the well founded semantics of generalized logic programs. We will also prove properties related to the computational efficiency of the semantics, such as relevance and the polynomial computational complexity of the approach. As for the refined semantics, we also present an operational equivalent definition of the semantics, based on a program transformation in normal logic programs. Part of the results of this chapter have been published in [BAB04, BAB05].

4.1 Introduction

In the previous chapter we have discussed several semantics for logic programs updates and, to overcome existing limitations, proposed a new one called the refined (or well supported) semantics.

All the semantics presented so far, included our proposal, are extensions of the stable models semantics of extended or generalized logic programs. This is a natural choice given the appropriateness of stable models for knowledge representation, and the simplicity of the definition of stable model semantics for normal LPs, which allows various extensions in a natural way.

However, it is our stance that there are application domains for logic programs updates with requirements demanding a different choice of basic semantics, such as the well founded semantics [GRS91].

Many practical applications require treatment of huge amount of data, and any reasonable outlook promises an enforcement of this tendency in the future. Moreover, data is frequently collected from heterogeneous sources. When some forms of logic structure and deductions on this data is required, KR in general, and specifically Logic Programming, come into play. In particular DyLPs could be a tool for managing updates that frequently occur in highly dynamic environments, automatically incorporating new knowledge within the existing information, without falling into an inconsistency each time the new knowledge is in conflict with previous one. An application domain showing all the characteristics listed above (huge amount of data, heterogeneity, structured information, need for knowledge processing and querying, frequent updates) is, for instance, the Semantic Web. In [Sem07] the authors list the desired requirement of a semantic web services language. Among these requirements there is the capability to handle updates and queries of data, knowledge representation and automated reasoning capabilities, the capability of supporting multi-party activities across organizational boundaries, robustness in the sense of continuing to operate despite abnormalities in inputs (like, for instance, inconsistencies) and a declarative semantics.

However, any KR paradigm (and specifically DyLPs) aiming to address applications with the discussed characteristics must face some specific problems. The first and most important one is complexity. When dealing with an overwhelming mass of information, it is very important to be able to quickly process such information, even at the cost of losing some inference power. A problem posed by heterogeneity is the presence of contradictions. In a KB with a great amount of large and distributed information, some errors may occur, that generate inconsistencies. Even in absence of errors, usually the knowledge comes from various sources. When merging together such knowledge, mutual conflicts are possible. A possible way to solve the problem is to use some mechanism of belief revision [RF89, Gär92]. Unfortunately, as it is well

known, the belief revision techniques are computationally expensive. This is in conflict with our fundamental request for efficient ways of processing knowledge. A less computationally expensive way to address contradictions is to adopt a paraconsistent semantics [DP98], i.e. a semantics where contradictions are detected, isolated, which means they do not prevent the KB to have a (non trivial) semantics, but not removed.

Stable models related approaches to DyLPs (included the refined semantics) seem not to be the proper answer to these issues, since they inherit the features of the stable model semantics (and corresponding implementations [DLV00, SMO00]). In the stable-models approaches to DyLPs, the meaning of the whole program is obtained at once. For this reasons it is not appropriate when we are only interested about specific information. Indeed, the stable model semantics does not complies with Relevance [Dix95b], making it hard to use query driven approaches. Moreover it does not assign a meaning to every program, even in absence of contradictory information. This is a drawback since irrelevant information may prevent a specific query to have an answer by making the whole program meaningless. Last but not least, inference in all the semantics discussed in chapter 3 is at least a Co-NP-complete problem.

On the other hand, in our opinion the well founded semantics for logic programs is an adequate answer to the issues of complexity and treatment of contradictions raised above. The well founded semantics has a polynomial complexity. Moreover it complies with the Property of Relevance [Dix95b], making it possible to implement query driven proof procedures that, for any given query, only need to explore a part of the whole KB. The well founded conclusions are sound with respect to the stable models semantics, i.e. the well founded semantics is a skeptical approximation of the stable model semantics. Paraconsistency has been already incorporated in the well founded semantics [AP96, DP96], and for any logic program its paraconsistent well founded semantics is always uniquely determined. Moreover, in a paraconsistent well founded semantics we can ignore the semantics of the program as a whole, and concentrate on the meaning of a specific query.

Hence, to address the issues of complexity and contradiction treatment within logic programs updates, we define a well founded paraconsistent semantics for dynamic logic programs.

As for the refined semantics, the approach here in is also based on the causal rejection principle. However, since the well founded semantics is not limited to two-valued interpretations, we extend such principle from the case of two valued semantics to the three valued case. Between the well founded and refined stable model semantics, for DyLPs there are the same relations that are proved to exist between the well founded and stable model semantics of normal logic programs [GRS91]. Indeed, any well founded conclusions is part of any refined model and the two semantics coincide whenever the well founded model of a DyLP is two valued. Moreover, the same relations do not hold for any other semantics based on causal rejection. It is also proved

here that the well founded semantics of DyLPs coincides with the well founded semantics of generalized logic programs [LW92, DP96] when the considered sequence of updates is a single logic program. Hence the former can be considered as an extension of the latter which is itself an extension of the well founded semantics for normal logic programs. As a side effect of our research, we present also a purely declarative definition of the well founded semantics for generalized logic programs. Another similarity with the static case is that the well founded semantics for DyLPs is *relevant* [Dix95b] which implies that inference over a particular conclusion is possible, in general, by considering only a subset of all the rules in the considered DyLP. The well founded semantics for DyLPs is paraconsistent and as such, it is always defined, even for contradictory programs.

Finally, as for the refined semantics, we also present an operational equivalent definition of the semantics, based on a program transformation in normal logic programs. We use this result to prove that the well founded semantics for DyLPs has polynomial complexity.

Prototypical implementations for the well founded semantics in the linear case that use the theoretical background of this chapter are available at [Ban05b]. This implementation takes advantage of XSB-Prolog language to compute the semantics of the transformed programs.

The rest of the chapter is organized as follows. Section 4.2 is devoted to present the extension of the causal rejection principle to the three valued case, while in Section 4.3 the well founded semantics for DyLPs is motivated and presented. Section 4.4 illustrates the behavior of the semantics by some specific examples and show how it is computed in practice. Sections 4.5 shows properties of the well founded semantics, in particular its relation with causal rejection principle, the relationship with the refined semantics and other stable model-like semantics based on causal rejection. Section 4.6 shows how the well founded semantics for DyLPs extends the well founded semantics for generalized and normal logic programs [DP96]. Section 4.7 describes the operational semantics for DyLPs. In Section 4.8 we compare our work with other approaches for defining a well founded semantics for DyLPs. In Section 4.9 we draw conclusions and sketch possible future works. In Appendix B, the reader can find the proofs of the results presented throughout the chapter.

4.2 The notion of causal rejection for three-valued semantics

According to the causal rejection principle introduced in chapter 3, a rule from an older program in a sequence is kept (by inertia) unless it is rejected by a more recent conflicting rule whose body is true. On the basis of this, the very basic notion of model has to be modified when dealing with updates. In the static case a model of a program is an

interpretation that satisfies all the rules of the program, where a rule is satisfied if its head is true or its body is false. If we want to adapt this idea to the dynamic setting, taking into consideration the causal rejection principle, we should only require non rejected rules to be satisfied.

In chapter 3 we have seen how the concept of supported model (see chapter 2) has to be revisited when dealing with updates. In the static case, a model M of P is supported iff for every atom $A \in M$, there is a rule in P whose head is A and whose body is satisfied in M. If we extend the concept of supportedness to DyLPs, it would be unnatural to allow rejected rules to support a the truth of a literal.

The causal rejection principle was defined for 2-valued semantics. We want now to extend it to a three-valued setting, in which the truth value of literals can be $undefined$, besides being $true$ or $false$. In the 2-valued setting, a rule is rejected iff there is a conflicting rule in a later update whose body is true in the considered interpretation. In this context, this is the same as saying that the body of the rejecting rule is not false. In a three-valued setting this is no longer the case, and the following question arises: should we reject rules on the basis of rejecting rules whose body is *true*, or on the basis of rules whose body is *not false*? We argue that the correct answer is the latter. In the remainder of the section, we give both practical and theoretical reasons for our choice, but we want now to give an intuitive justification. Let us suppose initially we believe a given literal L is true. Later on we get the information that L is false if some conditions hold, but those conditions are (for now) undefined. As usual in updates, we prefer later information to the previous one. On the basis of such information, can we be *sure* that L remains true? It seems to us we cannot. The more recent source of information says if some conditions hold then L is false, and such conditions *may* hold. We should then reject the previous information and consider, on the basis of the most recent one, that L is undefined.

On the basis of these intuitions, we extend the definition of update model and update supported model to the three-valued setting.

Definition 40 *Let $\mathcal{P}$ be any DyLP, and M a three-valued interpretation. M is an* update three-valued model *of $\mathcal{P}$ iff for each rule τ in any given P_i, M satisfies τ (i.e. $hd(\tau) \in M$ or $B(\tau) \not\subseteq M$) or there exists a rule η in P_j, $i < j$ such that $\tau \bowtie \eta$ and $B(\eta)$ is not false in M. We say M is a* supported three-valued update model *of $\mathcal{P}$ iff it is an update three-valued model and*

1. *for each atom $A \in M$, $\exists\, \tau \in P_i$ with head A such that $B(\tau) \subseteq M$ and $\nexists\, \eta \in P_j$, $i < j$ such that $\tau \bowtie \eta$, and $B(\eta)$ is not false in M.*

2. *for each negative literal not A, if not $A \in M$, then for each rule $A \leftarrow body \in \rho(\mathcal{P})$ such that $body$ is true in M, there exists a rule η, in a later update whose head is not A, and such that $B(\eta)$ is true in M.*

We illustrate, via an example, the intuitive meaning of the defined concepts.

Example 4.2.1 *Sara, Cristina and Bob, are deciding what they will do on Saturday. Sara decides she is going to a museum, Cristina wants to go shopping and Bob decides to go fishing in case Sara goes to the museum. Later on they update their plans: Cristina decides not to go shopping, Sara decides she will not go to the museum if it snows and Bob decides he will also go fishing if it is a sunny day. Moreover we know from the forecast that Saturday can be either a sunny day or a raining day. We represent the situation with the DyLP $P_1, P2$, where:*

$$
\begin{array}{ll}
P_1: & museum(s). \\
 & shopping(c). \\
 & fish(b) \leftarrow museum(s). \\
P_2: & fish(b) \leftarrow sunny. \\
 & sunny \leftarrow not\ rain. \\
 & rain \leftarrow not\ sunny. \\
 & not\ shopping(c). \\
 & not\ museum(s) \leftarrow snow.
\end{array}
$$

The intended meaning of P_1, P_2 is that it does not snow on Saturday, but we do not know if it does rain or not, we know Sara goes to the museum on Saturday, Bob goes fishing and, finally, Cristina does not go shopping. Indeed, every three-valued update model of P_1, P_2 contains $\{museum(s), not\ shopping(c), fish(b)\}$. Suppose now Bob decides that, in the end, he does not want to go fishing if it rains, i.e our knowledge is updated with: $P_3 : not\ fish(b) \leftarrow rain$. Intuitively, after P_3, we do not know whether Bob will go fishing since we do not know whether Saturday is a rainy day. According to Definition 40, there is a supported three-valued update model of P_1, P_2, P_3 in which $shopping(c)$ is false, $museum(s)$ is true and $fish(b)$ is undefined.

It can be checked that, according to the refined semantics (and all the existing stable models based semantics for DyLPs), P_1, P_2, P_3 has two stable models: one where rain is true and $fish(b)$ is false, and another where rain is false and $fish(b)$ is true. A notable property of the well founded model in the static case is that of being a subset of all stable models. If one wants to preserve this property in DyLPs, in the well founded model of this example one should neither conclude $fish(b)$ nor $not\ fish(b)$. If a rule would only be rejected in case there is a conflicting later one with true *body (rather than* not false *as we advocate), since the body of $not\ fish(b) \leftarrow rain$ is not true, we would not be able to reject the initial rule $fish(b) \leftarrow museum(s)$, and hence would conclude $fish(b)$. Hence, to preserve this relation to stable models based semantics, the well founded model semantics for DyLPs must rely on this notion of three-valued rejection.*

4.3 The Well founded semantics for DyLPs

On the basis of the notion of causal rejection extended to three-valued interpretations, we define the Well Founded Semantics for DyLPs. Formally, our definition is made in a way similar to the the definition of the well founded semantics for normal LPs

in [Gel92], (see chapter 2) where the well founded model is characterized by the least alternating fixpoint of the Gelfond-Lifschitz operator Γ^N (i.e. by the least fixpoint of $\Gamma^N\Gamma^N$). Following the approach illustrated in chapter 2 for normal LPs, we could select one of these two operators and define the well founded semantics in terms of the least fixpoint of the double application of either the Γ operator of Definition 19 or the Γ^R operator of Definition 27. Indeed, as stated in the following lemma, both the operators are anti monotonous.

Lemma 4.3.1 *The operators $\Gamma_{\mathcal{P}}$ and $\Gamma^R_{\mathcal{P}}$ are both antimonotonous.*

Unfortunately, if we apply literally this idea, i.e. define the well founded model as the least alternating fixpoint of the operator used for the dynamic stable (or refined stable) models of DyLPs, the resulting semantics turns out to be too skeptical as shown by the following example.

Example 4.3.1 *It is either day or night (but not both). Moreover, if the stars are visible it is possible to make astronomical observations. This knowledge is updated with the information that: if it is night the stars are visible and the stars are not visible. Finally we make a further update stating that if the observatory is closed if it is not possible to make observations and that it is a ugly night if the stars are not visible.*

$$
\begin{array}{lll}
P_1 : observe \leftarrow stars. & day \leftarrow not\ night. & night \leftarrow not\ day. \\
P_2 : stars \leftarrow night. & not\ stars. & \\
P_3 : closed \leftarrow not\ observe. & ugly \leftarrow not\ stars. &
\end{array}
$$

The intended meaning of $\mathcal{P} = P_1, P_2, P_3$ is that currently the stars are not visible, it is not possible to make astronomical observations and, hence, the observatory is closed. However, it is easy to check, the least alternating fixpoint of $\Gamma_{\mathcal{P}}$ is $\{not\ stars, ugly\}$, in which one is not able to conclude that the observatory is closed. We show the computation of the least fix point of $\Gamma_{\mathcal{P}}$ by iterating from the empty interpretation.

$$
Default(\mathcal{P}, \emptyset) = \left\{ \begin{array}{c} not\ day, not\ night, not\ observe, \\ not\ closed, not\ stars, not\ ugly \end{array} \right\}.
$$

$$
Rej(\mathcal{P}, \emptyset) = \{\}
$$

Then

$$
\Gamma(\emptyset) = least\,(P_1 \cup P_2 \cup P_3 \cup Default(\mathcal{P}, \emptyset)) = \left\{ \begin{array}{c} not\ day, not\ night, not\ closed, \\ not\ ugly, not\ stars, \\ not\ observe, day, night, \\ stars, observe, closed \end{array} \right\}
$$

We iterate again:

$$\begin{aligned} Default(\mathcal{P},\Gamma(\emptyset)) &= \{\} \\ Rej(\mathcal{P},\Gamma(\emptyset)) &= \{\} \\ \Gamma\Gamma(\emptyset) = least(P_1 \cup P_2 \cup P_3) &= \{not\ stars, ugly\} \end{aligned}$$

And found the least alternating fixpoint $F = \{not\ stars, ugly\}$*. Indeed:*

$$Default(\mathcal{P},F) = \left\{ \begin{array}{c} not\ day, not\ night, not\ observe, \\ not\ closed, not\ stars, \end{array} \right\}.$$

$$Rej(\mathcal{P},F) = \{\}$$

$$\Gamma(F) = least(P_1 \cup P_2 \cup P_3 \cup Default(\mathcal{P},F)) = \left\{ \begin{array}{c} not\ day, not\ night, not\ observe, \\ not\ closed, not\ stars, ugly \end{array} \right\} = F1$$

$$\begin{aligned} Default(\mathcal{P},F1) &= \{\} \\ Rej(\mathcal{P},F1) &= \{\} \\ \Gamma(F1) = least(P_1 \cup P_2 \cup P_3) &= \{not\ stars, ugly\} = F \end{aligned}$$

Also the least alternating fixpoint of Γ^R *yields to too skeptical results, since its least alternating fixpoint is* $\{not\ observe, closed\}$ *and hence ugly is not derived.*

Indeed:

$$Rej^R(\mathcal{P},\emptyset) = \{stars \leftarrow night\} = R1$$

Then

$$\Gamma^R(\emptyset) = least\left((P_1 \cup P_2 \cup P_3) \setminus R1 \cup Default(\mathcal{P},\emptyset)\right) = \left\{ \begin{array}{c} not\ day, not\ night, \\ not\ observe, not\ closed, \\ not\ ugly, not\ stars, \\ day, night, closed, ugly \end{array} \right\}$$

We iterate again:

$$\begin{aligned} Default(\mathcal{P},\Gamma^R(\emptyset)) &= \{not\ observe\} = D \\ Rej^R(\mathcal{P},\Gamma^R(\emptyset)) &= \{stars \leftarrow night, not\ stars,\} = R2 \\ \Gamma^R\Gamma^R(\emptyset) = least\left((P_1 \cup P_2 \cup P_2) \setminus R_2 \cup D\right) &= \{not\ observe, closed\} = G \end{aligned}$$

and G *is the least alternating fixpoint of* Γ^R*. As it results from the computation below.*

$$Default(\mathcal{P},G) = \left\{ \begin{array}{c} not\ observe, not\ day, not\ night, \\ not\ ugly, not\ stars \end{array} \right\}$$

$$\begin{array}{rcl} Rej^R(\mathcal{P},G) & = & R_1 \\ \Gamma^R(G) = least\left((P_1 \cup P_2 \cup P_2) \setminus R_1 \cup Default(\mathcal{P},G)\right) & = & \left\{ \begin{array}{c} not\ observe, not\ day, \\ not\ night, not\ ugly, \\ not\ starsday, \\ night, closed, ugly \end{array} \right\} \end{array}$$

$$\begin{array}{rcl} Default(\mathcal{P},\Gamma^R(G)) & = & \{not\ observe,\} = D \\ Rej^R(\mathcal{P},\Gamma^R(G)) & = & R_2 \\ \Gamma^R\Gamma^R(G) = least\left((P_1 \cup P_2 \cup P_2) \setminus R_1 \cup D\right) & = & \{not\ observe, closed\} = G \end{array}$$

And hence G is the least alternating fixpoint of Γ^R.

In order to overcome this excess of skepticism, we define the well founded semantic as the least fixpoint of the composition of two different (anti monotonous) operators. Such operators have to deal with the causal rejection principle described above, in which a rule is to be rejected in case there is a later conflicting one whose body is *not false*. In the well founded semantics of normal logic programs, if there exists a rule $A \leftarrow body$ (where A is an atom), such that *body* is not false in the well founded model, then A is not false as well. Consider now the same situation in an update setting with a rule $L \leftarrow body_1$, where $body_1$ is not false. In this situation we should conclude that L is not false *unless* there exists a rule $not\,L \leftarrow body_2$, where $body_2$ is true in the same or in a later program in the sequence. Indeed, note that the rule for $not\,L$ is not rejected by the one for L. Since the body of the former is true, according to the causal rejection principle $not\,L$ should be true (i.e. L should be false) unless the rule is rejected by some later rule. In any case, $L \leftarrow body_1$ is no longer playing any role in determining the truth value of L. For this reason we allow rules to reject other rules in previous *or in the same* update while determining the set of non-false literals of the well founded model and, accordingly, we use $\Gamma^R_{\mathcal{P}}$, as the first operator of our composition.

For determining the set of true literals according to the causal rejection principle, only the rules that are not rejected by conflicting rules in *later* updates should be put in place. For this reason we use $\Gamma_{\mathcal{P}}$ as the second operator, the well founded model being thus defined as the least fixpoint of the $\Gamma\Gamma^R$.

Definition 41 *The well founded model* $WFDy(\mathcal{P})$ *of a DyLP* $\mathcal{P}$ *is the (set inclusion) least fixpoint of* $\Gamma_{\mathcal{P}}\Gamma^S_{\mathcal{P}}$.

Since both Γ and Γ^R are anti monotonous (see lemma 4.3.1), $\Gamma\Gamma^R$ is monotonous, and so it always has a least fixpoint. By the results illustrated in chapter 2 on monotonous

operators, $WFDy$ is uniquely defined for every DyLP. Moreover $WFDy(\mathcal{P})$ can be obtained by (transfinitely) iterating $\Gamma\Gamma^R$, starting from the empty interpretation.

There is a notable difference between the well founded semantics of Definition 41 and the well founded semantics for normal logic programs as defined in Definition 12. While in the former the well founded model *is* the least fixpoint of the considered operator, in the latter the least fix pint is the set of well founded atomic (and hence positive) conclusions.

This difference is due to the fact that while the operator Γ^D of Definition 12 is defined for sets of atoms operators Γ and Γ^R are defined for set of literals (i.e. interpretations) and hence their composite iteration directly computes the set of (positive and negative) well founded conclusions.

The least fixpoint of $\Gamma\Gamma^R$ is the most credulous binary combination of the operators Γ and Γ^R. To prove this we first state a technical lemma on the relationship between the two operators.

Lemma 4.3.2 *Let X, Y be two interpretations with $X \subseteq Y$. Then:*

$$\Gamma^R(Y) \subseteq \Gamma(X)$$

From lemma 4.3.2 it follows that any fixpoint F of a binary combination of Γ and Γ^R is a prefixpoint of $\Gamma\Gamma^R$:

Let F be a fixpoint of $\Gamma\Gamma$ Then by lemmata 4.3.2and 4.3.1:

$$\Gamma(F) \supseteq \Gamma^R(F) \;\Rightarrow\; F = \Gamma\Gamma(F) \subseteq \Gamma\Gamma^R(F)$$

Let F be a fixpoint of $\Gamma^R\Gamma^R$ Then by lemma 4.3.2:

$$F = \Gamma^R\Gamma^R(F) \subseteq \Gamma\Gamma^R(F)$$

Let F be a fixpoint of $\Gamma^R\Gamma$ Then by lemmata 4.3.2and 4.3.1:

$$\Gamma(F) \supseteq \Gamma^R(F) \;\Rightarrow\; F = \Gamma^R\Gamma(F) \subseteq \Gamma\Gamma^R(F)$$

Hence by the results on monotonous operators illustrated in 2.4.2 it follows that the least fixpoint of any prefix binary combination of Γ and Γ^R is a subset of the least fixpoint of $\Gamma\Gamma^R$. Hence having adopted the well founded semantics as the least fixpoint of $\Gamma\Gamma^R$ we opted for the most credulous solution. As shown in example 4.4.1 the inclusions above may be strict.

4.4 Illustrative examples

Here we show by some illustrative examples how the well founded model is constructed. We being by computing the well founded model of the dynamic logic program of example 4.3.1 showing that it leads to the expected results.

Example 4.4.1 *Let $\mathcal{P}$ the the DyLP of example 4.3.1. Then its well founded model is $W = \{not\ stars, not\ observe, closed, ugly\}$ thus matching the intuitive meaning of the program. We show the computation of the well founded model. We already know that*

$$\Gamma^R(\emptyset) = least\left((P_1 \cup P_2 \cup P_3) \setminus R1 \cup Default(\mathcal{P}, \emptyset)\right) = \left\{ \begin{array}{c} not\ day, not\ night, \\ not\ observe, not\ closed, \\ not\ ugly, not\ stars, \\ day, night, closed, ugly \end{array} \right\}$$

Regarding the second iteration:

$$\begin{array}{rcl} Default(\mathcal{P}, \Gamma^R(\emptyset)) & = & \{not\ observe\} = D \\ Rej(\mathcal{P}, \Gamma^R(\emptyset)) & = & \{\} \\ \Gamma\Gamma^R(\emptyset) = least\left(P_1 \cup P_2 \cup P_2 \cup D\right) & = & \{not\ observe, closed, not\ stars, ugly\} = W \end{array}$$

And W is the well founded model of $\mathcal{P}$. Indeed, by iterating again:

$$\begin{array}{rcl} Default(\mathcal{P}, W) & = & \{not\ observe, not\ night, not\ day\} = D1 \\ Rej^R(\mathcal{P}, W) & = & \{stars \leftarrow night\} = R1 \\ \Gamma^R(W) & = & least\left((P_1 \cup P_2 \cup P_2) \setminus R1 \cup D1\right) \end{array}$$

$$\Gamma^R(W) = \left\{ \begin{array}{c} not\ day, not\ night, not\ observe, \\ day, night, closed, not\ stars, ugly \end{array} \right\} = W1$$

and

$$\begin{array}{rcl} Default(\mathcal{P}, W1) & = & \{not\ observe\} = D \\ Rej(\mathcal{P}, W1) & = & \{\} \\ \Gamma\Gamma^R(W) = least\left(P_1 \cup P_2 \cup P_2 \cup D1\right) & = & \{not\ observe, closed, not\ stars, ugly\} = W \end{array}$$

In chapter 3 we underlined that, with the exception of the refined semantics, all the semantics for DyLPs based on causal rejection show criticality with DyLPs containing conflicting rules in the same program updated by a program containing a tautology.

We show now by an example (and we will state it formally by Theorem 4.5.2) that such criticality is not shared by the well founded semantics.

Example 4.4.2 *Let $\mathcal{P} = P_1, P_2, P_3$ be the DyLP in example 4.3.1. If we add a tautology to P_3, i.e. we consider the program $P_3' = P_3 \cup \{stars \leftarrow stars\}$ and $\mathcal{P}' = (P_1, P_2, P_3')$ we may found that the well founded semantics of $\mathcal{P}'$ and $\mathcal{P}$ coincides (as it results from Theorem 4.5.2 stating that the addition of tautologies does not change the semantics of a program).*

The reader may notice that $\mathcal{P}'$ of example 4.4.2 is a simple elaboration of example 3.1.4 used for showing the problem of tautologies with the stable model-based semantics based on causal rejection. It is immediate to verify that the DyLP $\mathcal{P}$ of example 4.3.1 in any stable model-like semantics based on causal rejection has one model namely:

$$M_1 = \{day, not\ observe, closed, not\ stars, ugly\}$$

and that $WFDy(\mathcal{P}) \subseteq M_1$ (see example 4.4.1). The program $\mathcal{P}'$ with a tautological rule has instead *two models* in any semantics based on causal rejection except the refined one namely:

$$\begin{aligned} M_1 &= \{day, not\ night, not\ observe, closed, not\ stars, ugly\} \\ M_2 &= \{night, not\ day, not\ observe, not\ closed, stars, not\ ugly\} \end{aligned}$$

and $WFDy(\mathcal{P}) \not\subseteq M_2$. Hence: *with the exception of the refined semantics, the well founded model of a DyLP is not always a subset of the all the models of that DyLP according to the stable model based semantics for DyLPs based on causal rejection.* As stated in Theorem 4.5.1 the well founded model is indeed a subset of any refine model.

In Section 4.2 we introduced the notion of three-valued update model (see Definition 40). The example below illustrated where the well founded model is a three-valued update model (we refer the reader to Theorem 4.5.1 for the general case when this is true).

Example 4.4.3 *Let $\mathcal{P} = (P_1, P_2, P_3)$ be the DyLP of example 4.2.1. As expected, the well founded model of $\mathcal{P}$ is, $W = \{not\ snow,\ museum(s), not\ shopping(c)\}$ as shown by the computation below.*

$$\begin{aligned} Default(\mathcal{P}, \emptyset) &= \{not\ sunny, not\ rain, not\ snow\} \\ Rej^R(\mathcal{P}, \emptyset) &= \{not\ shopping(C)\} = R1 \\ \Gamma^R(\emptyset) &= \left\{ \begin{array}{c} not\ sunny, not\ rain, \\ not\ snow, fish(b), not\ fish \\ museum(s), not\ shopping(c) \end{array} \right\} \end{aligned}$$

We iterate again and find:

$$\begin{aligned} Default(\mathcal{P}, \Gamma^R(\emptyset)) &= \{not\ snow\} = D \\ Rej^R(\mathcal{P}, \emptyset) &= \{shopping(c), fish(b) \leftarrow museum(s), fish(b)\} = R1 \\ W = \Gamma\Gamma^R(\emptyset) &= \left\{ not\ snow, not\ shopping(c), museum(s) \right\} \end{aligned}$$

Where W is the well founded model of $\mathcal{P}$. Note that W is a supported three-valued update model.

We examine now a case when the given DyLP has a contradiction coming from the presence of conflicting rules in the same update. In this case the well founded is *not* an update model. As it results from Theorem 4.5.1 this is the only case when a well founded model is not an update model.

Example 4.4.4 *We know that the street is wet when it is raining, actually it is not wet. Then we update our knowledge with the information that it is raining and that the street is slippery.*

P_1 : $Wet \leftarrow Rain.$ $\quad not\ Wet.$

P_2 : $Rain.$ $\quad Slippery \leftarrow Wet.$

It is clear that $\mathcal{P} = P_1, P_2$ has no two valued model, since it contains contradictory information. The well founded model of $\mathcal{P}$ is:

$$W = \{Rain,\ Wet,\ not\ Wet,\ Slippery,\ not\ Slippery\}$$

As already pointed out, WFDy is defined for any DyLPs. As we see, the implicit contradiction in (P_1, P_2) is detected but it does not prevent the program to have a semantics, then WFDy is a inconsistent interpretation. From the example it also clear that the inconsistency i.e. the presence of a pair $(A,\ not\,A)$ in the well founded model propagates to other literals that depend on the literals A and $not\,A$. This behavior of the semantics allows a more easy detection of contradictions. We say that any Dynamic Logic Program $\mathcal{P}$ is consistent (or non contradictory) iff $WFDy(\mathcal{P})$ is consistent. In Section 4.5.2 we show a complete characterization of consistent DyLPs.

We end-up with an example involving a large database system.

Example 4.4.5 *A deductive database collects data on companies. Information about company is provided by predicates whose first argument is the key-name of the company and the others are information about the company. For instance, the predicate $employ(Cname, E)$ is true whenever E is the number of employers of the company whose key name is $Cname$, while $turnover(Cname, T)$ is true whenever T is the turnover (expressed in millions of Euros) of $Cname$. As an example, the company $reptile\_112$ has 70 employers and a turnover of 12 millions of Euros, hence the predicates $employ(reptile\_112, 70)$ and $turnover(reptile\_112, 12)$*

hold. A medium company is a company with at least 50 *and at most* 200 *employers. This is translated by the following rule:*

$$mediumC(Cname) \leftarrow employ(Cname, E),\ E \geq 50,\ E \leq 200.$$

This definition of medium company is later updated. The new definition of a medium company requires that the turnover of the company is at least of 15 *millions of Euros. The database may be represented by the following DyLP* $\mathcal{P} = P_1, P_2, P_3$.

$$\begin{array}{ll} P_1: & \ldots \quad employ(reptile\_112, 70). \quad turnover(reptile\_112, 12). \quad \ldots \\ P_2: & mediumC(Cname) \leftarrow employ(Cname, E),\ E \geq 50,\ E \leq 200. \\ P_3: & not\ mediumC(Cname) \leftarrow turnover(Cname, T),\ T \geq 15. \end{array}$$

where P_1 *contains all the stored data (with the dots representing the data we are not interested in),* P_2 *the initial rule and* P_3 *its update. According to* $\mathcal{P}$*, the company* $reptile\_112$ *is not a medium company. The fragment of the well founded model of* $\mathcal{P}$ *we are interested in is:*

$$\{employ(reptile\_112, 70), turnover(reptile\_112, 12), not\ mediumC(reptile\_112)\}$$

By resorting, for instance, to the implementation of the well founded semantics for DyLPs [Ban05b] it is possible to query the DyLP on, for instance, the truth value of the single predicate $mediumC(reptile\_112)$*. The implementation takes advantages of the program transformation described in Section 4.7 where a DyLP is turned into a normal logic program with the same semantics. The cited implementation does not need to compute the whole well founded model of* $\mathcal{P}$ *in order to answer to a query like the one above.*

Example 4.4.6 *The updates showed in this example may either represent temporal updates or different degrees of importance among rules, when rules with different degrees are conflicting, priority is assigned to rules with the higher one. This is achieved simply by putting rules in different updates, the higher is their degree of importance, the more recent is the update they belong to:*

Vampire *is an agent whose function is to crawl through the web searching for information, and downloading interesting files and web pages. The KB of Vampire contains knowledge, in the form of logic programming rules, about the degrees of importance of the found material. This knowledge is important for the agent in order to select which files and web pages it should download. Let us suppose, for instance, that the following rule is part of an update* P_1

$$Download(X) \leftarrow File(X),\ Subject(X, Matrix\_movie).$$

This rule states that any file X should be downloaded if its subject is related to the movie "The Matrix". Another rule in the update P_2 *states that a file should not be downloaded if the process*

is estimated to require more then 10 minutes.

$$not\ Download(X) \leftarrow File(X), Greater(Dowloading\_time(X), 10\_min).$$

For instance, the file enter_matix.zip, containing the zipped game "Enter the Matrix" has an estimated downloading time of 2 hours and is then not downloaded according to the program P_1, P_2. Let us suppose now that, a higher update P_3, contains the following rule.

$$Download(X) \leftarrow File(X),\ Subject(X, Matrix\_movie),\ Is\_a(X, game)$$

The meaning of this rule is that a video game whose subject is "The Matrix", had to be downloaded. Since $Is\_a(enter\_matrix.zip, game)$ is true and the later rule has an higher priority than the one in P_2, the former rejects the latter. Hence Vampire downloads the file, even if it will take time.

Example 4.4.7 *Health care has been designed has a major application area for the Semantics Web by the W3C and an official W3C Interest Group has been found for promoting its development [W3C04]. The purpose is to automatically merge knowledge and activities on the subject by the Web. In the following scenario, a health care system is responsible for detecting and responding to emerging public health problems. The system collects data and knowledge from different sources like hospitals and medical researcher teams, by automatic updates though the Web. A similar Semantics Web application really exists and it is called SAPPHIRE [sap04] although it is not currently implemented by resorting on logic programming.*

The considered fragment attempts to make diagnosis on a possible infections of a specific kind of influenza say, for instance, the Spanish Influenza (shortly si) by the symptoms of the patient. Since knowledge is collected from different sources, contradictions among rules may arise. For instance, an empirical rule suggested by hospital A says that it is likely that any patient (shortly Pa) has the Spanish Influenza if it has high fever (shortly hf), and diarrhea (shortly di). Another empirical rule suggested by hospital B says that it is likely that the patient does not have the Spanish Influenza if it has dry cough (shortly dc).

This disease taxonomy is encoded by the following program P

$$\begin{aligned} disease(si, Pa) &\leftarrow symptom(hf, Pa), symptom(di, Pa). \\ not\, disease(si, Pa) &\leftarrow symptom(dc, Pa). \end{aligned}$$

The patient Jonathan has both high fever ($symptom(hf, jonathan)$ is true) and diarrhea ($symptom(di, jonathan)$ is true), but it has no dry cough ($symptom(dc, jonathan)$ is false). Hence, according to the rules above, a Spanish Influenza is diagnosed and, indeed, according to the well founded model of the program P, $disease(si, jonathan)$ is true.

Another patient Jacqueline, still has both high fever and diarrhea symptoms ($symptom(hf, jacqueline)$ and $symptom(di, jacqueline)$ are both true) but it has also dry cough ($symptom(dc, jacqueline)$ is also true). According to the well founded semantics of P, $disease(si, jacqueline)$ is contradictory, since both $disease(si, jacqueline)$ and

not $disease(si, jacqueline)$ *belong to its well founded model.*

Note that, although P is a contradictory program, this does not prevent a user to query P and obtain consistent answers about the non contradictory part of its knowledge. On the other side, it is possible to detect the contradictions in the system by asking if both the literals A and not A belong to the well founded model of the program.

It is also important to note that, since the system collects data on possibly thousands of patients, it is necessary to have an inference system capable of quickly and selectively querying a huge amount of data and rules. Moreover, since the rules are automatically collected by different sources through the web, it is virtually impossible to guarantee its global consistency.

Since contradictions can be detected, it is also possible to further update the program in order to remove such inconsistencies. In our case, a research team detects the inconsistency and remove it by updating the system with the single-rule program P_1

$$disease(si, Pa) \leftarrow symptom(hf, Pa), symptom(di, Pa), symptom(hf, dc).$$

According to the well founded semantics of P, P_1 *the literal* $disease(si, jacqueline)$ *is true.*

The reader may notice that, since the system is supposed to collect data and rules from different sources, the best way to implement it by LP updates would be to resort on multidimensional *rather than simply linear DyLPs also enabling the developer to establish priorities among the various sources. This would require an extension of the well founded semantics presented herein to the case of multidimensional DyLPs. Although this issue is beyond the scope of this work, we will trace a possible route to this generalization of the framework in Section 4.9.*

4.5 Properties of the well founded semantics of DyLPs

In Section 4.2 we introduced the notion of three-valued update model (see Definition 40) example 4.4.3 illustrates a DyLP whose well founded model is a three-valued update model, while example 4.4.4 shows a DyLP whose well founded model is contradictory and it is not a well supported model. This is the only existing contradiction, whenever the well founded model does not contain any pair of complementary literals it is a three-valued update model of the considered DyLP.

Theorem 4.5.1 *Let* $\mathcal{P}$ *be a DyLP and W its well founded model. Then, if W contains no pair of complementary literals, W is a supported three-valued update model of* $\mathcal{P}$*.*

The proviso of W not containing any pair of complementary literals is due to the fact that, since the notion of interpretation we use allows contradictory sets of literals, the well founded model of a DyLP can be contradictory. We say that a DyLP $\mathcal{P}$ is consistent (or non contradictory) iff $WFDy(\mathcal{P})$ is consistent, i.e. it does not contain any pair of complementary literals.

We show now two important properties of the proposed semantics that extends well known results for the well founded semantics of normal logic programs, namely

that the well founded model is a subset of any refined stable model and that whenever the well founded model is two-valued it coincides with the unique refined stable models, thus establishing a strong link between the well founded and the refined stable model semantics of DyLPs.

Proposition 4.5.1 *Let $\mathcal{P}$ be any DyLP and M a stable model of $\mathcal{P}$ in the refined semantics. The well founded model of $\mathcal{P}$ is a subset of M.*

Proposition 4.5.2 *Let $\mathcal{P}$ be any Dynamic Logic Program. Let us suppose $WFDy(\mathcal{P})$ is a two valued interpretation, i.e. for every atom A in $\mathcal{H}_P$ exactly one of the two literals A and not A belongs to $WFDy(\mathcal{P})$. Then $WFDy(\mathcal{P})$ coincides with the unique stable model of $\mathcal{P}$.*

proof Presented as a consequence of Corollary 4.5.1. ♢

As it follows from example 4.4.2, and the further discussion on it, the results above are no longer true if, instead of the refined semantics, we consider any other semantics for DyLPs based on causal rejection.

Proposition 4.5.3 *The well founded model of a given DyLP is not always subset of each stable model in any of the stable models semantics for logic programs updates defined in [ALP+98b, BFL99, EFST02a, LP97, LP98, ZF98, Lei03]*

Another similarity between the refined and well founded semantics for DyLPs is that both are immune to tautologies, i.e. the equivalent of Theorem 3.4.2 holds for the well founded semantics as well.

Theorem 4.5.2 *Let $\mathcal{P}$ be any DyLP and $\mathcal{E}$ a sequence of sets of tautologies. W is the well founded model of $\mathcal{P}$ iff W is the well founded model of $\mathcal{P}' = \mathcal{P} \cup \mathcal{E}$.*

In the following we study some properties of the fixpoints of the $\Gamma\Gamma^R$ operator. Since the well founded model is the least fix of such points, these properties hold for the well founded model as well.

4.5.1 Properties of the fixpoints of $\Gamma\Gamma^R$

In the following, given a Dynamic Logic Program $\mathcal{P}$, F will be a fixpoint of the $\Gamma\Gamma^R$ operator.

This initial proposition is a fundamental result, since it establishes a strong link between a fixpoint F and $\Gamma^R(F)$.

Proposition 4.5.4 *Let F be any fixpoint of $\Gamma\Gamma^R_{\mathcal{P}}$. For each literal L*

$$L \in F \Leftrightarrow not\ L \notin \Gamma^R(F)$$

Clearly we can reason by contradiction and obtain the following result.

Corollary 4.5.1 *Let F be any fixpoint of $\Gamma\Gamma^R_{\mathcal{P}}$. For each literal L*

$$L \notin F \Leftrightarrow not\ L \in \Gamma^R(F)$$

The results of Proposition 4.5.4 and Corollary 4.5.1 also applies to the well founded model.

Let C be any conjunction or disjunction of literals. From Proposition 4.5.4 and Corollary 4.5.1 it follows that

$$WFDy(\mathcal{P}) \models C \ \Leftrightarrow\ \Gamma^R(WFDy(\mathcal{P})) \not\models not\ C$$

and

$$WFDy(\mathcal{P}) \ \not\models C \ \Leftrightarrow\ \Gamma^R(WFDy(\mathcal{P})) \models not\ C$$

These results constitute a valid tool to prove properties of $WFDy$. In the following we see two very simple but significant consequences of the results above.

Proof of Proposition 4.5.2:

let W be the well founded model of $\mathcal{P}$. Given a literal L, form Corollary 4.5.1 and from the fact that W is two valued interpretation, we know that

$$L \in \Gamma^R(W) \Leftrightarrow not\ L \notin W \Leftrightarrow L \in W$$

Then $W = \Gamma^R(W)$, and then, by definition, W is a refined model of $\mathcal{P}$. By Proposition 4.5.1 W is a subset of each stable model, but since W is two valued, W coincide with each stable model, hence W is the unique stable model of $\mathcal{P}$. ♢

In the well founded semantics for normal logic programs, given the least fixpoint W of the $\Gamma\Gamma$ operator, the set of negative conclusion is obtained computing $not(\mathcal{H}_P \setminus \Gamma(W))$. In $WFDy$ such conclusion are derived directly from the least fixpoint. Nevertheless it is easy to prove that such conclusion are the same obtained in the "normal logic programs" way.

Proposition 4.5.5 *Let F be a fixpoint of $\Gamma\Gamma^R_{\mathcal{P}}$. For each negative literal not A*

$$not\ A \in F \Leftrightarrow not\, A \in \mathcal{H}_P \setminus \Gamma^R(F)$$

proof This proposition cames as a particular case of Proposition 4.5.4. ♢

In the case of normal logic programs, a fixpoint F of the $\Gamma\Gamma$ operator is called a partial stable models iff $F \subseteq \Gamma(F)$. The well founded model is the least partial stable model. When negation in the head is possible, as for ELPs, since two different anti monotonous operators are used, the definition has to be refined, and F is a partial stable model iff $F \subseteq \Gamma^R(F)$. In this case the well founded model W is also a partial

stable model iff W is consistent i.e. it contains no contradictions.

We prove similar results for DyLPs, but here we obtain a more general conclusion.

Definition 42 *Let $\mathcal{P}$ be any Dynamic Logic Program. A set of literals S is a partial stable model iff it is a fixpoint of $\Gamma\Gamma^R$ and $S \subseteq \Gamma^R(S)$.*

It turns out that partial stable models are the consistent fixpoints.

Proposition 4.5.6 *Let F be a fixpoint of $\Gamma\Gamma^R$. F is a partial stable model if and only if it is consistent.*

proof Let F be any fixpoint of $\Gamma\Gamma^R$. Let us further suppose F is consistent. Then $L \in F$ implies *not* $L \notin F$. Then by Corollary 4.5.1 $L \in \Gamma^R(F)$.

Let us suppose now F is a subset of $\Gamma^R(F)$. Then $L \in F$ implies $L \in \Gamma^R(F)$. Then by Corollary 4.5.1 *not* $L \notin \Gamma^R(F)$. Since this follows for every L, F is consistent by definition. ♢

Since W is a subset of each fixpoint of $\Gamma\Gamma^R$, if W is not consistent there are no consistent fixpoints, i.e. no partial stable models.

If a *DyLP* has one update P it is a generalized logic program. Moreover if P contains no rule with a negative literal in the head it is a normal logic program. In the next Section we will show that Definition 41 coincides with the, respectively, the transformational well founded and the well founded of generalized and normal logic programs.

4.5.2 Characterization of consistent DyLPs

As stated in Section 4.3, *WFDy* is a paraconsistent semantics, i.e. it admits the possibility of having some inconsistent conclusions. In Section 4.3 we have defined the case when a program is consistent. Now we characterize the class of consistent DyLPs. Intuitively we show that the only situations where consistency is violated is when there are two conflicting rules in the same update which are supported, and none of them is rejected by some later update.

Theorem 4.5.3 *Let W be the well founded model of $\mathcal{P}$. W is consistent iff*

$$\forall \tau, \eta \in P_i,\ (\tau \bowtie \eta, W \models body(\tau), W \models body(\eta)) \Rightarrow$$

$$\exists \gamma \in P_j\ i < j \text{ such that } \gamma \bowtie \tau \text{ or } \gamma \bowtie \eta \text{ and } \Gamma^R(W) \models body(\gamma)$$

As for the refined semantics, it is possible to define an operational equivalent of Definition 41 by a program transformation of a DyLP into a normal logic program preserving the well founded semantics.

4.6 Relations with the well founded semantics of GLPs

In this section we prove that the well founded semantics for DyLPs coincides with the transformational well founded semantics of GLPs [DP96] when the considered DyLP is any single program P. In the single update case it is possible to simplify the definition of the set of default assumption. This gives a definition of the well founded semantics for GLPs that is purely declarative i.e. it does not use any kind of program transformation or extension of the language.

Let X be any set of literals and $\mathcal{H}$ the Herbrand base. In the following we use the notation $NH(X)$ for $not\,(\mathcal{H} \setminus X)$.

Definition 43 *Let P be any GLP. The well founded model $WFG(P)$ of P is the least fixpoint of the monotonous operators $\Gamma_P^G \Gamma_P^{GR}$ where Γ_P^G and Γ_P^{GR} are so defined:*

$$\Gamma_P^G(X) = least(P \cup NH(X))$$

$$\Gamma_P^{GR}(X) = least(P \setminus Rej^R(P,X)) \cup NH(X))$$

The following equivalence result holds.

Theorem 4.6.1 *Let P be any DyLP of one single update. The well founded semantics of P as a generalized logic program coincides with $WFDy(P)$.*

Moreover the semantics of Definition 43 is equivalent to the semantics proposed in [DP96].

Theorem 4.6.2 *The well founded semantics of generalized logic programs coincides with the transformational one.*

If the considered program P is a normal logic programs, by Theorem 2.4.6 we obtain as a corollary that the well founded semantics for DyLPs also generalizes the well founded semantics for normal logic programs.

Corollary 4.6.1 *Let P be any DyLP of one single update and let P be a normal logic program. The well founded semantics of P according to Definition 12 coincides with $WFDy(P)$.*

4.7 Transformational well founded semantics

The well founded transformation turns a given DyLP $\mathcal{P}$ in the language $\mathcal{L}$ into a normal logic program $\mathcal{P}^{TW}$ in an extended language $\mathcal{L}^W$ called the *well founded transformational equivalent of $\mathcal{P}$*.

Let $\mathcal{L}$ be a language. By $\mathcal{L}^W$ we denote the language whose atoms are either atoms of $\mathcal{L}$, or are atoms of one of the following forms: A^R A^{-S}, $rej(A,i)$, $rej(A^R,i)$, $rej(A^-,i)$,

and $rej(A^{-S}, i)$, where i is a natural number, A is any atom of $\mathcal{L}$ and no one of the atoms above belongs to $\mathcal{L}$. Given a conjunction of literals C, use the notation C^R for the conjunction obtained by replacing any occurrence of an atom A in C with A^R.

Definition 44 *Let $\mathcal{P}$ be a Dynamic Logic Program on the language $\mathcal{L}$. By the well founded transformational equivalent of $\mathcal{P}$, denoted $\mathcal{P}^{TW}$, we mean the normal program $P_1^W \cup \ldots \cup P_n^W$ in the extended language $\mathcal{L}^W$, where each P_i exactly consists of the following rules:*

Default assumptions *For each atom A of $\mathcal{L}$ appearing in P_i, and not appearing in any other P_j, $j \leq i$ the rules:*

$$A^- \leftarrow not\ rej(A^{-S}, 0) \qquad A^{-S} \leftarrow not\ rej(A^-, 0)$$

Rewritten rules *For each rule $L \leftarrow body$ in P_i, the rules:*

$$\overline{L} \leftarrow \overline{body},\ not\ rej(\overline{L}^R, i) \qquad \overline{L}^R \leftarrow \overline{body}^R,\ not\ rej(\overline{L}, i)$$

Rejection rules *For each rule $L \leftarrow body$ in P_i, a rule:*

$$rej(\overline{not\ L}, j) \leftarrow \overline{body}$$

where $j < i$ is the largest index such that P_j has a rule with head not L. If no such P_j exists, and L is a positive literals, then $j = 0$, otherwise this rule is not part of P_i^W.

Moreover, for each rule $L \leftarrow body$ in P_i, a rule:

$$rej(\overline{not\ L}^R, k) \leftarrow \overline{body}^R.$$

where $k \leq i$ is the largest index such that P_k has a rule with head not L. If no such P_j exists, and L is a positive literals, then $j = 0$, otherwise this rule is not part of P_i^W.

Finally, for each rule $L \leftarrow body$ in P_i, the rules:

$$rej(\overline{L}^R, j) \leftarrow rej(\overline{L}^R, i) \qquad rej(\overline{L}, j) \leftarrow rej(\overline{L}, i)$$

where $j < i$ is the largest index such that P_j also contains a rule $L \leftarrow body$. If no such P_j exists, and L is a negative literal, then $j = 0$, otherwise these rules are not part of P_i^W.

As the reader can see, the program transformation above resembles the one of Definition 38. The main difference is that the transformation of Definition 44 duplicates the language and the rules. This is done for simulating the alternate application of the two different operators, Γ and Γ^R, used in the definition of $WFDy$. The difference between these two operators is on the rejection strategies: the Γ^R operator allows rejection of rules in the same state, while the Γ operator does not. In the transformation above this difference is captured by the definition of the rejection rules. Let $L \leftarrow body$ be a rule in the update P_i. In the rules of the form $rej(\overline{L}, j) \leftarrow \overline{body}^R$ and $rej(\overline{L}^R, j) \leftarrow \overline{body}^R$, j is *less* than i, in the first case, and *less or equal* than i in the second one. A second difference

is the absence of the *totality constraints*. This is not surprising since the well founded model is not necessarily a two valued interpretation. Note however, that the introduction of any rule of the form $u \leftarrow not\, u,\ body$ would not change the semantics. As for the refined transformation, the atoms of the extended language $\mathcal{L}^W$ that do not belong to the original language $\mathcal{L}$ are merely auxiliary. Let $\mathcal{P}$ be any Dynamic Logic Program, P_i and update of $\mathcal{P}$, and let $\rho(\mathcal{P})^{Wi} = P_0^W \cup \ldots \cup P_i^W$.

Theorem 4.7.1 *Let $\mathcal{P}$ be any Dynamic Logic Program in the language $\mathcal{L}$, P_i and update of $\mathcal{P}$, and let $\rho(\mathcal{P})^{Wi}$ be as above. Moreover, Let W_i be the well founded model of the normal logic program $\rho(\mathcal{P})^{Wi}$ and $WFDy(\mathcal{P}^i)$ be the well found model of $\mathcal{P}$ at P_i. Then:*

$$WFDy(\mathcal{P}^i) = \{A \mid \ \in W_i\} \cup \{not\ A \mid A^- \in W_i\}$$

To compute the well founded semantics of a given DyLP $P_1, \ldots, P_n$ at a given state, it is hence sufficient to compute its well founded transformational equivalent $P_1^W, \ldots P_n^W$, then to compute the well founded model of the normal logic program $\rho(\mathcal{P})^{Wn}$ and, finally, to consider only those literals that belong to the original language of the program.

We present here a result analogous that of Theorem 3.11.2 that provides an upper bound to the size of the well founded transformational equivalent.

Theorem 4.7.2 *Let $\mathcal{P} : P_1, \ldots P_m$ be any finite ground DyLP in the language $\mathcal{L}$ and let $\rho(\mathcal{P})^{Wn}$ be the set of all the rules appearing in the transformational equivalent of $\mathcal{P}$. Moreover, let m be the number of clauses in $\rho(\mathcal{P})$ and l be the cardinality of $\mathcal{L}$. Then, the program $\rho(\mathcal{P})^{Wn}$ consists of at most $6m + l$ rules.*

The problem of computing the well founded model of a normal LP has a polynomial complexity [GRS91]. Hence, from Theorems 4.7.1 and 4.7.2, it follows that such a problem is polynomial also under the well founded semantics for DyLPs.

Moreover, the transformed programs are normal logic programs that can be queried by proof procedures like the one implemented in XSB-Prolog. This is the base of the implementation of the well founded semantics for DyLPs that can be found in appendix [Ban05b]. Since the well founded semantics for normal logic programs comply with the Property of Relevance (see [Dix95b]) in general only a subset of all the rules in the program must be processed in order to establish the truth value of a literal (or of a conjunction of literals).

4.8 Related works

The existing attempts in the literature (see [Lei97, APP+99]) for defining a well founded semantics are fallouts of the research for a stable model-like operational semantics for DyLPs based on causal rejection analogous to the program transformation of Section

3.11. The basic idea is to define the well founded model of a DyLP as the well founded model, limited to the language of the original DyLP, of the extended or generalized logic program obtained by the considered program transformation.

- In the master thesis [Lei97] the author defines a semantics for DyLPs where the various updates are extended, rather than generalized logic programs. A part from the basic choice of logic programs the defined semantics is equivalent to the justified updates semantics of [LP97]. This proposed semantics is based on a program transformation of a DyLP into an extended logic program and the model of a DyLP, according to this semantics, are the stable models of the transformed program (previous a limitation to the language of the original DyLP).

  Then the author proposed a notion of partial stable model as the partial stable models of the transformed program. Although not directly stated in the work, a notion of well founded model naturally follows as the least partial stable model.

- In [APP+99] the authors propose a three-valued interpretation of the transformational semantics of the update language LUPS [APPP02] (see Chapter 5 for a brief description). Since the semantics of LUPS is based on the the dynamic stable model semantics, the transformational semantics of LUPS also provide a transformational semantics for DyLPs.

The two semantics cited above are based on a complex syntactical transformation of DyLPs into single programs with many more rules and auxiliary literals than the original one, making it difficult to grasp what the declarative meaning of a sequence is. Moreover, being based on the operational equivalent of semantics for DyLPs different from the refined one, the obtained semantics share the same counter intuitive underlined for the related stable model-like semantics for DyLPs. In the case of stable model-like semantics based on causal rejection, the addition of tautologies or rules inducing self-dependencies among literals may introduce unwanted models. In the case of their well founded equivalents, the candidate well founded model fails to conclude some literals derived by the well founded semantics for DyLPs of Definition 41, i.e. the semantics proposed in [ALP+00b, APPP02, Lei97] are more skeptical than the well founded semantics for DyLPs.

For instance, in all the cited semantics, the candidate well founded model of the DyLP $\mathcal{P}'$ of example 4.4.2 fails to derive conclusions *not star* and *observe* [1] exactly as it happens for the corresponding stable model semantics.

[1] The semantics proposed in [Lei97], is based on extended LP, rather then on generalized LP, hence in the example the rule *not star* must be replaced with the fact ($\neg star$)

4.9 Concluding remarks

Guided by the potential needs of some application areas, in this chapter we defined a well founded semantics for DyLPs.

Our first step was to extend the causal rejection principle from two-valued to three-valued interpretations. This formally results in the definition of three-valued supported model.

On the basis of these concepts we defined a well founded semantics for DyLPs. The well founded semantics for DyLPs is defined as the least fixpoint of an monotonous operator obtained as the combination of the two anti monotonous operators used for the definition of the dynamic stable model and the refined semantics for DyLPs.

The well founded model is always defined, even for contradictory DyLPs, in which case, the well founded model is an inconsistent interpretation. Whenever the well founded model is not three-valued, it is a three-valued well supported update model.

The well founded model is a subset of any refined stable model and it coincides with the unique refine model whenever the well founded model is two-valued. On the other hand, there are DyLPs for which the well founded model is not a subset of the models according to the other stable model-like semantics based on causal rejection. Indeed, unlike these semantics (and like the refined one) the well founded semantics is not affected by the additions of tautologies in the DyLP.

The well founded semantics is a generalization of the well founded semantics of generalized (or normal) logic programs, and it coincides with them when the DyLP consists of a single generalized (or normal) logic program.

The computation of the well founded semantics has polynomial complexity. We provided an operational equivalent of the well founded semantics by a program transformation of a DyLP into a normal logic program preserving that preserves the well founded model. This transformation is the basis of an XSB-based implementation allowing to query a DyLP.

There are still a number of open questions on the subject.

A level mapping characterization of the well founded semantics In Chapter 3 we provided an alternative characterization of the refined semantics as the well supported model semantics which is in turn based on the notion of level mappings. Definition 40 extends the definition of supported model to the update three-valued setting and Theorem 4.5.1 states that the well founded model is a three-valued update models whenever it is consistent. Moreover, there exists an alternative definition of the well founded semantics of normal logic programs based on level mappings (see [HW05]). These facts suggest that it should possible to find an alternative characterization of Definition 41 based on the notion of level mappings.

Strong and weak properties of the well founded model In [Dix95a, Dix95b] the author proposes a list of properties for semantics of logic programs. One such property is that of Relevance (see [Dix95b]). Most of these properties are good for implementation reasons, others are more suited for testing whether a given semantics behaves intuitively. These properties are clearly satisfied by the transformational well founded equivalent of a DyLP according to Definition 44. Nevertheless it is not yet clear which of the properties listed in [Dix95a, Dix95b], or their equivalent for the update setting, hold in the case of the well founded semantics.

The well founded semantics for multidimensional dynamic logic programs. Definition 41 provides a well founded semantics for DyLPs. It remains to establish a well founded semantics for MDyLPs. As we argued in the end of Example 4.4.7 this extension could be particularly suitable for those applications requiring to merge and update knowledge from different sources. The well founded model is defined as the least fixpoint of the $\Gamma\Gamma^R$ where Γ and Γ^R are the operators used for defining, respectively, the dynamic and the refined stable model semantics for DyLPs. These two operators have been generalized for the multidimensional case by the homonymous operators of Definition 36 that extend the dynamic and refined semantics to the multidimensional case. A natural way to extend Definition 41 to MDyLPs would be to define the well founded model of an MDyLP as the least fixpoint of the composition of the two operators above. However, it is still unclear whether such a semantics would behave properly and extend the properties of the well founded semantics mentioned in the chapter to the multidimensional case.

Part II

Reasoning about and executing actions

Chapter 5

Reasoning about actions

Contents

5.1 Introduction **120**

5.2 An overview of LP updates and action description languages **122**

5.3 Evolving action programs **127**

5.4 Relationship to existing action languages **134**

5.5 Updates of action domains **136**

5.6 Conclusions and future work **140**

Having defined the basic semantics for dynamic logic programs, in this chapter we explore the knowledge representation and reasoning capabilities of updates languages specifying when and how to update a program and, in particular, we focus on Evolp [ABLP02] as the most simple, but yet very expressive logic programming updates language.

An important branch of investigation in the KR field has been the definition of high level languages for representing effects of actions, the programs written in such languages being usually called action programs.

In this chapter, we define evolving action programs as collections of macros of Evolp rules. Then, we provide translations of some of the most known action description languages into evolving action programs. This means that evolving action programs (and thus, in general, Evolp) are at least as expressive as these other action description languages and it can thus be used for reasoning about actions. We also underline some peculiar features of this newly defined paradigm. One of such features is that evolving action programs can easily express changes in the rules of the domains, including rules describing changes. Part of the results of this chapter have been published in [ABB04].

5.1 Introduction

While in the previous chapter we provided a meaning to logic program updates, described by semantics of logic programs (named dynamic logic programs) nothing has been said so far about how the various updating programs are generated, nor have the intrinsic knowledge representation capabilities of the framework been explored.

The first issue has been faced in the literature by defining *logic programming updates languages* (see [APPP02, EFST01, Lei03, ABLP02]), i.e. languages for specifying, by statements, how an existing program should be updated. The second issue is a whole area of exploration which, in our opinion, has been only touched by the existing research (see for instance [EFST01, Lei03, APP+99]). For instance, in [APP+99] the authors show, by examples, a possible usage of the LP updates language LUPS [APPP02] for representing effects of actions. We view this possibility as an important and promising area of research.

The concept of *action* is a key issue in AI fields like reactive systems and agents."*An agent is just something that acts*" [RN95]. Given the importance of the concept, ways of representing actions and their effects on the environment have been studied. A branch of investigation in this topic has been the definition of high level languages for representing effects of actions [BGP97, GL93, GLL+97, GLL+03], the programs written in such languages being usually called *action programs*. Action programs specify which facts (or fluents) change in the environment after the execution of a set of actions. Several works exist on the relation between these action languages and Logic Programming (e.g. [APP+99, GL93, Lif99]). However, despite the fact that LP has been successfully used as a language for declaratively representing knowledge, the mentioned works basically use LP for providing an operational semantics, and corresponding implementation, for action programs. This is so because normal logic programs, and most of their extensions, have no built in means for dealing with changes, something that is quite fundamental for action languages.

Even if the, already cited, "*Preliminary exploration* on actions as updates" [APP+99] provides a hint for the possibility of using LP updates languages as an action description paradigm, it does not provide a clear view on how to use LP updates for representing actions, and it does not establish an exact relationship between this new possibility and existing action languages. Thus, the possible advantages of the LP updates languages approach to actions are still not clear.

In this chapter we clarify these points through a more structural approach. Our investigation starts from the Evolp language [ABLP02]. The reason of our choice is the simplicity of the language and its semantics (which, unlike other languages for updates is based on a single command called *assert*). While Evolp is illustrated in this chapter, the discussion of the other LP updates languages is postponed to Chapter

6. The rationale for this choice is that a detailed discussion of these languages is not required here, while it is necessary in Chapter 6 where these languages are compared to ERA, a language extending Evolp newly defined here.

On top of Evolp we define a new action description language, called evolving action programs (EAPs), as a macro language for Evolp. More specifically, we define the syntax and the semantics of an action descriptions language based on LP updates. We then focus on the basic problem in the field, i.e. the *prediction* of the possible future states of the world given knowledge of the current state and the performed actions. Our purpose is to check, already at this early stage, the potentiality of an action description language based on the Evolp paradigm. However, a complete framework for action description would require more than that. It would require to establish a framework for *querying* an action program about various aspects of the environment described by the program and its possible evolutions. While the basic problem of prediction is addressed, other problems such as prediction with incomplete knowledge, post-diction and planning (both possibly with complete and incomplete knowledge) are not treated here and left for future work. Usually these querying problems are addressed by defining specific *action query languages* [GL98a]. We briefly discuss in Section 5.6 these other issues related to the framework of reasoning about actions.

We illustrate the usage of EAPs by an example involving a variant of the classical Yale Shooting Problem. An important point to clarify is the comparison of the expressive capabilities of the newly defined language with that of the existing paradigms. We consider the action languages $\mathcal{A}$ [GL93], $\mathcal{B}$ [GL98a] (which is a subset of the language proposed in [GLL$^+$97]), and (the definite fragment of) $\mathcal{C}$ [GLL$^+$03]. We provide simple translations of such languages into EAPs, hence proving that EAPs are *at least as expressive* as the cited action languages.

Coming to this point, the next natural question is what are the possible advantages of EAPs over the extant languages. The underlying idea of action frameworks is to describe dynamic environments. This is usually done by describing rules that specify, given a set of external actions, how the environment evolves. In a dynamic environment, however, not only the facts but also the "rules of the game" can change, in particular *the rules describing the changes*. The capability of describing such kind of *meta level changes* is, in our opinion, an important feature of an action description language. This capability can be seen as an instance of *elaboration tolerance*, i.e. *"the ability to accept changes to a person's or a computer's representation of facts about a subject without having to start all over"* [McC88]. In [GLL$^+$03] this capability is seen as a central point in the action descriptions field and the problem is addressed in the context of the $\mathcal{C}$ language. The final words of [GLL$^+$03] are *"Finding ways to further increase the degree of elaboration tolerance of languages for describing actions is a topic of future work"*. We address this topic in the context of EAPs and show EAPs are, in this sense, more flexible than other paradigms. Evolp provides specific commands that allow for the specification of

updates to the initial program, but also provides the possibility to specify updates of these updates commands. We illustrate, by successive elaborations of the well known Yale shooting problem, how to use this feature to describe updates of the problem that come along with the evolution of the environment.

The rest of the chapter is structured as follows. In Section 5.2 we briefly review existing action languages, we introduce the general framework of LP update languages and describe syntax and semantics of the language Evolp. In Section 5.3 we define the syntax and semantics of Evolp action programs, and we illustrate the usage of EAPs by an example involving a variant of the classical Yale Shooting Problem. In Section 5.4 we establish the relationship between EAPs and the languages $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$. In Section 5.5 we discuss the possibility of updating the EAPs, and provide an example of such feature. Finally, in Section 5.6, we conclude the chapter and trace a route for future developments in the usage of updates for specifying actions.

5.2 An overview of LP updates and action description languages

In this section we review existing work on logic programming updates languages (and, particularly, on Evolp) and on the action description languages $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$.

5.2.1 LP updates languages

Although the existing semantics for DyLPs provide a meaning to logic program updates up to this point in the thesis, nothing has been said about how the various updates should be specified. This is the subject of *logic programming updates languages*, i.e. languages specifying commands for updating logic programs. The common idea to all these languages is that an initial program is updated by rules specified through *statements* or containing *commands* to be executed. The statements can be *external* representing instructions passed by the programmer in order to update the program or *internal*, representing instances of *self evolution* of the programs.

The language *LUPS* [APPP02] is, chronologically speaking, the first of these languages and represents the archetype of such languages, having inspired the syntax and semantics of other languages. A statement in *LUPS* is has the form

$$Com(L \leftarrow B) \ when \ Cond.$$

where *Com* is a Command having a rule $L \leftarrow B$ has an argument and specifies how the current program must be changed (for instance, the rule taken as an argument can be added by the command *assert* or removed by the command *retract* from the current program) and *Cond* is a condition that has to be satisfied in order to trigger the com-

mand. Syntactically, condition *Cond* is a conjunction of literals whose truth value it evaluated on the basis of the current knowledge base. *EPI* [EFST01] is a language very similar to *LUPS* and it extends its syntax by adding external observations to the condition for executing a command. In other words, the execution of a command may be conditioned both to the satisfaction of some conditions internal to the program and to other conditions referring to the external environment. The language *EPI* introduced the concept of *sensibility* to the external environment. The language *KUL* [Lei03] basically extends the syntax of the commands of *LUPS* and it refines the semantics of the language by correcting some counterintuitive behavior. In all these three languages command statements are assumed to be external instructions imparted to the program in order to modify it.

This assumption is outmoded in *KABUL* [Lei03] where statements can be either external but they can also belong to a self update program. Syntactically, *KABUL* extends *KUL* by allowing to tests a much wider pattern of conditions. For instance it allows to test whether other commands are concurrently executed, or whether some rules are already in the program or not, and finally to test external observations (as in the language *EPI*). Moreover, also the pattern of possible commands of *KUL* is utterly extended.

Although *KABUL* (which stands for Knowledge And Behavior Update Language) already incorporates update statements inside the as part of the program there is a persistent dichotomy between the *knowledge part* i.e. represented by logic programming rules organized as a DyLP, and the *behavior part*, i.e. represented by a set of statements. This dichotomy is outmoded in Evolp where the update statements are merged within the knowledge base and where the statements are syntactically similar to inference rules. This homogeneity allows the use of the update mechanism of DyLPs also to update command statements rather than simply rules, thus enabling to update the update statements themselves.

If, in the line of evolution from *LUPS* to *KABUL*, the syntax of commands becomes more and more rich and complex, in Evolp there is a complete inversion. Evolp is a minimalist language with a unique command *assert* taking a rule τ (which can mean an inference rule or a rule encoding a statement) as argument and whose intuitive meaning is "update the program with τ".

An Evolp program is influenced by external observations as well as from external instructions, both represented as sets of LP rules called *input programs*.

Even from such a schematic overview, it emerges how Evolp unifies a strong concept of evolution (by allowing self and external updates to inference rules and statements) with the simplest syntax and semantics. This is the reason we focused on Evolp as logic programming updates language. In the rest of the chapter we will show how, despite of its simplicity, Evolp captures the expressivity of several frameworks for reasoning about the effect of actions and how its updates capabilities allow to reason about

successive elaborations of a scenario by simply updating its description (i.e. without the need of rewriting completely the program and restarting a simulation).

In the next Section we will present Evolp with some more details. For a more detailed description of *LUPS*, *EPI*, *KUL* and *KABUL* we refer the reader to Section 6.5.3 of Chapter 6 where these languages are presented and compared to an extension of Evolp called ERA.

5.2.2 The language Evolp

Having defined the meaning of sequences of programs, we are left with the problem of how to come up with those sequences. Among the existing LP updates languages Evolp uses a particulary simple syntax, which extends the usual syntax of GLPs by introducing the special predicate $assert/1$. Given any language $\mathcal{L}$, the language $\mathcal{L}_{assert}$ is recursively defined as follows: every atom in $\mathcal{L}$ is also in $\mathcal{L}_{assert}$; for any rule τ over $\mathcal{L}_{assert}$, the atom $assert(\tau)$ is in $\mathcal{L}_{assert}$; nothing else is in $\mathcal{L}_{assert}$. An *Evolp program* over $\mathcal{L}$ is any GLP over $\mathcal{L}_{assert}$. An *Evolp sequence* is a sequence (or DyLP) of Evolp programs. The rules of an Evolp program are called *Evolp rules*. Intuitively an expression $assert(\tau)$ stands for "update the program with the rule τ". Notice the possibility in the language to nest an assert expression in another.

The intuition behind the Evolp semantics is quite simple. Given a stable model like semantics for DyLPs like, for instance, the previously existing semantics based on the causal rejection principle (see [ALP+00a, BFL99, EFST02b, Lei03, LP97] and Chapter 3), starting from the initial Evolp sequence $\mathcal{P}_m$ (with this notation, in the following, we denote a DyLP consisting of a sequence of m logic programs) we compute the set, $\mathcal{SM}(\mathcal{P}_m)$, of the models of $\mathcal{P}_m$ according to a semantics *Sem*. Then, for any element M in $\mathcal{SM}(\mathcal{P}_m)$, we update the initial sequence with the program P_{m+1} consisting of the set of rules τ such that the atom $assert(\tau)$ belongs to M. In this way we obtain the sequence $\mathcal{P}_m \oplus P_{m+1}$. Since $\mathcal{SM}(\mathcal{P}_m)$ contains, in general, several models we may have different lines of evolution. The process continues by obtaining the various $\mathcal{SM}(\mathcal{P}^{m+1})$ and, with them, various $\mathcal{P}^{m+2}$. Intuitively, the program starts at step 1 already containing the sequence $\mathcal{P}_m$. Then it updates itself with the rules asserted at step 1, thus obtaining step 2. Then, again, it updates itself with the rules asserted at this step, and so on. The evolution of any Evolp sequence can also be influenced by external updates. An external update is itself an Evolp program called *input program*. If, at a given step n, the programs receives the input program E_n, the rules in E_n are added to the last self update for the purpose of computing the stable models determining the next evolution but, in the successive step $n+1$, they are no longer considered. In the formal definition of the semantics of the language, we will assume that the basic semantics for DyLPs is the refined semantics. However, it would be possible to adapt Definition 46 below to any semantics for DyLPs *Sem* by simply substituting the expression "refined stable

model" with the model according to the semantics Sem[1].

Definition 45 *Let n and m be natural numbers. An* evolving interpretation *of length n, of an evolving logic program $\mathcal{P}_m : P_1, \ldots P_m$ over the language $\mathcal{L}_{assert}$ is any finite sequence $\mathcal{M} = M_1, \ldots, M_n$ of two valued interpretations over $\mathcal{L}_{assert}$. The* evolution trace *associated with $\mathcal{M}$ and $\mathcal{P}$ is the sequence $\mathcal{P}_m \oplus \mathcal{T} = P_1, \ldots P_m, T_1 \ldots, T_{n-1}$, where*

$$T_i = \{\tau \mid assert(\tau) \in M_i\}$$

for $1 \leq i < n$

Definition 46 *Let $\mathcal{P}_m$ and $\mathcal{E}^n$ be any Evolp sequences, and $\mathcal{M} = M_1, \ldots, M_n$ be an evolving interpretation. Let $\mathcal{P}_m \oplus \mathcal{T}$ be the evolution trace associated with $\mathcal{M}$ and $\mathcal{P}$. We say that $\mathcal{M}$ is an evolving stable model of $\mathcal{P}_m$ with the sequence of input programs $\mathcal{E}^n$ iff M_k is a refined stable model of the program $\mathcal{P}_m \oplus (T_1, \ldots, (T_k \cup E_k))$ for any k, with $1 \leq k \leq n-1$.*

5.2.3 Action languages

The purpose of an action language is to provide ways of describing how an environment evolves given a set of external actions. A specific environment that can be modified through external actions is called an *action domain*. To any action domain we associate a pair of sets of atoms $\mathcal{F}$ and $\mathcal{A}$. We call the elements of $\mathcal{F}$ *fluent atoms* or simply *fluents*, and the elements of $\mathcal{A}$ *action atoms* or simply *actions*. Basically, the fluents are the observables in the environment and the actions are, clearly, the external actions. A *fluent literal* (resp. *action literal*) is an element of $\mathcal{F}_{Lit}$ (resp. an element of $\mathcal{A}_{Lit}$). In the following, we will use the letter Q to denote a fluent atom, the letter F to denote a fluent literal, and the letter A to denote an action atom. A *state of the world* (or simply a *state*) is any interpretation over $\mathcal{F}$. We say a fluent literal F is true at a given state s iff F belongs to s. Given a conjunction of fluent literals Cond we say *s satisfies Cond ($s \models Cond$) iff $Cond \subseteq s$*.

Each action language α provides ways to describe action domains through sets of expression called *action programs*. Usually, the semantics of an action program written in a language α is defined in terms of a *transition system*, i.e. a function whose argument is any pair (s, K), where s is a state of the world and K is a subset of $\mathcal{A}$, and whose value

[1]Indeed, Evolp was originally proposed in [ABLP02] using the dynamic stable model semantics [Lei03] rather than the refined stable model one as the basic semantics for DyLPs. Another difference between the language proposed herein and the original language of [ABLP02] is that in the original proposal the initial Evolp *sequence* was always a single Evolp program rather than a sequence of programs. Finally, in the present work, external updates to an Evolp sequence are called "input programs" while in [ABLP02] they where called "events". We opted for a different terminology in order to create no confusion when, in Chapter 6, we define the event-condition-action language ERA which extends Evolp and where the word "event" is used with a different meaning, which is the one it has in the literature on this kind of languages

is any set of states of the world. Intuitively, given the current state of the world, the semantics of a program written in the language α is given by a transition system that specifies which are the possible *resulting states* (according to the semantics of α) after simultaneously performing all actions in K.

Two kinds of expressions that are common within action description languages are *static and dynamic rules*. The *static rules* basically describe the rules of the domain, while *dynamic rules* describe effects of actions. A dynamic rule has a set of *preconditions*, namely conditions that have to be satisfied in the present state in order to have a particular effect in the future state, and *postconditions* describing such an effect. In the following we will consider three existing action languages, namely: $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$.

The language $\mathcal{A}$ [GL98a] is very simple. It only allows dynamic rules of the form

$$A \textbf{ causes } F \textbf{ if } Cond$$

where *Cond* is a conjunction of fluent literals. Such a rule intuitively means: performing the action A causes F to be true in the next state if *Cond* is true in the current state. The language $\mathcal{B}$ [GL98a] is an extension of $\mathcal{A}$ which also considers static rules. In $\mathcal{B}$, static rules are expressions of the form

$$F \textbf{ if } Body$$

where *Body* is a conjunction of fluent literals which, intuitively, means: if *Body* is true in the current state, then F is also true in the current state. A fundamental notion, in both $\mathcal{A}$ and $\mathcal{B}$, is *fluent inertia* [GL98a]. A fluent F is inertial if its truth value is preserved from a state to another, unless it is changed by the (direct or indirect) effect of an action. Hereafter a program written in the language $\mathcal{B}$ will be called a $\mathcal{B}$ program.

Given a state s and a set of actions K, we first consider the set $D(s,K)$ of fluents literals that are true as a (direct) consequence of actions i.e. any literal F that is the head of a dynamic rule A **causes** F **if** *Cond* such that $A \in K$ and *Cond* is true in s. Then s' is a possible resulting state, (according to the semantics of $\mathcal{A}$) from s iff any fluent literal in s is an element of $D(s,K)$ or is a true literal in s or is a consequence of a static rule.

Given any static rule r, F **if** *Body*, we say $F \leftarrow Body$ is the logic programming equivalent of r. Given any set of static rules $\mathcal{R}$, we define $\mathcal{R}^{LP}$ as the set of rules given by the LP equivalents of all the rules in $\mathcal{R}$.

Definition 47 *Let P be any $\mathcal{B}$ program with set of fluents $\mathcal{F}$, $\mathcal{R}$ the set of all the static rules of P, s a state and K any set of actions. Let $D(s,K)$ be the following set of literals*

$$D(s,K) = \{F : \exists\, A \textbf{ causes } F \textbf{ if } Cond \in P \; s.t. \; A \in K \wedge s \models Cond\}$$

A state s' is a resulting state (according to the semantics of $\mathcal{B}$) from s given P and the set of actions K iff

$$s' = least\left(\left(s \cap s'\right) \cup D(s,K) \cup \mathcal{R}^{LP}\right)$$

For a detailed explanation of $\mathcal{A}$ and $\mathcal{B}$ see [GL98a].

Static and dynamic rules are also the ingredients of the action language $\mathcal{C}$ [GLL$^+$03, GL98b]. Static rules in $\mathcal{C}$ are of the form

$$\textbf{caused } J \textbf{ if } H \tag{5.1}$$

while dynamic rules are of the form

$$\textbf{caused } J \textbf{ if } H \textbf{ after } O \tag{5.2}$$

where J and H are formulas such that any literal in them is a fluent literal, and O is any formula such that any literal in it is a fluent or an action literal. The formula O is the precondition of the dynamic rule and the static rule **caused** J **if** H is its postcondition.

Definition 48 *An action program written in $\mathcal{C}$ is any set of static and dynamic rules of the form, respectively, 5.1 and 5.2.*

The semantic of $\mathcal{C}$ is based on *causal theories* [GLL$^+$03]. Causal theories are sets of rules of the form **caused** J **if** H, each such rule meaning: If H is true this is an explanation for J. A basic principle of causal theories is that something is true iff it is caused by something else.

Definition 49 *Let P be any action program written in $\mathcal{C}$, let s be any state and K any set of actions, and let T be the causal theory given by the static rules of P and the post conditions of the dynamic rules of P whose preconditions are true in $s \cup K$. Then s' is a possible resulting state (according to the semantics of $\mathcal{C}$), iff it is a causal model of T.*

5.3 Evolving action programs

As we have seen, Evolp and action description languages share the idea of a system that evolves. In both, the evolution is influenced by external inputs (respectively, updates and actions). Evolp is actually a programming language devised for representing any kind of computational problem, while action description languages are devised for the specific purpose of describing actions. A natural idea is then to develop special kind of Evolp sequences for representing actions, and then compare such kind of programs with existing action description languages. We will call this kind of programs *evolving action programs* (EAPs).

Following the underlying notions of Evolp, we use the basic construct *assert* for defining special-purpose macros. As it happens with other action description languages, EAPs are defined over a set of fluents $\mathcal{F}$ and a set of actions $\mathcal{A}$. In EAPs, a state of the world is any interpretation over $\mathcal{F}$. To describe action domains we use an initial Evolp sequence I, D. The Evolp program D contains the description of the environment, while I contains some initial declarations, as it will be clarified later. As in $\mathcal{B}$ and $\mathcal{C}$, EAPs contain static and dynamic rules.

A *static rule* over $(\mathcal{F}, \mathcal{A})$ is simply an Evolp rule of the form

$$F \leftarrow Body.$$

where F is a fluent literal and *Body* is a set of fluent literals.

A *dynamic rule* over $(\mathcal{F}, \mathcal{A})$ is a (macro) expression

$$\textbf{effect}(\tau) \leftarrow Cond.$$

where τ is any static rule of the form $F \leftarrow Body$ and *Cond* is any set of fluent or action literals. The intuitive meaning of such a rule is that the static rule τ has to be considered *only* in those states whose predecessor satisfies condition *Cond*. Since some of the conditions literals in *Cond* may be action atoms, such a rule may describe the effect of a given set of actions under some conditions. Such an expression stands for the following set of Evolp rules (where τ is the short-form for $F \leftarrow Body$):

$$F \leftarrow Body,\ event(\tau). \tag{5.3}$$

$$assert(event(\tau)) \leftarrow Cond. \tag{5.4}$$

$$assert(not\ event(\tau)) \leftarrow event(\tau), not\ assert(event(\tau)) \tag{5.5}$$

where $event(F \leftarrow Body)$ is a new literal.

Let us see how the above set of rules fits with its intended intuitive meaning. Rule (5.3) is not applicable whenever $event(F \leftarrow Body)$ is false. If at some step n, the conditions *Cond* are satisfied, then, by rule (5.4), $event(F \leftarrow Body)$ becomes true at step $n+1$. Hence, at step $n+1$, rule (5.3) will play the same role as static rule $F \leftarrow Body$. If at step $n+1$ *Cond* is no longer satisfied, then, by rule (5.5) the literal $event(F \leftarrow Body)$ will become false again, and then rule (5.3) will be again not effective.

Besides static and dynamic rules, we still need another ingredient to complete our construction. As we have seen in the description of the $\mathcal{A}$ and $\mathcal{B}$ language, a notable concept is fluent inertia. This idea is not explicit in Evolp where *the rules* (and not the fluents) are preserved by inertia. Nevertheless, we can show how to obtain fluent inertia by using macro programming in Evolp. An *inertial declaration* over $(\mathcal{F}, \mathcal{A})$ is a (macro) expression

$$\textbf{inertial}(\mathcal{K})$$

where $\mathcal{K} \subseteq \mathcal{F}$. The intended intuitive meaning of such an expression is that the fluents in $\mathcal{K}$ are inertial. Before defining what this expression stands for, we state that the above mentioned program I is always of the form **initialize($\mathcal{F}$)**, where **initialize($\mathcal{F}$)** stands for the set of rules $Q \leftarrow prev(Q)$, where Q is any fluent in $\mathcal{F}$, and $prev(Q)$ are new atoms not in $\mathcal{F} \cup \mathcal{A}$. The *inertial declaration* **inertial($\mathcal{K}$)** stands for the set (where Q ranges over $\mathcal{K}$):

$$assert(prev(Q)) \leftarrow Q. \qquad assert(not\ prev(Q)) \leftarrow not\ Q.$$

Let us consider the behavior of this macro. If we do not declare Q as an inertial fluent, the rule $Q \leftarrow prev(Q)$ has no effect. If we declare Q as an inertial literal, $prev(Q)$ is true in the current state iff in the previous state Q was true. Hence, in this case, Q is true in the current state *unless* there is a static or dynamic rule that rejects such assumption. Viceversa, if Q was false in the previous state, then Q is true in the current one iff it is derived by a static or dynamic rule. We are now ready to formalize the syntax of evolving action programs.

Definition 50 *Let $\mathcal{F}$ and $\mathcal{A}$ be two disjoint sets of propositional atoms. An evolving action program (EAP) over $(\mathcal{F}, \mathcal{A})$ is any Evolp sequence I, D, where $I = initialize(\mathcal{F})$, and D is any set with static and dynamic rules, and inertial declarations over $(\mathcal{F}, \mathcal{A})$.*

Given an evolving action program I, D, the initial state of the world s (which, as stated above, is an interpretation over $\mathcal{F}$) is passed to the program together with the set K of the actions performed at s, as part of an input program. A resulting state, is the last element of any evolving stable model of I, D given the input program $s \cup K$ restricted to the set of fluent literals.

Definition 51 *Let I, D be any EAP over $(\mathcal{F}, \mathcal{A})$, and s a state of the world. Then s' is a resulting state (according to the semantics of EAPs) from s given I, D and the set of actions K iff there exists an evolving stable model M_1, M_2 of I, D given the sequence of input programs $s \cup K, \emptyset$ such that $s' \equiv_{\mathcal{F}} M_2$, where by $s' \equiv_{\mathcal{F}} \mathcal{M}_2$ we simply mean $s' \cap \mathcal{F}_{Lit} = M_2 \cap \mathcal{F}_{Lit}$.*

This definition can be easily generalized to sequences of set of actions.

Definition 52 *Let I, D be any EAP and s a state of the world. Then s' is a resulting state (according to the semantics of EAPs) from s given I, D and the sequence of sets of actions $K_1 \ldots, K_n$ iff there exists an evolving stable model $M_1, \ldots, M_{n+1}$ of I, D given the input program $(s \cup K_1), \ldots, K_n, \emptyset$ such that $s' \equiv_{\mathcal{F}} M_{n+1}$.*

Since EAPs are based on the Evolp semantics, which in turn is an extension of the stable model semantics for normal logic programs, it immediately follows that the complexity of the computation of the two semantics is the same.

Proposition 5.3.1 *Let I,D be any EAP over $(\mathcal{F},\mathcal{A})$, s a state of the world and $K \subseteq \mathcal{A}$. To find a resulting state s' from s given I,D and the set of actions K is an NP-hard problem.*

It is important to notice that, if the initial state s does not satisfies the static rules of the EAP, the correspondent Evolp sequence has no stable model, and hence there will be no successor state. This is, in our opinion, a good result: The initial state is just a state as any other. It would be strange if such state would not satisfy the rules of the domain. If this situation occurs, most likely either the translation of the rules, or the one of the state, have some errors. From now onwards we will assume that the initial state satisfies the static rules of the domain.

It is possible to provide an alternative characterization of the the possible resulting states.

Theorem 5.3.1 *Let I,D be any EAP, s a state of the world and K a set of actions. Let $\mathcal{R}$ be the set of static rules in D, I the following set of fluent literals*

$$I = \{Q \in \mathcal{F} : \textbf{\textit{inertial(Q)}} \in D\} \cup \{not\, Q : Q \in \mathcal{F} : \textbf{\textit{inertial(Q)}} \in D\}$$

and $D(s,K)$ be the following set of rules:

$$D(s,K) = \{\tau : \textbf{\textit{effect}}(\tau) \leftarrow Cond \in D \wedge K \cup s \models Cond\}$$

Then s' is a resulting state from s (according to the semantics of EAPs) given I,D and the set of actions K iff

$$s' = least\left((s \cap s' \cap I) \cup Default(s', \mathcal{R} \cup D(s,K))|_{(\mathcal{F} \setminus I)} \cup D(s,k) \cup \mathcal{R}\right) \tag{5.6}$$

In the extreme cases where the set of inertial fluents coincides with the whole set of fluents and, when the set of inertial fluents is empty, we obtain two simplifications of the equivalence 5.6.

Corollary 5.3.1 *Let I,D be any EAP, s a state of the world and K a set of actions. Let $\mathcal{R}$, $D(s,K)$ be as in Theorem 5.3.1. Moreover let every fluent be an inertial fluent. Then s' is a resulting state (according to the semantics of EAPs) from s given I,D and the set of actions K iff*

$$s' = least\left((s \cap s') \cup D(s,k) \cup \mathcal{R}\right)$$

proof Follows trivially as a special case of Theorem 5.3.1. ♢

Corollary 5.3.2 *Let I,D be any EAP, s a state of the world and K a set of actions. Let $\mathcal{R}$, $D(s,K)$ be as in Theorem 5.3.1. Moreover let the set of inertial fluents be the empty set. Then s' is a resulting state from s given I,D and the set of actions K iff s' is a stable model of the logic program $D(s,k) \cup \mathcal{R}$*

proof It follows trivially as a special case of Theorem 5.3.1 that

$$s' = least\left(Default(s', \mathcal{R} \cup D(s,K))|_{(\mathcal{F} \setminus I)} \cup D(s,k) \cup \mathcal{R}\right)$$

As proved in [Lei03] this amount to say s' is a stable model of $D(s,k) \cup \mathcal{R}$. ♢

Given an evolving action program I,D, the first basic problem is to make predictions of the possible evolutions given a complete knowledge of the initial state and the sequence of performed actions. Formally, let I,D be an EAP, let s be the current state of the world, and let $K_1,\ldots,K_n$ be a sequence of sets of actions. The problem is to determine the set J of resulting states from s given I,D and the sequence of sets of actions $K_1\ldots,\ K_n$. The answer to this problem is already entailed in Definition 52. First, the set $\mathcal{M}$ of evolving stable models of the form $M_1,\ldots,M_{n+1}$ of I,D given the input program $(s \cup K_1),\ldots,\ K_n,\emptyset$ is computed. Then, the resulting states are the last elements of the various sequences in $\mathcal{M}$ restricted to the set of fluents. Formally:

$$J = \{s' :\ \exists\, M_1,\ldots,M_{n+1} \in \mathcal{M}\ \wedge\ s' \equiv_{\mathcal{F}} M_{n+1}\}$$

It is clear from this formalization that several states can result from the execution of a sequence of sets of actions. These several states represent incomplete knowledge about the results of executing actions: the fluents belonging to all states are true, those belonging to no state are false and all the others are unknown. This behavior can be used also to deal with the problem of prediction with incomplete knowledge about the initial state. For this, just consider a complete initial state, and than add a first set of actions that, for each unknown fluent F, generates two subsequent states, one with F and another without F. Another possibility is to start with an action program whose models are exactly the possible initial states. It is easy to check that, to obtain this, it is sufficient, for instance, to add a fact for each fluent known to be true in the initial situation, no rules for fluents known to be false and the pair of rules

$$F \leftarrow not\, F. \qquad F_N \leftarrow not\, F.$$

for each fluent F (here F_N is an auxiliary atom not representing any fluent or action).

To illustrate EAPs, we now show an example of their usage by elaborating on probably the most famous example of reasoning about actions. The presented elaboration highlights some important features of EAPs, viz. the possibility of handling nondeterministic effects of actions, non-inertial fluents, non-executable actions, and effects of actions lasting for just one state.

5.3.1 An elaboration of the Yale shooting problem

In the original Yale shooting problem [SH87], there is a single-shot gun which is initially unloaded, and a turkey which is initially alive. One can load the gun and shoot the turkey. If one shoots, the gun becomes unloaded and the turkey dies. We consider a slightly more complex scenario where there are several turkeys, and where the shooting action refers to a specific turkey. Each time one shoots a specific turkey, one either hits and kills the bird, or misses it. Moreover, the gun becomes unloaded and there is a bang. It is not possible to shoot with an unloaded gun. We also add the property that any turkey moves iff it is not dead.

For expressing that an action is not executable under some conditions, we make use of a well known behavior of the stable model semantics. Suppose a given EAP contains a dynamic rules of the form

$$\textbf{effect(}u \leftarrow not\ u\textbf{)} \leftarrow Cond$$

where u is a literal which does not appear elsewhere (in the following, for representing such rules, we use the notation $\textbf{effect(}\bot\textbf{)} \leftarrow Cond$). With such a rule, if $Cond$ is true in the current state, then there is no resulting state. This happens because, as it is well known, programs containing $u \leftarrow not\ u$, and no other rules for u, have no stable models.

To represent the problem, we consider the fluents $dead(X)$, $moving(X)$, $hit(X)$, $missed(X)$, $loaded$, $bang$, plus the auxiliary fluent u, and the actions $shoot(X)$ and $load$ (where the Xs range over the various turkeys). The fluents $dead(X)$ and $loaded$ are inertial fluents, since their truth value should remain unchanged until modified by some action effect. The fluents $missed(X)$, $hit(X)$ and $bang$ are not inertial. The problem is encoded by the EAP I, D, where

$$I = \textbf{initialize(}dead(X), moving(X), missed(X), hit(X),\ loaded,\ bang,\ u\textbf{)}$$

and D is the following set of expressions

$$\begin{array}{l}
\textbf{inertial(}loaded\textbf{)} \\
\textbf{inertial(}dead(X)\textbf{)} \\
moving(X) \leftarrow not\ dead(X). \\
\textbf{effect(}\bot\textbf{)} \leftarrow shoot(X),\ not\ loaded. \\
\textbf{effect(}dead(X) \leftarrow hit(X)\textbf{)} \leftarrow\ shoot(X). \\
\textbf{effect(}loaded\textbf{)} \leftarrow load. \\
\textbf{effect(}hit(X) \leftarrow not\ missed(X)\textbf{)} \leftarrow shoot(X). \\
\textbf{effect(}bang\textbf{)} \leftarrow shoot(X). \\
\textbf{effect(}missed(X) \leftarrow not\ hit(X)\textbf{)} \leftarrow shoot(X). \\
\textbf{effect(}not\ loaded\textbf{)} \leftarrow shoot(X).
\end{array}$$

Let us analyze this EAP. The third rule encodes the impossibility to execute the action $shoot(X)$ when the gun is unloaded. The static rule $moving(X) \leftarrow not\, dead(X)$ implies that, for any turkey X, $moving(X)$ is true if $dead(X)$ is false. Since this is the only rule for $moving(X)$, it further holds that $moving(X)$ is true iff $dead(X)$ is false. Notice that declaring $moving(tk)$ as inertial, would result, in our description, in the possibility of having a moving dead turkey! This is why fluents $moving(X)$ have not been declared as inertial. Indeed, suppose we insert **inertial(**$moving(X)$**)** in the EAP above. Suppose further that $moving(tk)$ is true at state s, that one shoots at tk and kills it. Since $moving(tk)$ is an inertial fluent, in the resulting state $dead(tk)$ is true, but $moving(tk)$ remains true by inertia. Also notable is how effects that last only for one state, like the noise provoked by the shoot, are easily encoded in EAPs. The fourth and the sixth dynamic rules encode a non deterministic behavior: each shoot action can either hit and kill a turkey, or miss it.

To see how this EAP encodes the desired behavior of this domain, consider the following example of evolution. In this example, to lighten the notation, we omit the negative literals belonging to interpretations. Let us consider the initial state $\{\}$ (which means that all fluents are false). The state will remain unchanged until some action is performed. If one loads the gun, the program is updated with the input program $\{load\}$. In the unique successor state, the fluent $loaded$ is true and nothing else changes. The truth value of $loaded$ remains then unchanged (by inertia) until some other action is performed. The same applies to fluents $dead(X)$. The fluents $bang$, $missed(X)$, and $hit(X)$ remain false by default. If one shoots at a specific turkey, say Smith, and the program is updated with the input $shoot(smith)$, several things happen. First, $loaded$ becomes false, and $bang$ becomes true, as an effect of the action. Moreover, the rules:

$$\begin{aligned} hit(smith) &\leftarrow not\, missed(smith). \\ missed(smith) &\leftarrow not\, hit(smith). \\ dead(smith) &\leftarrow hit(smith). \end{aligned}$$

are considered as rules of the domain for one state. As a consequence, there are two possible resulting states. In the first one, $missed(smith)$ is true, and all the others fluents are false. In the second one, $hit(smith)$ is true, $missed(smith)$ is false and, by the rule $dead(smith) \leftarrow hit(smith)$, the fluent $dead(smith)$ becomes true. In both the resulting states, nothing happens to the truth value of the fluents $dead(X)$, $hit(X)$, and $dead(X)$ for $X \neq smith$.

5.4 Relationship to existing action languages

In this Section we show embeddings into EAPs of the action languages $\mathcal{B}$ and (the definite fragment of) $\mathcal{C}$[2]. We will assume that the considered initial states are consistent w.r.t. the static rules of the program, i.e. if the body of a static rule is true in the considered state, the head is true as well.

Let us consider first the $\mathcal{B}$ language. The basic ideas of static and dynamic rules are very similar in $\mathcal{B}$ and in EAPs. The main difference between the two is that in $\mathcal{B}$ *all* the fluents are inertial, whilst in EAPs only those that are declared as such are inertial. The translation of $\mathcal{B}$ into EAPs is then straightforward: All fluents are declared as inertial and then the syntax of static and dynamic rules is adapted. In the following we use, with abuse of notation, *Body* and *Cond* both for conjunctions of literals and for sets of literals.

Definition 53 *Let P be any action program in $\mathcal{B}$ with set of fluents $\mathcal{F}$. The translation $B(P,\mathcal{F})$ is the triple $(I^B, D^{BP}, \mathcal{F}^B)$ where: $\mathcal{F}^B = \mathcal{F}$, $I^B =$ **initialize($\mathcal{F}$)** and D^{BP} contains exactly the following rules:*

- ***inertial(Q)** for each fluent $Q \in \mathcal{F}$*
- *a rule $F \leftarrow Body$ for any static rule F **if** $Body$ in P.*
- *a rule **effect(F)** $\leftarrow A, Cond$. for any dynamic rule A **causes** F **if** $Cond$ in P.*

Theorem 5.4.1 *Let P be any $\mathcal{B}$ program with set of fluents $\mathcal{F}$, $(I^B, D^{BP}, \mathcal{F})$ its translation, s a state and K any set of actions. Then s' is a resulting state (according to the semantics of $\mathcal{B}$) from s given P and the set of actions K iff it is a resulting state (according to the semantics of EAPs) from s given I^B, D^{BP} and the set of actions K.*

proof It trivially follows from Corollary 5.3.1. ♢

We see now how there is a close relationship between *EAPs* and the $\mathcal{B}$ language. In practice *EAPs* generalize $\mathcal{B}$ by allowing both inertial and non inertial fluents and by admitting rules, rather then simply facts, as effects of actions.

Let us consider now the action language $\mathcal{C}$. Given a complete description of the current state of the world and performed actions, the problem of finding a resulting state is a problem of the satisfiability of a causal theory, which is known to be $\sum_P^2$-hard (cf. [GLL$^+$03]). So, this language belongs to a category with higher complexity than EAPs whose satisfiability is NP-hard. However, only a fragment of $\mathcal{C}$ is implemented and the complexity of such fragment is *NP*. This fragment is known as the

[2]The embedding of language $\mathcal{A}$ is not explicitly exposed here since $\mathcal{A}$ is a (proper) subset of the $\mathcal{B}$ language.

definite fragment of $\mathcal{C}$ [GLL+03]. In this fragment, static rules are expressions of the form **caused** F **if** *Body* where F is a fluent literal and *Body* is a conjunction of fluent literals, while dynamic rules are expressions of the form **caused** *not* F **if** *Body* **after** *Cond* where *Cond* is a conjunction of fluent or action literals[3]. For this fragment it is possible to provide a translation into EAPs.

The main problem of the translation of $\mathcal{C}$ into EAPs lies in the simulation of causal reasoning with stable model semantics. The approach followed here to encode causal reasoning with stable models is in line with the one proposed in [Lif99]. We need to introduce some auxiliary predicates and define a syntactic transformation of rules. Let $\mathcal{F}$ be a set of fluents, and let $\mathcal{F}^C$ denote the set of fluents $\mathcal{F} \cup \{Q_N \mid Q \in \mathcal{F}\}$. We add, for each $Q \in \mathcal{F}$, the constraints:

$$\leftarrow \quad not\ Q, not\ Q_N. \tag{5.7}$$

$$\leftarrow \quad Q,\ Q_N. \tag{5.8}$$

Let Q be a fluent and $Body = F_1, \ldots, F_n$ a conjunction of fluent literals. We will use the following notation: $\overline{Q} = not\ Q_N$, $\overline{not\ Q} = not\ Q$ and $\overline{Body} = \overline{F_1}, \ldots, \overline{F_n}$

Definition 54 *Let P be any action program in the definite fragment of $\mathcal{C}$ with set of fluents $\mathcal{F}$. The translation $C(P, \mathcal{F})$ is the tuple $(I^C, D^{CP}, \mathcal{F}^C)$ where: $\mathcal{F}^C$ is defined as above, $I^C \equiv$ **initialize**$(\mathcal{F}^C)$ and D^{CP} consists exactly of the following rules:*

- *a rule* **effect**$(Q \leftarrow \overline{Body}) \leftarrow Cond$, *for any dynamic rule in P of the form* **caused** Q **if** *Body* **after** *Cond;*
- *a rule* **effect**$(Q_N \leftarrow \overline{Body}) \leftarrow Cond$, *for any dynamic rule in P of the form* **caused** *not* Q **if** *Body* **after** *Cond;*
- *a rule $Q \leftarrow \overline{Body}$, for any static rule in P of the form* **caused** Q **if** *Body;*
- *a rule $Q_N \leftarrow \overline{Body}$, for any static rule in P of the form* **caused** *not* Q **if** *Body;*
- *The rules (5.7) and (5.8), for each fluent $Q \in \mathcal{F}$.*

For this translation we obtain a result similar to the one obtained for the translations of the $\mathcal{B}$ language.

Theorem 5.4.2 *Let P be any action program in the definite fragment of $\mathcal{C}$ with set of fluents $\mathcal{F}$, $(I^C, D^{CP}, \mathcal{F}^C)$ its translation, s a state, s^C the interpretation over $\mathcal{F}^C$ defined as follows:*

$$s^C = s \cup \{Q_N \mid Q \in s\} \cup \{not\ Q_N \mid not\ Q \in s\}$$

[3]The definite fragment defined in [GLL+03] is (apparently) more general, allowing *Cond* and *Body* to be arbitrary formulae. However, it is easy to prove that such kind of expressions are equivalent to a set of expressions of the form described above

and K any set of actions. Then s^* is a resulting state (according to the semantics of EAPs) from s^C given I^C, D^{CP} and the set of actions K iff there exists s' such that s' is a resulting state (according to the semantics of $\mathcal{C}$) from s, given P and the set K and $s^* \equiv_{\mathcal{F}} s'$.

proof By Corollary 5.3.2, s^* is a resulting state (according to the semantics of EAPs) from s^C given I^C, D^{CP} and the set of actions K iff s' is a stable model of the program $\mathcal{R} \cup D(s, K)$ where $\mathcal{R}$ and $D(s^C, K)$ are defined as in Theorem 5.3.1. From the translation of definite causal theories into logic programs presented in [GLL$^+$03], it follows that this is equivalent to say s' is a model of the causal theory obtained by all the static rules of P plus the rules of the form **caused** J **if** H such that the dynamic

caused J **if** H **after** O

belongs to P, and Q is true in $s \cup K$, this, in turn, is equivalent to say that s' is a resulting state (according to the semantics of $\mathcal{C}$) from s given P and the set of actions K as desired. ♢

By showing translations of the action languages $\mathcal{B}$ and the definite fragment of $\mathcal{C}$ into EAPs, we proved that EAPs are *at least as expressive* as such languages. Moreover, the translations above are quite simple: basically one EAP static or dynamic rule for each static or dynamic rule in the other languages. The next natural question is: Are EAPs *more expressive*?

5.5 Updates of action domains

Action description languages describe the rules governing a domain where actions are performed, and the environment changes. In practical situations, it may happen that the very rules of the domain change with time too. When this happens, it would be desirable to have ways of specifying the necessary updates to the considered action program, rather than to have to write a new one. EAPs are just a particular kind of Evolp sequences. As in general Evolp sequences, they can be updated by input programs.

When one wants to update the existing rules with a rule τ, all that has to be done is to add the fact $assert(\tau)$ as an input program. This way, the rule τ is asserted and the existing Evolp sequence is updated. Following this line, we extend EAPs by allowing the input programs to contain facts of the form $assert(\tau)$, where τ is an Evolp rule, and we show how they can be used to express updates to EAPs. For simplicity, below we use the notation $assert(R)$, where R is a set of rules, for the set of expressions $assert(\tau)$ where $\tau \in R$.

To illustrate how to update an EAP, we come back to the example of Section 5.3.1.

Let I, D be the EAP defined in that Section. Let us now consider that after some shots, and dead turkeys, rubber bullets are acquired. One can now either load the gun with normal bullets or with a rubber bullet, but not with both. If one shoots with a rubber loaded gun, the turkey is not killed.

To describe this change in the domain, we introduce a new inertial fluent representing the gun being loaded with rubber bullets. We have to express that, if the gun is rubber-loaded, one cannot kill the turkey. For this purpose we introduce the new macro:

$$not\ \textbf{effect(}F \leftarrow Body\textbf{)} \leftarrow Cond.$$

where F, is a fluent literal, $Body$ is a set of fluents literals and $Cond$ is a set of fluent or action literals. We refer to such expressions as *effects inhibitions*. This macro simply stands for the rule

$$assert(not\ event(F \leftarrow Body)) \leftarrow Cond.$$

where $event(F \leftarrow Body)$ is as before. The intuitive meaning is that, if the condition $Cond$ is true in the current state, any dynamic rule whose effect is the rule $F \leftarrow Body$ is ignored.

To encode the changes described above, we update the EAP with the input program E_1 consisting of the facts $assert(I_1)$ where $I_1 = (\textbf{initialize(}rubber\_loaded\textbf{)})$. Then, in the subsequent state, we update the program with the external update $E_2 = assert(D_1)$ where D_1 is the set of rules:[4]

$$\begin{aligned}
&\textbf{inertial(}rubber_loaded\textbf{)}.\\
&\textbf{effect(}rubber_loaded\textbf{)} \leftarrow rubber_load.\\
&\textbf{effect(}not\ rubber_loaded\textbf{)} \leftarrow shoot(X).\\
&\textbf{effect(}\bot\textbf{)} \leftarrow rubber_loaded,\ load.\\
&\textbf{effect(}\bot\textbf{)} \leftarrow loaded,\ rubber_load.\\
&not\ \textbf{effect(}dead(X) \leftarrow hit(X)\textbf{)} \leftarrow rubber_loaded.
\end{aligned}$$

Let us analyze the proposed updates. First, the fluent *rubber_loaded* is initialized. The newly introduced fluent is declared as inertial, and two dynamic rules are added specifying that load actions are not executable when the gun is already loaded in a different way. Finally we use the new command to specify that the effect $dead(X) \leftarrow hit(X)$ does not occurs if, in the previous state, the gun was loaded with rubber bullets. Since this update is more recent than the original rule $\textbf{effect(}dead(X) \leftarrow hit(X)\textbf{)} \leftarrow shoot(X)$, the dynamic rule is updated.

Basically updating the original EAP with the rule

$$not\ \textbf{effect(}dead(X) \leftarrow hit(X)\textbf{)} \leftarrow rubber\_loaded.$$

[4] In the remainder, we use $assert(U)$, where U is a set of macros (which are themselves sets of Evolp rules), to denote the set of all facts $assert(\tau)$ such that there exists a macro η in U with $\tau \in \eta$.

has the effect of adding *not rubber_loaded* to the preconditions of the dynamic rule

$$\textbf{effect(}dead(X) \leftarrow hit(X)\textbf{)} \leftarrow shoot(X).$$

So far we have shown how to update the preconditions of a dynamic rule. It is also possible to update static rules and the descriptions of effects of actions. Suppose the cylinder of the gun becomes dirty and, whenever one shoots, the gun may either work properly or fail. If the gun fails, the action *shoot* has no effect. We introduce two new fluents in the program with the input program $assert(I_2)$ where $I_2 = \textbf{initialize(}fails, work\textbf{)}$. Then, we assert the input program $E_2 = assert(D_2)$ where D_2 is the following EAP

$$\begin{aligned}
\textbf{effect(}fails \leftarrow not\ work\textbf{)} \leftarrow &\ shoot(X).\\
\textbf{effect(}work \leftarrow not\ fails\textbf{)} \leftarrow &\ shoot(X).\\
not\ missed(X) \leftarrow &\ fails.\\
not\ hit(X) \leftarrow &\ fails.\\
not\ bang \leftarrow &\ fails.\\
\textbf{effect(}loaded \leftarrow fails\textbf{)} \leftarrow &\ loaded.\\
\textbf{effect(}rubber_loaded \leftarrow fails\textbf{)} \leftarrow &\ rubber_loaded.
\end{aligned}$$

The first two dynamic rules simply introduce the possibility that a a failure may occur every time we shoot. The three static rules describe changes in the behavior of the environment when the gun fails which amount to negate what was entailed by static and dynamic rules in D. The last two dynamic rules updates updates two dynamic rules in D and D_1 respectively. These rules specify that, when a failure occurs, the gun remain loaded with the same kind of bullet. Since the new rules of D_2 are more recent then the rules in D and D_1 they update these last ones. This last example shows how to update static and dynamic rules with new static and dynamic rules. To better illustrate, we now show a possible evolution of the updated system. Suppose currently the gun is not loaded. One loads the gun with a rubber bullet, and then shoots at the turkey named Trevor. The initial state is $\{\}$. The first set of actions is $\{rubber\_load\}$ The resulting state after this action is $s' \equiv \{rubber\_loaded\}$. Suppose one performs the action $load$. Since the EAP is updated with the dynamic rule $\textbf{effect(}\bot\textbf{)} \leftarrow rubber\_loaded,\ load$. there is no resulting state. This happens because we have performed a non executable action. Suppose, instead, the second set of actions is $\{shoot(trevor)\}$. There are three possible resulting states. In one the gun fails. In this case, the resulting state is, again, s'. In the second, the gun works but the bullet misses Trevor. In this case, the resulting state is $s''_1 \equiv \{missed(trevor)\}$. Finally, the gun works and the bullet hits Trevor. Since the bullet is a rubber bullet, Trevor is still alive. In this case the resulting state is $s''_2 \equiv \{hit(trevor)\}$.

The input programs may introduce changes in the behavior of the original EAP.

This opens a new problem. Within classical action languages it is not necessary to keep trace of the *history* of the world. Indeed, given an action program written in either $\mathcal{B}$ or $\mathcal{C}$, by Definitions 47 and 49 it follows that the set of possible resulting states solely depends on the current state s and the set of actions K. In the case of EAPs the situation is different, since external updates can change the behavior of the considered evolving action program by updating the program itself. Fortunately, we do not have to care about the *whole* history of the world, but just about those input programs containing new initializations, inertial declarations, effects inhibitions, and static and dynamic rules.

It is possible to have a compact description of an EAP that is updated several times via input programs. For that we need to further extend the original definition of EAPs.

Definition 55 *An* updated *evolving action program over* $(\mathcal{F},\mathcal{A})$ *is any sequence* $I,D_1,\ldots,D_n$ *where* I *is* ***initialize***$(\mathcal{F})$*, and the various* D_k *are sets consisting of static rules, dynamic rules, inertial declarations and effects inhibitions such that any fluent appearing in* D_k *belongs to* $\mathcal{F}$*.*

Definition 56 *Let* $I,D_1,\ldots,D_n$ *be any* updated *EAP and* s *a state of the world. Then* s' *is a resulting state (according to the semantics of EAPs) from* s *given* $I,D_1,\ldots,D_n$ *and the sequence of sets of actions* $K_1\ldots,\ K_n$ *iff there exists an evolving stable model* $M_1,\ldots,M_n$ *of* $I,D_1,\ldots,D_n$ *given the sequence of input programs* $(s\cup K_1),\ldots,\ K_n,\emptyset$ *such that* $s'\equiv_{\mathcal{F}} M_n$*.*

In general, if we update an evolving action program I,D with the subsequent input programs $assert(I_1)$, $assert(D_1)$, where I_1,D_1 is another EAP, we obtain the equivalent updated evolving action program $(I\cup I_1),D,D_1$.

Theorem 5.5.1 *Let* $I_0\cup I,D_0,D_1,\ldots,D_k$ *be any update EAP over* $(\mathcal{F},\mathcal{A})$*. Let* $\bigoplus E_i^n$ *be a sequence of input programs such that:* $E_1=K_1\ \cup\ s$*, where* s *is any state of the world and* K_1 *is any set of actions, and the others* E_i*s are any set of actions* K_α*, or any set* $assert($***initialize***$(\mathcal{F}_\beta))$ *where* $\bigcup\mathcal{F}_\beta=I$*, or any set* $assert(D_i)$ *with* $1\leq i\leq k$*. Let* $s_1,\ldots,s_n$ *be a sequence of possible resulting states from* s *given the EAP* I_0,D_0 *and the sequence of input programs* $\bigoplus E_i^n$ *and* K_{n+1} *a set of actions. Then* $s_1,\ldots,s_n,s'$ *is a sequence of possible resulting states from* s *given* I_0,D_0 *and the sequence of input programs* $\bigoplus E_i^n,K_{n+1}$ *iff* s' *is a resulting state from* s_n *given* $I_0\cup I,D_0,D_1,\ldots,D_k$ *and the set of actions* K_{n+1}*.*

For instance, the updates to the original EAP of the example in this Section are equivalent to the updated EAP I_{sum},D,D_1,D_2, where $I_{sum}\equiv I\ \cup\ I_1\ \cup\ I_2$, I and D are as in the example of Section 5.3.1, and the I_is and D_is are as described above.

Yet one more possibility opened by updated Evolp action programs is to cater for successive elaborations of a program. Consider an initial problem described by an EAP I,D. If we want to describe an elaboration of the program, instead of *rewriting* I,D we can simply *update* it with new rules. This gives a new answer to the problem

of elaboration tolerance [McC88] and also opens the new possibility of *automatically update* action programs by other action programs.

The possibility to elaborate on an action program is also discussed in [GLL+03] in the context of the $\mathcal{C}$ language. The solution proposed there, is to consider $\mathcal{C}$ programs whose rules have one extra fluent atom in their bodies, all these extra fluents being false by default. The elaboration of an action program P is the program $P \cup U$ where U is a new action program. The rules in U can defeat the rules in P by changing the truth value of the extra fluents. An advantage of EAP over that approach is that, in EAPs the possibility of updating rules is a built-in feature rather then a programming technique involving manipulation of rules and introduction of new fluents. Moreover, in EAPs we can simply encode the new behavior of the domain by new rules and then let these new rules update the previous ones.

5.6 Conclusions and future work

In this chapter we have explored the possibility of using logic programs updates languages as action description languages. In particular, we have focused our attention on the Evolp language. As a first point, we have defined a new action language paradigm, christened evolving action programs, defined as a macro language over Evolp. We have provided an example of usage of this language, and compared evolving action programs with action languages $\mathcal{A}$, $\mathcal{B}$ and the definite fragment of $\mathcal{C}$, by defining simple translations into Evolp of programs in these languages. Finally, we have also shown and argued about the capability of EAPs to handle changes in the domain during the execution of actions.

The goal of the thesis is to design a framework for developing AI application capable of reasoning and react to events by executing actions. The last two topics are not yet touched and will be treated in the sequel, in Chapter 6. Since this was the main goal, some other issues, briefly discussed below and related to action descriptions, are not further explored here, and are left open for future research.

Action description programs, as we have just seen, define functions that, given one state and a set of actions, determine a state that results from (concurrently) executing all the action from the initial state. These functions can be used in several ways, resulting in solving different problems in the domain of actions. One of these problem is that of *prediction*, i.e. of determining the resulting set of states given an initial state and a sequence of sets of actions. In this Chapter we illustrated the use of EAPs for solving prediction problems. This amounted to encode the initial knowledge in the initial EAP, and then subsequently update it with the sets of actions. The resulting state corresponds then to the stable models of the updated EAP.

Another problem is that of *postdiction* [McC59], i.e. to determine, given a final state s and a sequence of sets of actions $K_1, \ldots, K_n$, the set J of initial states such that given

any state s in J and the sequence of sets of actions $K_1, \ldots, K_n$, the final resulting state is s'. A generalization of this problem is that of postdiction with incomplete knowledge, i.e. to determine the set J of possible initial states but having only a partial knowledge of the final state s'.

Yet another problem studied in the literature of action languages is that of *planning* [McC59], i.e. do determine, given a complete initial state and a goal (i.e. a condition we want to satisfy), a sequence of sets of actions that, if executed, brings to a state satisfying the goal. Planning is even a more complex issue when the problem has elements of uncertainty either regarding the effect of actions or the initial state [ea00].

Of course in theory EAPs can deal with both these problems; e.g. for planning, just find one such sequence of sets of actions that satisfies the conditions according to the function defined by the action description program. However, contrary to what happens with prediction, we have not provided any method for coming up with these plans, except from the (non-satisfactory) generation and testing of all possible sequences, given a finite set of possible actions. The same happens for the case of postdiction. Methods for postdiction and planning based on EAPs are outside the scope of this work, and are left open for future research.

Closely related to this issue is that of *action query language* [GL98a]. Action query languages are formalisms for expressing knowledge about the various states under the form of axioms, and for formulating queries whose truth value has to be tested. According to the expressivity of the action query language, it is possible to formulate prediction, postdiction and planning, or even combinations of these problems by stating axioms (possessed knowledge about the evolution of the world) and queries (statements whose truth value has to be tested and expressing the knowledge that is required to be inferred). The definition of a query action language for EAPs and the development of techniques for inferring the truth value of possible queries is a complex issue, at least as complex as that of planning and postdiction with EAPs, and certainly deserves further research.

Yet another topic that can be further studied involves the possibility of concurrent execution of actions. Evolp already provides this possibility. Nevertheless, we have not fully explored this topic here. In particular we did not confronted the results with extant works [BG97, LL03]. One of the most crucial aspects regards the so called incremental fluents (see, for instance, [LL03]) fluents with a related numerical valued that can be increased or decreased. For instance, given a vat with 100 liters how to describe the effect of the two simultaneously executed actions “spill 5 liters into the vat" and “draw 7 liters from the vat?".

Chapter 6

Evolving Reactive Algebraic Programs

Contents

6.1 Introduction **144**

6.2 Outline and syntax of the language **147**

6.3 The Semantics of ERA **152**

6.4 Remarks **161**

6.5 Comparisons to other updates languages **164**

6.6 Comparisons to ECA formalisms and event languages **181**

6.7 Conclusions and open issues **188**

Having defined a framework for reasoning about the effects of actions, we are left with the goal of defining a paradigm also suitable for programming reactive behavior. Event-Condition-Action (ECA) languages are an intuitive and powerful paradigm for programming reactive systems. Usually, important features for an ECA language are reactive and reasoning capabilities, the possibility to express complex actions and events, and a declarative semantics. In this chapter, we introduce ERA, an ECA language based on the framework of logic programs updates that, together with these features, also exhibits capabilities to integrate external updates and perform self updates to its knowledge (data and inference rules) and behavior (active rules). ERA combines the logic framework of DyLPs with an event and an action algebra allowing to define complex events and actions. We show then that ERA extends the logic programming updates language Evolp, i.e. every Evolp program (and thus every EAP program) has a straightforward encoding in ERA preserving the semantics of the programs. Part of the results of this chapter have been published in [ABB06].

6.1 Introduction

In Chapter 5 we have shown that the language Evolp is a suitable formalism for writing formal descriptions of the effect of actions and reasoning about such effects. However, our goal is to define a single framework for reasoning about and *executing* actions. The natural question is then whether it is possible to use Evolp for programming reactive behavior. In particular we are interested in developing some form of *Event Condition Action* (ECA) formalism.

Event Condition Action languages are an intuitive and powerful paradigm for programming reactive systems. The fundamental construct of ECA languages are *active rules* of the form:

$$\textbf{On } \mathit{Event} \quad \textbf{If } \mathit{Condition} \textbf{ Do } \mathit{Action} \tag{6.1}$$

which mean: when *Event* occurs, if *Condition* is verified, then execute *Action*. ECA systems receive inputs (mainly in the form of *events*) from the external environment, test conditions in the current knowledge base, and, if they hold, react by performing actions that change the stored information (internal actions) or influence the environment itself (external actions). There are many potential and existing areas of applications for ECA languages such as active and distributed database systems [WC96a, BL96], Semantic Web applications [ABM+02, SCM+99], distributed systems [GNF98], Real-Time Enterprize and Business Activity Management and agents [CT04]. Particularly, in the field of active databases, ECA languages are already subject of applications and research [WC96b] and could become a key instrument for developing networks of communicating autonomous software systems. For instance, one of the goals of the REWERSE project [Rew] is the development of reactive autonomously acting and communicating Web services.

To be useful in a wide spectrum of applications an ECA language has to satisfy several properties. First of all, events occurring in an active rule can be complex, resulting from the occurrence of several basic ones. A widely used way for defining complex events is to rely on some event algebra [CL03, AC03], i.e., to introduce operators that define complex events as the result of compositions of more basic ones that occur at the same or at different instants. Actions that are triggered by active rules may also be complex operations involving several (basic) actions that have to be performed concurrently or in a given order and under certain conditions. The possibility to define events and actions in a compositional way (in terms of sub events and sub actions), permits a simpler and more elegant programming style by breaking complex definitions into simpler ones and by allowing the usage of the definition of the same entity in different fragments of code.

An ECA language would also benefit from a declarative semantics taking advan-

tage of the simplicity of its basic concepts. Moreover, an ECA language must in general be coupled with a knowledge base, which, in our opinion, should be richer than a simple set of facts, and allow for the specification of both relational data and classical rules, i.e. rules that specify knowledge about the environment, besides the ECA rules that specify reactions to events. Together with the richer knowledge base, an ECA language should exhibit inference capabilities in order to extract knowledge from such data and rules.

Clearly ECA languages deal with systems that evolve. However, in existing ECA languages this evolution is mostly limited to the evolution of the (extensional) knowledge base. But in a truly evolving system, that is able to adapt to changes in the considered domain, there can be evolution of more than the extensional knowledge base: derivation rules of the knowledge base (intensional knowledge), as well as the active rules themselves may change over time. We believe another capability that should be considered is that of *evolving* in this broader sense. Here, by evolving capability we mean that a program should be able to automatically integrate external updates and to autonomously perform self updates. The language should allow updates of both the knowledge (data and classical rules) and the behavior (active rules) of the considered ECA program, due to external and internal changes.

An utterly useful feature would be that of being naturally combinable with some actions description formalism. This would allow to program descriptions of both the reactive system and its environment and hence to prove formal properties of the integrated system or to run simulations.

To the best of our knowledge no existing ECA language provides all the above mentioned features (a detailed discussion is presented in Section 6.6). In particular, none provides the evolving capability, nor it is immediately clear how to incorporate such capability to these languages.

The purpose of this chapter is to define an ECA language based on logic programming that satisfies all these features. We already described the simple and declarative semantics of Evolp, its inference capabilities (provided by the underlying logic programming semantics), its evolving and self-evolving features and finally the possibility of being used (by the macro language EAPs) as an action description language. Thus, many of the features demanded above are already possessed by Evolp . A natural question is hence whether Evolp or, possibly, a slightly modified version of Evolp is suitable for the above enounced purpose.

We start by examining which of the listed desirable features for ECA frameworks[1] are not shared by Evolp .

First of all the semantics of Evolp proposed in Chapter 5 brings, in general, to

[1]namely: the capability to detect and react to basic and complex event, the capability to execute basic and complex actions, the capability to represent and reason about knowledge, the capability of self evolution, a declarative semantics, and the possibility of being combined with some action description formalism

several possible evolutions. This is a desirable feature when the goal is to reason about the possible outputs of the execution of actions but the objective of ECA languages is to *execute* actions rather than to simply reason about their effects. Moreover, unlike ECA paradigms, Evolp does not provide mechanisms for specifying the execution of external actions nor it has a formal notion of events in the sense of ECA formalism.

Nevertheless, it would be probably possible to introduce slight modifications to Evolp in order to achieve these desired features without altering the basic structure of the language. For instance, in [LS06] the authors explore the possibility to have external actions to Evolp .

What, in our opinion, is the main shortcoming of Evolp is the lack of mechanisms for specifying complex events or actions by combining simpler ones. In the last chapter we defined EAPs has a language whose statements are sets of particular Evolp statements. The definition of complex commands and statements by combining simpler ones is claimed to be the driving approach to Evolp programming by its authors (see [ABLP02]). However, there are in general no built-in facilities in Evolp allowing to build such complex commands. In particular, there is not clear way for determining the occurrence of events at different instants (or, more in general, of facts occurring at different instants) nor for specifying the sequential execution of actions, nor finally, for clustering a combination of actions into a single unit with a name. For instance, it is not possible to assign a name to a combination of actions and to write simpler codes by using that name in place of the original combination of actions. Moreover, these limitations are not bound to Evolp but also to all the other languages for LP updates.

Hence, to overcome the limitations of both ECA and LP updates languages, we define a new ECA language incorporating complex events, external and complex actions, and with the inference and evolving capabilities of Evolp . The new framework allows to combine simpler events and actions into more complex ones by two distinct *algebras of operators*. For these reasons the new framework is called ERA (after Evolving Reactive Algebraic programs). The semantics of ERA is defined by means of an *inference system* (specifying what conclusions are derived by a program) and of an *operational semantics* (specifying the effects of actions). The former is derived from the refined semantics for DyLPs [ABBL05]. The latter is defined by a transition system inspired by existing work on process algebras. [Mil89, Hoa85, SJG96].

We show in Section 6.5.1 that every Evolp program (and thus every EAP program) has a straightforward encoding in ERA preserving the semantics of the programs. Hence ERA can be considered as an extension of Evolp and, moreover, ERA incorporates the action description formalism of EAPs.

The rest of the chapter is structured as follows: in Section 6.1.1 we introduce some notation. Section 6.2 starts with an informal introduction to the language, presents its constructs highlighting its main features and then proceeds to the formal definition of the syntax of the language. Section 6.3 presents the semantics of ERA based on the

notion of *ERA system*. This is done in two steps by first providing inference relations on ERA systems (Section 6.3.3) and then providing a semantics for the execution of actions. Section 6.4.3 illustrates examples of complex actions that are used in the rest of the chapter. Section 6.5 compares ERA to existing LP updates languages. In particular, it provides an interpretation of Evolp programs as ERA programs. Section 6.6, instead, compares ERA to other ECA formalisms. Finally Section 6.7 provides concluding remarks and trace the line for possible future work.

6.1.1 Notation

Since the concept of concatenating a program to a sequence will be extensively used in the remainder, we introduce some specific notation to simplify the exposition.

Let E_S be a sequence of programs and E_i a single program. By $E_i.E_S$ we denote the sequence with head E_i and tail E_S. If E_S has length n. By $E_S..E_{n+1}$ we denote the sequence whose first n elements are those of E_S and whose $(n+1)^{th}$ element is E_{n+1}. For simplicity, we use the notation $E_i.E_{i+1}.E_S$ and $E_S..E_i..E_{i+1}$ instead of $E_i.(E_{i+1}.E_S)$ and $(E_S..E_i)..E_{i+1}$, respectively, whenever this creates no confusion. Symbol *null* denotes the empty sequence. Let E_S be a sequence of n programs and $i \leq n$ a natural number, by E_S^i we denote the sequence of the first i elements of E_S. Let $\mathcal{P} = \mathcal{P}'..P_i$ be a sequence of programs and E_i a program, by $\mathcal{P} \uplus E_i$ we denote the sequence $\mathcal{P}'..(P_i \cup E_i)$. Given any tuple $a = (a_1, \ldots, a_n)$, of length n, by $\pi(a)_i$ we denote the $i-th$ element a_i of a. Given any set V of tuples of length n, by $\Pi(V)_i$ we denote the $i-th$ projection of V, i.e. the set of elements occurring as the i^{th} element of a tuple in V. Formally:

$$\Pi(V)_i = \{a_i : (a_1, \ldots, a_i, \ldots, a_n) \in V\}$$

6.2 Outline and syntax of the language

Before the formal definition of ERA, we start here by informally introducing the various constructs of the language. As stated in the introduction, we aim at defining a language exhibiting both the advantages of ECA languages (with active rules, complex events and action) and of LP updates (with inference rules, possibility of declaratively specifying self-updates). As such, expressions in an ERA programs are divided in *rules* (themselves divided into *active, inference and inhibition rules*) and *definitions* (themselves divided into *event* and *action definitions*).

Active rules are as usual in ECA languages, and have the form (6.1), i.e.

On *Event* **If** *Condition* **Do** *Action*

where *Event* is a basic or a complex event expressed in an event algebra, *Condition* is a

conjunction of (positive or negative) literals and *Action* is a basic or a complex action. *Inference rules* are LP rules (possibly with default negation in the head).

Finally, ERA also includes *inhibition rules* of the form:

When B **Do** *not Action*

where B is a conjunction of literals and events. Such an expression intuitively means: when B is true, do not execute *Action*. Inhibition rules are useful for updating the behavior of active rules. If the inhibition rule above is asserted all the rules with *Action* in the head are updated with the extra condition that B must *not* be satisfied in order to execute *Action*.

ERA allows to combine basic events to obtain complex ones by an event algebra. The operators we use are similar to the Snoop algebra (see [AC03]) : $\triangle$, $\bigtriangledown$, A, *not*. Intuitively, $e_1 \triangle e_2$ occurs at an instant i iff both e_1 *and* e_2 occur at i; $e_1 \bigtriangledown e_2$ occurs at instant i iff e_1 *or* e_2 occurs at instant i; *not* e occurs at instant i iff e *does not* occur at instant i; $A(e_1, e_2, e_3)$ occurs at the same instant of e_3, in case e_1 occurred before, and e_2 did not occur in the middle. This operator is very important since it allows to combine (and reason with) events occurring at different time points.

Actions can also be basic or complex, and they may affect both the stored knowledge (internal actions) or the external environment. Basic external actions are related to the specific application of the language. Basic internal actions are for adding or retracting facts and expressions i.e. inference, active or inhibition rules and definitions of actions and events (see more on these definitions below) of the form $assert(\tau)$ and $retract(\tau)$, respectively, and for raising basic events, of the form $raise(e)$. There is also an internal basic action *skip* that has no effect but is nevertheless useful for defining complex actions. Complex actions are obtained by applying algebraic operators on basic actions. Such operators are: $\triangleright$, $\|$, IF, the first for executing actions *sequentially*, and the second for executing them *concurrently*. Executing $IF(C, a_1, a_2)$ amounts to execute a_1 in case C is true, and to execute a_2 otherwise.

To improve the modularity of the language, ERA allows for *event* and *action definition* expressions. These are of the form, respectively, e_{nam} **is** e and a_{nam} **is** a, where e_{nam} (resp. a_{nam}) is an atom representing a new event and e (resp. a) is an event (resp. an action) obtained by the event (resp. action) algebra above. It is also possible to use defined events (resp. actions) in the definition of other events (resp. actions).

To better motivate and illustrate these various constructs of the language ERA, including how they concur with the features mentioned in the introduction, we present now an example from the domain of monitoring systems.

Example 6.2.1 *Let us consider an (ECA) system for managing electronic devices in a building, e.g. the phone lines and the fire security system. The system receives inputs such as signals of sensors and messages from employees and administrators, and can activate devices like electric doors or fireplugs. Sensors alert the system whenever an abnormal quantity of smoke is found.*

If a (basic) event [2] $alE(S)$ *, signaling a warning from sensor S occurs, the system opens all the fireplugs Pl in the floor Fl where S is located. This behavior is encoded by the active rule:*

$$\textbf{On}\ alE(S)\ \ \textbf{If}\ floor(S,Fl), firepl(Pl), floor(Pl,Fl)\ \textbf{Do}\ openA(Pl) \quad (\alpha)$$

In addition to the one encoded by rule α*, another reactive behavior is required when the signals are given by several sensors. If two signal events* $alE(S_1)$*,* $al(S_2)$ *occur without a* $stop\_alertE(S_1)$ *event (signaling that the alert coming from* S_1 *extinguished) occurring in the meanwhile, the event* $alert2E(S_1,S_2)$ *occurs*[3]*. Event* $alert2E(S_1,S_2)$ *is defined as follows:*

$$alert2E(S_1,S_2)\ \textbf{is}\ A(alE(S_1),alE(S_2),stop\_alertE(S_1)) \bigtriangledown (alE(S_1) \bigtriangleup alE(S_2)).$$

The event $A(alE(S_1),alE(S_2),stop\_alertE(S_1))$ *occurs if first* $alE(S_1)$ *occurs, then* $alE(S_2)$ *occurs and* $stop\_alertE(S_1)$ *does not occur at any instant between* $alE(S_1)$ *and* $alE(S_2)$ *or, alternatively if* $alE(S_1)$ *and* $alE(S_2)$ *occur at the same instant.*

Whenever event $alert2E(S_1,S_2)$ *occurs and the sensors* S_1 *and* S_2 *are located in different rooms, the system reacts by executing the complex action* $fire\_alarmA$*. This is encoded by the following active rule:*

$$\textbf{On}\ alert2E(S_1,S_2)\ \ \textbf{If}\ not\ same\_room(S_1,S_2)\ \textbf{Do}\ fire\_alarmA.$$

where the literal $not\ same\_room(S_1,S_2)$ *is true iff the sensors* S_1 *and* S_2 *are located in different rooms. The truth value of* $same\_room(S_1,S_2)$ *is determined by the inference rule*

$$same\_room(S_1,S_2) \leftarrow room(S_1,R), room(S_2,R).$$

where the atom $room(S,R)$ *is true iff the sensor S is located in the room* [4] *R.*

The action $fire\_alarmA$ *applies a security protocol: All the doors are unlocked (by the basic action* $opendoorsA$*) to allow people to leave the building. At the same time, a phone call is sent to a firemen station (by the action* $firecallA$*). Then the system cuts the electricity in the building (by action* $turnA(elect,off)$*). Actions* $opendoorsA$ *and* $firecallA$ *can be executed simultaneously, but* $turnA(elect,off)$ *has to be executed after the electric doors have been opened and the fireman station have been called, i.e. after* $opendoorsA$ *and* $firecallA$ *have been executed. The action* $fire\_alarmA$ *is hence defined as follows.*

$$fire\_alarmA\ \textbf{is}\ (opendoorsA \| firecallA) \triangleright turnA(elect,off).$$

We provide now an example of evolution*. More formal details of how to update an ERA*

[2] In the sequel, we use names of atoms ending in E to represent events, and ending in A to represent actions.

[3] If, instead, S_2 occurs before S_1, the event $alert2E(S_2,S_1)$ occurs.

[4] The KB of the system may take note of which sensor is located in which room by a database of facts of the form $room(s,r)$ relating each sensor s to its location room r.

system are given in Section 6.3. At some point, the administrators decide to update the behavior of the system in such a way that, from then onwards, when a sensor S raises an alarm, only the fireplugs in the room R where S is located is opened. This update to the behavior of the system is encoded by asserting the following inhibition rule:

$$\tau: \quad \textbf{When } alE(S), room(S,R), not\ room(Pl,R)\ \textbf{Do } not\ openA(Pl).$$

When τ is asserted, if $alE(S)$ occurs in room R, any fire plug Pl which is not in R is not opened, even if Pl and S are on the same floor.

Let us assume, also, that the update at the behavior encoded by the inhibition rule τ is supposed to be done at time t. Assuming that the system has an internal clock and that the event $clockE(t)$ occurs at time t, it is sufficient to assert the following active rule:

$$\textbf{On } c\ lockE(t)\ \textbf{If } true\ \textbf{Do } assert(\tau).$$

The rule above is an example of a rule that should be triggered only once. It would be more efficient to delete this kind of rules after they exhaust their function. In Section 6.4.3 we will examine how to write rules that retract themselves after being triggered.

After the informal presentation above, we formally define the syntax of ERA.

Definition 57 *Let $\mathcal{L}$, $\mathcal{E}_B$, $\mathcal{E}_{nam}$, $\mathcal{A}_X$ and $\mathcal{A}_{nam}$ be disjoint sets of atoms called:* condition alphabet *and set of, respectively,* basic events, event names, external actions, action names, *and let L, e_b, e_{nam}, a_x and a_{nam} be generic elements of, respectively, $\mathcal{L}$, $\mathcal{E}_B$, $\mathcal{E}_{nam}$, $\mathcal{A}_X$ and $\mathcal{A}_{nam}$.*

The set of positive events *(denoted $\mathcal{E}$) over $\mathcal{E}_B$, and $\mathcal{E}_{nam}$ consists of the atoms form:*

$$e_p ::= e_b \mid e_1 \triangle e_2 \mid e_1 \bigtriangledown e_2 \mid A(e_1,e_2,e_3) \mid e_{nam}.$$

where e_1, e_2, e_3 are generic elements of $\mathcal{E}$. An event over $\mathcal{E}$ *is any literal over $\mathcal{E}$. A* negative event *over $\mathcal{E}$ is any literal of the form not e_p.*

A basic action *a_b over $\mathcal{E}, \mathcal{L}, A_X, \mathcal{A}_{nam}$ is any atom of the form:*

$$a_b ::= skip \mid a_x \mid raise(e_b) \mid assert(\tau) \mid retract(\tau).$$

where τ is any ERA rule, action or event definition over $\mathcal{L}^{ERA}$.

The set of actions *$\mathcal{A}$ over $\mathcal{E}, \mathcal{L}, A_X, \mathcal{A}_{nam}$ is the set of atoms a of the form:*

$$a ::= a_b \mid a_1 \triangleright a_2 \mid \|(A) \mid IF(C, a_1, a_2) \mid a_{nam}$$

where a_1 and a_2 are arbitrary elements of $\mathcal{A}$, A is any subset of $\mathcal{A}$ and $C = (C_1, \ldots, C_n)$ is any conjunction of literals [5] *over $\mathcal{E} \cup \mathcal{L}$.*

[5] The round parentheses are used to avoid confusion between the elements of C and actions a_1, a_2. We drop the parentheses whenever C is a single literal.

The ERA alphabet $\mathcal{L}^{ERA}$ *over* $\mathcal{L}$, $\mathcal{E}_B$, $\mathcal{E}_{nam}$, $\mathcal{A}_X$ *and* $\mathcal{A}_{nam}$ *is the triple* $\mathcal{E}, \mathcal{L}, \mathcal{A}$. *Let* e *and* a *be arbitrary elements of, respectively,* $\mathcal{E}$ *and* $\mathcal{A}$, *Body any set of literals over* $\mathcal{L}$, B *any set of literals over* $\mathcal{L}^{ERA}$ *and Cond any set of literals over* $\mathcal{L} \cup \mathcal{A}$.

- *An* ERA expression *is either an ERA definition or an ERA rule.*
- *An* ERA definition *is either an event definition or and action definition.*
- *An* event definition *over* $\mathcal{L}^{ERA}$ *is any expression of the form* e_{nam} ***is*** e.
- *An* action definition *over* $\mathcal{L}^{ERA}$ *is any expression of the form* a_{nam} ***is*** a.
- *An* ERA rule *is either an inference, active or inhibition rule over* $\mathcal{L}^{ERA}$.
- *An* inference rule *over* $\mathcal{L}^{ERA}$ *is any rule of the form* $L \leftarrow Body$.
- *An* active rule *over* $\mathcal{L}^{ERA}$ *is any rule of the form* ***On*** e ***If*** $Cond$ ***Do*** a.
- *An* inhibition rule *over* $\mathcal{L}^{ERA}$ *is any rule of the form* ***When*** B ***Do*** *not* a.
- *An* ERA program *over* $\mathcal{L}^{ERA}$ *is any set of ERA expressions over* $\mathcal{L}^{ERA}$.
- *A* dynamic ERA program *over* $\mathcal{L}^{ERA}$ *is any sequence of ERA programs over* $\mathcal{L}^{ERA}$.
- *An* ERA input program *over* $\mathcal{L}^{ERA}$, *is any set of ERA expressions over* $\mathcal{L}^{ERA}$, *actions or basic events.*

In the actions and events definitions of the form

$$e_{nam} \textbf{ is } e. \qquad a_{nam} \textbf{ is } a.$$

both e_{nam} *and* a_{nam} *are the* heads of the definitions, *while* e *and* a *are the* bodies of the definitions.

In the following, as a convention, all the elements of E_{nam} will end with suffix E and all the elements of $\mathcal{A}_{nam}$ will end with suffix A.

If a is a complex action obtained from b and other actions by the operators above we say that b is a *sub action* of a. Given a finite set of actions $\{a_1, \ldots a_n\}$ for the sake of simplicity we use the notation $\|(a_1, \ldots, a_n)$ instead of notation $\|(\{a_1, \ldots a_n\})$ and notation $a_1 \| a_2$ in place of notation $\|(a_1, a2)$ whenever this creates no ambiguity.

If necessary, we use round parentheses to specify precedence among the operators. Given n actions $a_1, a_2, \ldots, a_n$, to simplify the notation we use the expression $a_1 \triangleright a_2 \triangleright \ldots \triangleright a_n$ instead of $a_1 \triangleright (a_2 \triangleright \ldots \triangleright a_n) \ldots)$.

6.3 The Semantics of ERA

The defined syntax allows to program reactive systems, hereafter called *ERA systems*. An ERA system has, at each moment, an ERA dynamic program describing and determining its behavior, receives input (called *input program*) from the outside, and acts. The actions determine both the evolution of the system (by e.g. adding a new ERA program to the running sequence) and the execution in the external environment. As anticipated in the definition of the syntax, an input program E_i, over an alphabet $\mathcal{L}^{ERA}$, is any set of ERA expressions over $\mathcal{L}^{ERA}$ or basic events or finally actions of $\mathcal{L}^{ERA}$. At any instant i, an ERA systems receives a, possibly empty, input program[6] E_i. The sequence of programs $E_1, \ldots E_n$ denotes the sequence of input programs received at instants $1, \ldots, n$. A basic event e_b *occurs* at instant i iff the fact e_b belongs to E_i.

Since a complex event can be obtained by composing basic events that occurred in distinct time instants (viz., when using operator A), for detecting the occurrence of complex events it is necessary to store the sequence of all the received input programs. Formally, an *ERA system* $\mathcal{S}$ is a triple of the form $(\mathcal{P}, E_P, E_i.E_F)$ where $\mathcal{P}$ is an ERA dynamic program, E_P is the sequence of all the previously received input programs and $E_i.E_F$ is the sequence of the current (E_i) and the future (E_F) input programs. As it will be clear from Sections 6.3.2 and 6.3.3, the sequence E_F does not influence the system at instant i and hence no "look ahead" capability is required. However, since a system is capable (via action *raise*) of autonomously *raising* future events, future input programs are included as "passive" elements that are modified as effects of actions (see rule (6.4)).

The semantics of an ERA system specifies, at each instant, which conclusions are derived, which actions are executed, and what are the effects of those actions. Given a conclusion B, and an ERA system $\mathcal{S}$, $\mathcal{S} \vdash_e B$ denotes that $\mathcal{S}$ derives B (or that B is inferred by $\mathcal{S}$) under an inference system taking into account the occurrence of events. The definition of $\vdash_e$ is to be found in Section 6.3.2.

At each instant, an ERA system $\mathcal{S}$ *concurrently* executes all the actions a_k such that $\mathcal{S} \vdash_e a_k$. As a result of these actions an ERA system *transits* into another ERA system. While the execution of basic actions requires one transition, complex actions may involve the execution of several basic actions in a given order and hence require several transitions to be executed. For this reason, the effects of actions are defined by transitions of the form $< \mathcal{S}, A > \mapsto^G < \mathcal{S}', A' >$ where $\mathcal{S}, \mathcal{S}'$ are ERA systems, A, A' are sets of actions and G is a set of basic actions. The basic actions in G are the first step of the execution of the set of actions A, while the set of actions A' represents the remaining steps to complete the execution of A. For this reason, A' is also called the *set of residual actions* of A. The transition relation $\mapsto$ is defined by a transition system in Section 6.3.3. At each instant an ERA system receives an input program, derives a new set of actions A_N

[6] ERA adopts a discrete concept of time, any input program is indexed by a natural number representing the instant at which the input program occurs.

and starts to execute these actions together with the residual actions not yet executed. As a result, the system evolves according to the transition relation [7] $\rightarrow$. Formally:

$$\frac{A_N = \{a_k \in \mathcal{A} : \ \mathcal{S} \vdash_e a_k\} \ \wedge \ <\mathcal{S}, (A \cup A_N)> \mapsto^G <\mathcal{S}', \mathcal{A}'>}{<\mathcal{S}, A> \rightarrow^G <\mathcal{S}', \mathcal{A}'>} \tag{6.2}$$

Having defined a one-step transition from a couple $<\mathcal{S}, A>$ to a couple $<\mathcal{S}', \mathcal{A}'>$, the transition in i steps from $<\mathcal{S}, A>$ to a couple $<\mathcal{S}', \mathcal{A}'>$ is defined as the i-th iteration of the transition $\mapsto^G$ leading from $<\mathcal{S}, A>$ to a couple $<\mathcal{S}', \mathcal{A}'>$.

Definition 58 *Let $\mathcal{S}$, $\mathcal{S}'$ and $\mathcal{S}''$ be ERA systems and G, A, A' A' be sets of actions. For any natural number i, the transition operator $\mapsto^i$ leading $<\mathcal{S}, A>$ into $<\mathcal{S}', \mathcal{A}'>$ is defined as follows.*

$$\begin{aligned} &<\mathcal{S}, A> \mapsto^0 <\mathcal{S}, A> \\ &<\mathcal{S}, A> \mapsto^i <\mathcal{S}', \mathcal{A}'> \Leftrightarrow \ \exists\, (\mathcal{S}'', \mathcal{A}'', G)\ t.c. \\ &\qquad\qquad <\mathcal{S}, A> \mapsto^G <\mathcal{S}'', \mathcal{A}''> \ \wedge \\ &\qquad\qquad <\mathcal{S}'', \mathcal{A}''> \mapsto^{i-1} <\mathcal{S}', \mathcal{A}'> \end{aligned}$$

So far, we have considered the syntax and semantics of ERA systems without variables. However, ERA systems written for managing practical situations will always contain variables. A non-ground ERA system is an ERA system whose literals (either event, action or normal literals) are non ground (i.e. contain variables). See Section 2.2 for more details on non ground literals. The semantics of a non ground ERA system $\mathcal{S}$ is given by the semantics of the *grounded version of* $\mathcal{S}$, i.e. the ERA system obtained from $\mathcal{S}$ by substituting in all possible ways each of the variables of in $\mathcal{S}$ by elements of its Herbrand universe.

6.3.1 Inferring conclusions on DyLPs

In the following, a conclusion over an alphabet $\mathcal{L}$ is any set of literals over $\mathcal{L}$. An inference relation $\vdash$ is a relation between a DyLP and a conclusion. The notation $\mathcal{P} \vdash B$ means that the conclusion B is inferred (or derived) from $\mathcal{P}$ or, alternatively, that $\mathcal{P}$ infers (or derives) B. A notion of inference is strictly linked to the meaning of a program, i.e. to its semantics. Different choices of the basic semantics lead to different inference relations. We present here three possible notions of inference.

Well founded-based inference relation. If the basic semantics for DyLPs is the well founded semantics of Chapter 4, then the most natural inference relation (denoted by $\vdash_{WF}$) is the one deriving every conclusion that is true according to the well founded model of the considered *DyLP*. Formally:

$$\mathcal{P} \vdash_{WF} B \ \Leftrightarrow \ B \subseteq WFDy(\mathcal{P})$$

[7] Note that transition relation $\mapsto$ defines the effect of the execution of a set of actions, while $\rightarrow$ defines the global evolution of the system.

Refined semantics-based inference relation: cautious reasoning. Given a DyLP $\mathcal{P}$ with a unique refined model M and a conclusion B, it is natural to define an inference relation $\vdash$ as follows: $\mathcal{P} \vdash B \Leftrightarrow B \subseteq M$ (B is derived iff B is a subset of the unique refined model). However, in the general case of programs with several refined models, there could be several reasonable ways to define such a relation. A possible choice is to derive a conclusion B iff B is a subset of the intersection of all the refined models of the considered program. Formally:

$$\mathcal{P} \vdash_{\cap} B \Leftrightarrow B \subseteq M \; \forall \, M \in RS(\mathcal{P})$$

where $\mathcal{RS}(\mathcal{P})$ is the set of all refined models of $\mathcal{P}$. This choice is called *cautious reasoning*.

Refined semantics-based inference relation: brave reasoning. Another possibility is to select one model M (by a selecting function Se) and to derive all the conclusions that are subsets of that model. Formally:

$$\mathcal{P} \vdash_{Se} B \Leftrightarrow B \subseteq Se(\mathcal{RS}(\mathcal{P}))$$

This choice is called *brave reasoning*. Note that $\vdash_{Se}$ also depends from the chosen selecting function Se.

In the following, in the context of DyLPs, whenever an inference relation $\vdash$ is mentioned, we assume that $\vdash$ is any one of the relations defined above.

6.3.2 Inferring conclusions on ERA systems

The inference mechanism of ERA is derived from the semantics for DyLPs described in Chapters 3 and 4. We just provided three distinct ways to define an inference relation $\vdash$ between a DyLP and a conclusion on the basis of either the refined or the well founded semantics for DyLPs. In the following we show how such an inference relation $\vdash$ defined on DyLPs, is extended to ERA systems in order to include conclusions involving the occurrence of events. The new, extended, inference relation is denoted $\vdash_e$. Accordingly, we denote by, respectively, $\vdash_{\cap\, e}$, $\vdash_{WF\ e}$ and $\vdash_{Se\ e}$ the inference relations obtained by extending, respectively, $\vdash_{\cap}$, $\vdash_{WF}$ and $\vdash_{Se}$.

Let $\mathcal{S} = (\mathcal{P}, E_P, E_i.E_F)$ be an ERA system over the alphabet $\mathcal{L}^{ERA} : (\mathcal{E}, \mathcal{L}, \mathcal{A})$, with $E_P = E_1.\cdots.E_{i-1}.null$. For any positive integer $m < i$, let $\mathcal{S}^m$ be the ERA system[8] $(\mathcal{P}^m, E_P^{m-1}, E_m.null)$. Sequence E_F represents future input programs and is irrelevant for the purpose of inferring conclusions in the present, and sequence E_P stores previous events, and is only used for detecting complex events. The relevant expressions, hence, are those in $\mathcal{P}$ and E_i. As a first step we reduce to LP rules the expressions of

[8]We recall that, for any DyLP $\mathcal{P}$, $\mathcal{P}^m$ is the sequence of the first m updates of $\mathcal{P}$

these programs. An event definition associates an event e to a new atom e_{nam}. This is encoded by the rule $e_{nam} \leftarrow e$. Action definitions, instead, specify what are the effects of actions and hence are not relevant for inferring conclusions. Within ERA, actions are executed iff they are inferred as conclusions. Hence, active (resp. inhibition) rules are replaced by LP rules whose heads are actions (resp. negation of actions) and whose bodies are the events and conditions of the rules. In this section we show how the inference relation $\vdash$, just defined on DyLPs, is extended to ERA systems in order to include conclusions involving the occurrence of events.

Formally: let $\mathcal{P}^R$ and E_i^R be the DyLP and GLP obtained by $\mathcal{P}$ and E_i by deleting every action definition and by replacing:

every rule	**On** e **If** *Condition* **Do** *Action*.	with	$Action \leftarrow Condition, e.$
every rule	**When** B **Do** *not Action*	with	$not\, Action \leftarrow B.$
every definition	e_{nam} **is** e.	with	$e_{nam} \leftarrow e.$

Basically events are reduced to ordinary literals. Since events are meant to have special meanings, we encode these meanings by extra rules. Intuitively, operators $\triangle$ and $\bigtriangledown$ stands for the logic operators $\wedge$ and $\vee$. This is encoded by the following set of rules

$$ER(\mathcal{E}) : \triangle(e_1,e_2) \leftarrow e_1,e_2.\ \ \bigtriangledown(e_1,e_2) \leftarrow e_1.\ \ \bigtriangledown(e_1,e_2) \leftarrow e_2.\ \ \forall\, e_1,e_2,e_3\ \in \mathcal{E}$$

Event $A(e_1,e_2,e_3)$ occurs at instant i iff e_2 occurs at instant i, e_1 occurred in some previous instant, and e_3 did not occur in the meanwhile. This is formally encoded by the set of rules $AR(\mathcal{S},i)$ defined as follows[9]:

$$AR(\mathcal{S},i) = \left\{ \begin{array}{l} A(e_1,e_2,e_3) \leftarrow e_2\ s.t.\ e_1,e_2,e_3\ \in \mathcal{E} \wedge \\ \exists\, m < i\ s.t.\ \mathcal{S}^m\ \vdash_e e_1 \wedge\ \forall j:\ m < j \leq i,\ \mathcal{S}^j\ \nvdash_e e_3 \end{array} \right\}$$

The sets of rules E_i^R, $ER(\mathcal{E})$ and $AR(\mathcal{S},i)$ are added to $\mathcal{P}^R$ and conclusions are derived by the inference relation $\vdash$ applied on the obtained DyLP[10].

Definition 59 *Let $\vdash$ be an inference relation defined as in Section 6.3.1, $\mathcal{S}$, $\mathcal{P}^R$, E_i^R, $ER(\mathcal{E})$, $AR(\mathcal{S},i)$ be as above and K be any conclusion over $\mathcal{E} \cup \mathcal{L} \cup \mathcal{A}$. Then:*

$$\mathcal{S} = (\mathcal{P}, E_P, E_i.E_F)\ \vdash_e K \quad \Leftrightarrow \quad \mathcal{P}^R \uplus Ev(\mathcal{S},i) \vdash K$$

where

$$Ev(\mathcal{S},i) = E_i^R \cup ER(\mathcal{E}) \cup AR(\mathcal{S},i)$$

[9]The definition of $AR(\mathcal{S},i)$ involves relation $\vdash_e$ which is defined in terms of $AR(\mathcal{S},i)$ itself. This mutual recursion is well-defined since, at each recursion, $AR(\mathcal{S},i)$ and $\vdash_e$ are applied on previous instants until eventually reaching the initial instant (i.e. , the base step of the recursion)

[10]The program transformation above is functional for defining a declarative semantics for ERA, rather than providing an efficient tool for an implementation. For that, specific algorithms for event-detection clearly seem to provide a more efficient alternative.

In this chapter $\vdash_e$ will denote the chosen inference relation. As the reader may notice, we specified no rules for operator *not* . These rules are not needed since event (literal) *not* e_p is inferred by default negation whenever there is no proof for e_p.

The following proposition formalizes the intuitive meanings of the various operators provided in Section 6.2.

Proposition 6.3.1 *Let $\mathcal{S}$ be as above, e_b, a basic event, e_p a positive event, e_{nam} an event name and e_1, e_2, e_3 three events, the following double implications hold:*

$$
\begin{array}{lll}
1 \quad \mathcal{S} \vdash_e e_b & \Leftrightarrow \quad e_b \in E_i & \\
2 \quad \mathcal{S} \vdash_e e_1 \triangle e_2 & \Leftrightarrow \mathcal{S} \vdash_e e_1 \ \wedge \ \mathcal{S} \vdash_e e_2. & \\
3 \quad \mathcal{S} \vdash_e e_1 \bigtriangledown e_2 & \Leftrightarrow \mathcal{S} \vdash_e e_1 \ \vee \ \mathcal{S} \vdash_e e_2. & \\
4 \quad \mathcal{S} \vdash_e e_{nam} & \Leftrightarrow \mathcal{S} \vdash_e e \quad \wedge & e_{nam} \ \textbf{is} \ e \ \in \mathcal{P} \\
5 \quad \mathcal{S} \vdash_e A(e_1, e_2, e_3) & \Leftrightarrow \mathcal{S} \vdash_e e_2 \wedge \exists \, m < i \ s.t. & \mathcal{S}^m \vdash_e e_1 \wedge \\
 & \quad \forall j \ s.t. \ m < j \leq i : & \mathcal{S}^j \ \nvdash_e e_3. \\
6 \quad \mathcal{S} \vdash_e not \, e_p & \Leftrightarrow \quad \mathcal{S} \nvdash_e e_p. &
\end{array}
$$

The inference relation $\vdash_e$ may be defined in terms of any of the underlying inference relations on DyLPs $\vdash_{Se}$, $\vdash_{\cap}$ or $\vdash_{WF}$. By Proposition 6.3.1 it follows that, for any event e, to establish weather $\mathcal{S} \vdash_e e$ or not, the choice of the underlying inference relation on DyLPs is irrelevant. Indeed, by statement 1, a basic event e_b is inferred iff it belongs to the current input program E_i. This clearly does not depend from the choice of the underlying inference relation on DyLPs. By statements $2-6$, the occurrence of a complex event e solely depends on the occurrence of its sub events. Hence, by induction, we conclude that the choice of the underlying inference relation on DyLPs is irrelevant for determining the occurrence of e. Hence, for the sake of deriving events, it does not matter what is the basic inference relation. Note that this is not, in general, the case for a literal L.

6.3.3 Execution of actions

The effect of the concurrent execution of a set of actions is determined by combining the effects of the single actions in A. The effect of the execution of a single action is determined by transitions of the form

$$< \mathcal{S}, a > \mapsto^G < \mathcal{S}', a' >$$

where a and a' are actions and a' is the residual action of a. Transition rules 6.11 and 6.12 determines how to combine these effects.

The effects of basic actions on the current ERA program are defined by the *updating function* $up/2$. Let $\mathcal{P}$ be an ERA dynamic program and A a set of, either internal or

external, basic actions. The output of function $up/2$ is the updated program $up(\mathcal{P},A)$ obtained in the following way: First delete from $\mathcal{P}$ all the rules and definitions retracted according to A; then update the obtained ERA dynamic program with the program consisting of all the rules and definitions asserted according to A. Formally :

$$\begin{array}{lcl} DR(A) & = & \{\tau : \ assert(\tau) \in A\} \\ R(A) & = & \{\tau : \ retract(\tau) \in A\} \\ up(\mathcal{P},A) & = & (\mathcal{P} \setminus R(A)\ ).. DR(A) \end{array}$$

Let e_b be any basic event, and a_i an external action or an internal action of the form: $assert(\tau)$ or $retract(\tau)$. On the basis of function $up/2$ above, we define the effects of (internal and external) basic actions. At each transition, the current input program E_i is evaluated and stored in the sequence of past events, and the subsequent input program in the sequence E_F becomes the current input program. The only exception involves action $raise(e_b)$ that adds e_b in the subsequent input program E_{i+1}. When an action a is completely executed its residual action is 0 where 0 is a *termination action* not belonging to the free grammar of definition 57 and representing the termination of the actions. Unlike other actions, there exists no transition rule for 0. The termination action 0 must not be confused with the *skip* action. As an effect of the execution of *skip* the current input program is consumed and the next input program is examined (see transition rule 6.3). Intuitively, the effect of an action *skip* (assuming that no other action is executed concurrently) is to wait for the next set of inputs.

The execution of an action may also terminate because there are no transition rules applicable to it. An action which has an applicable transition rule is said to be *executable* while an action (different from 0) which has no applicable rule is said to be *non executable*. In general, since a transition rule is always applied to a couple of an ERA system and an action, an action is executable or not depending on the related ERA system on which it is to be executed.

Any basic action a_b (unlike complex ones) is completely executed in one step, and so it has no residual actions. Note that, according to definition 57, 0 is not a basic action).

$$< (\mathcal{P}, E_P, E_i.E_F), skip > \ \mapsto^{\emptyset} \ < (\mathcal{P}, E_P..E_i, E_F), 0 > \tag{6.3}$$

$$< (\mathcal{P}, E_P, E_i.E_{i+1}.E_S), raise(e_b) > \ \mapsto^{\emptyset} \ < (\mathcal{P}, E_P..E_i, (E_{i+1} \cup \{e_b\}).E_F), 0 > \tag{6.4}$$

$$\frac{a_b = assert(\tau) \ \vee \ a_b = retract(\tau) \ \vee \ a_b \in A_X}{< (\mathcal{P}, E_P, E_i.E_F), a_b > \mapsto^{\{a_b\}} < (up(\mathcal{P}, \{a_b\}), E_P..E_i, E_F), 0 >} \tag{6.5}$$

Note that, although external actions do not affect the ERA system, as they do not affect the result of $up/2$, they are nevertheless *observable*, since they are registered in the set

of performed actions (cf. rule 6.5).

Unlike basic actions, generally the execution of a complex action involves several transitions. Action $a_1 \triangleright a_2$ consists of first executing all basic actions for a_1, until the residual action is 0, then execute all the basic actions for a_2. Formally:

$$\frac{< \mathcal{S}, a_1 > \mapsto^G < \mathcal{S}', a_1' > \wedge\, a_1' \neq 0}{< \mathcal{S}, a_1 \triangleright a_2 > \mapsto^G < \mathcal{S}', a_1' \triangleright a_2 >} \quad \frac{< \mathcal{S}, a_1 > \mapsto^G < \mathcal{S}', 0 >}{< \mathcal{S}, a_1 \triangleright a_2 > \mapsto^G < \mathcal{S}', a_2 >} \tag{6.6}$$

The execution of $IF(\mathcal{C}, a_1, a_2)$ amounts to the execution of a_1 if the system infers $\mathcal{C}$, or to the execution of a_2 otherwise. Formally:

$$\frac{\mathcal{S} \vdash_e \mathcal{C} \wedge < \mathcal{S}, a_1 > \mapsto^{G_1} < \mathcal{S}', a_1' >}{< \mathcal{S}, IF(\mathcal{C}, a_1, a_2) > \mapsto^{G_1} < \mathcal{S}', a_1' >} \tag{6.7}$$

$$\frac{\mathcal{S} \nvdash_e \mathcal{C} \wedge < \mathcal{S}, a_2 > \mapsto^{G_2} < \mathcal{S}'', a_2'' >}{< \mathcal{S}, IF(\mathcal{C}, a_1, a_2) > \mapsto^{G_2} < \mathcal{S}'', a_2'' >} \tag{6.8}$$

Let us consider an ERA system $\mathcal{S} = (\mathcal{P}, E_P, E_i.E_F)$ with $\mathcal{P} = P_1 \ldots P_n$, and let d be the action definition

$$a_{nam} \textbf{ is } a_k.$$

By abuse of notation, by $d \in \mathcal{S}$ we mean either $d \in P_j$ for some index j or $d \in E_i$. The execution of action a_{nam}, where a_{nam} is defined by one or more action definitions, corresponds to the executions of any action a_k such that a_{nam} **is** a_k belongs to $\mathcal{S}$ (this last transition rule introduces a source of non determinism in the transition system, since there could be several definitions and each of them correspond to a possible transition). Formally:

$$\frac{a_{nam} \textbf{ is } a_k. \in \mathcal{S} \;\wedge\; < \mathcal{S}, a_k > \mapsto^G < \mathcal{S}', a_k' >}{< \mathcal{S}, a_{nam} > \mapsto^G < \mathcal{S}', a_k' >} \tag{6.9}$$

Note that it is not always guaranteed that an action name has a transition rule. This is the case, for instance, when there are no action definitions whose head is the considered action name.

The argument of the parallel execution operator $\|$ is a set of actions A. The parallel execution of a set of actions is defined in terms of the execution of the various elements of the set. Given an ERA system $\mathcal{S}$, the execution of $\|(A)$ is the concurrent execution of all the actions a_k in A. Since, in general, an action a_k in A may have more than one transition, it is necessary to select *one* transition of the form

$$< (\mathcal{P}, E_P, E_i.E_{i+1}.E_F), a_k > \mapsto^{Gk} < (\mathcal{P}_k, E_P..E_i, \Delta_k.E_F), a_k' >$$

for each executable action a_k.

Ultimately, the the execution of $\|(A)$ is a transition of the form

$$< (\mathcal{P}, E_P, E_i.E_{i+1}.E_F), \|(A) > \mapsto^G < up(\mathcal{P}, G), E_P..E_i, E'_{i+1}.E_F, a' >$$

where G is the union of the basic sets of actions G_k for any action a_k in A, $up(\mathcal{P}, G)$ is the ERA dynamic program resulting by concurrently executing [11] all the basic actions in G, E''_{i+1} is the input program obtained by union of all the new input programs given by the execution of all the actions[12] a_ks, and finally a' is ether the action $\|(A')$ where A' is the set of all the residual actions a'_ks (different from 0) or the success action 0 if all the actions in A has 0 as residual action. Formally:

$$\frac{\begin{array}{c} X \subseteq \{(a_k, G_k, \Delta_k, a'_k) : a_k \in A \wedge \\ < (\mathcal{P}, E_P, E_i.E_{i+1}.E_F), a_k > \mapsto^{Gk} < (\mathcal{P}_k, E_P..E_i, \Delta_k.E_F), a'_k > \} \\ \wedge \forall\, x, y \in X :\ x \neq y \ \Rightarrow\ \pi(x)_1 \neq \pi(y)_1 \\ \wedge\, \Pi(X)_1 = A \ \wedge\ G = \bigcup \Pi(X)_2 \ \wedge \\ E'_{i+1} = E_{i+1} \cup \Delta \ \wedge\ \Delta = \bigcup \Pi(X)_3 :\ \wedge A' = \Pi(V)_4 \end{array}}{< (\mathcal{P}, E_P, E_i.E_{i+1}.E_F), \|(A) > \mapsto^G < up(\mathcal{P}, G), E_P..E_i, E'_{i+1}.E_F, succ(\|(A')) >} \tag{6.10}$$

where

$$succ(\|(A')) = \|(A' \setminus \{0\})\ \ if A' \setminus \{0\} \neq \emptyset \qquad succ(\|(A')) = 0\ \ if A' \setminus \{0\} = \emptyset$$

Having defined the parallel execution $\|(A)$ of a set of actions A, it is quite easy to define the execution of the set A as required by transition rule 6.2. While the parallel execution operator $\|(A)$ requires that *all* the actions of A are executed (and hence that all the action must be executable), when a set of actions is executed non executable actions are simply discarded. Apart from that, executing a set of actions A has the same effect of executing the action $\|(A)$, with the difference that the residual of a partially executed set of actions A is the set of the residual actions A' for each executable action in A. Formally:

$$\frac{\begin{array}{c} A_N = \{a \mid a \in A \wedge \nexists\, G', \mathcal{S}', a' : \\ < \mathcal{S}, a > \mapsto^{G'} < \mathcal{S}', a' > \} \\ \wedge\, B = A \setminus A_N \ \wedge\ < \mathcal{S}, \|(B) > \mapsto^G < \mathcal{S}', \|(B') > \end{array}}{< \mathcal{S}, A > \mapsto^G < \mathcal{S}', B' >} \tag{6.11}$$

[11] Note that $up/2$ takes as argument a *set* of basic actions.

[12] Note that this set is always a superset of E_{i+1}, even if the subset K of executable actions of A is empty.

$$\frac{\begin{array}{c} A_N = \{a \mid a \in A \wedge \nexists\, G', \mathcal{S}', a' : \\ < \mathcal{S}, a > \mapsto^{G\prime} < \mathcal{S}', a' > \} \\ \wedge\, B = A \setminus A_N \wedge\ < \mathcal{S}, \|(B) > \mapsto^{G} < \mathcal{S}', 0 > \end{array}}{< \mathcal{S}, A > \mapsto^{G} < \mathcal{S}', \emptyset >} \tag{6.12}$$

As it results from transition rule 6.2, the execution of a set of actions occurs when several actions are triggered, at the same instant by different reactive rules. According to transition rules 6.11 and 6.12 non executable actions are discarded. A possible alternative choice would have been to include the non-executable actions in the set of residual actions A' and let the system try to execute them again.

In case of this second option an action would never be discarded, the system would attempt to execute it until, possibly, it would succeed. Since actions are constantly generated by active rules, the result could be a permanent and constantly growing accumulation of actions. In order to prevent this possibility, the transition system was designed for discarding non executable actions. Moreover, also the possibility that, ultimately, an action succeeds a considerable amount of time after it was supposed to be executed is not so much desirable. For instance, the action might succeed several days after it has been triggered since the KB of the system was updated in a way that matches the conditions required by the action, but, after so much time, also the conditions determining the execution of that action may have changed in such a way that the execution of the action is no longer required or even undesirable. Even worse is the possibility that, in the meanwhile, the definition of the action, or of some of its sub actions, have been updated, resulting in the execution of an action different from the one expected.

As it comes from the rules above, the definition of concurrent execution of actions in ERA does not rely on any concept of *serialization*. Actions are (at least at a semantic level) truly concurrently executed. Indeed, actions may have three different effects. Namely: to update the system, to rise new events, or to modify the external environment (by external actions). Semantically, internal updates are defined by function $up/2$ which is defined over an ERA dynamic program and a *set* of basic actions, while the raised events are added to the next input program and are then processed concurrently. No serialization is then needed for this kind of actions. Finally, the description and execution of external actions does not belong to the semantics of ERA, since the meaning and effects of these actions depend on the application domains. Specific applications may require some notion of serialization for external actions (for instance, messages sent over the same communication channel are sent one by one).

6.4 Remarks

Before comparing ERA with other approaches in Sections 6.5 and 6.6, we make some remarks on specific aspects of the language that deserve a closer look.

6.4.1 Failure of actions

We have already seen that actions are divided into executable and non executable. It is not possible to establish whether an action is executable without referring to the underlying ERA system, in particular to the set of action definitions.

An action may be non executable only in case its execution would require the execution of some action name with no related applicable action definition. The simplest example of non executable actions are action names for which there exists no action definition whose body is executable. By convention, the action name $failA$ is never in the head of any action definition. Hence $failA$ is never executable. The action $failA$ can be used to *abort* the execution of the action or to prune some of the possible executions. Other examples of non executable actions involve actions whose definition is recursive. For instance, the action names a_{loop}, a_{loop1}, a_{loop2} that are head of the following actions definitions:

$$a_{loop} \textbf{ is } a_{loop}. \qquad a_{loop1} \textbf{ is } a_{loop2}. \qquad a_{loop2} \textbf{ is } a_{loop1}.$$

are not executable.

6.4.2 Concurrent execution of actions

In ERA, when a set of concurrent actions is executed, all the actions in the set are executed *in the same instant.* This happens, in particular, for the internal basic actions of the form $assert(\tau)$ and $retract(\tau)$ that modify the dynamic ERA program of the system. The semantics of the concurrent execution of these actions is given by the function $up/2$ defined in Section 6.3.3 taking as arguments a dynamic ERA program $\mathcal{P}$ and a set of actions A and determining the new ERA system $\mathcal{P}'$. In practice the function $up/2$ deletes from $\mathcal{P}$ all the rule τ such that $retract(\tau)$ belongs to A and update $\mathcal{P}$ with the ERA program P consisting of all the ERA expressions τ such that $assert(\tau)$ belongs to A. It is important to notice that, in general, the concurrent execution of actions is *not* equivalent to any sequential execution of the same actions. This is the result of the underlying DyLP semantics of ERA.

Given a DyLP $\mathcal{P}$ and an update U, it is not always possible to find a sequence of single rule updates $\mathcal{U} = U_1, \ldots, U_n$ such that $\bigcup U_i = U$, any U_i consists of a single rule and $\mathcal{P} \oplus \mathcal{U}$ is update equivalent to $\mathcal{P} \oplus U$. The reason for this, lies in the potential presence of conflicting rules in $\mathcal{P}$. We provide a simple example of this fact. Let $\mathcal{P}$ be

the following single update DyLP P, Q and K the following programs:

$$\begin{array}{lll} P & & d. \\ & & b \leftarrow not\ c. \\ & & c \leftarrow not\ b. \\ Q & \tau_1: & not\ a. \\ & \tau_2: & a \leftarrow b. \\ K & \tau_1: & not\ a. \\ & \tau_3: & a. \end{array}$$

The DyLP P,Q has the single refined model $\{d, c, not\ a, not\ b\}$ since the pair of rules in Q imposes a constraint that b must be false in in order to preserve consistency. The well founded model of P,K is, instead, the paraconsistent interpretation $\{a, not\ a, d\}$.

Let $\mathcal{Q} = Q_1, Q_2$, $\mathcal{Q}' = Q_2, Q_1$, $\mathcal{K} = Q_1, Q_3$, $\mathcal{K}' = Q_3, Q_1$ be DyLPs with:

$$Q_1 : \tau_1 \quad Q_2 : \tau_2 \quad Q_3 = \tau_3$$

The DyLP $P.\mathcal{Q}$ has two refined models: $\{d, not\ a, not\ b, c\}$ and $\{d, a, b, not\ c\}$. Indeed, if c is true and b is false, the rule τ_2 is not supported and hence a is not derived. On the other side, if b is true and c is false, the rule in τ_2 rejects τ_1 and derives a.

Also $P.\mathcal{Q}'$ has two refined models: $\{d, c, not\ b, not\ a\}$ and $\{d, b, not\ c, not\ a\}$, since the rule in τ_1 always rejects τ_2.

On the other side, the well founded model of $P.\mathcal{K}$ is $\{d, a\}$, since τ_3 rejects τ_1, while the well funded model of $P.\mathcal{K}'$ is $\{d, not\ a\}$, since τ_1 rejects τ_3,

Coming back to ERA, if the DyLP P in the example above is the current KB of the system, the concurrent execution of $assert(\tau_1)$ and $assert(\tau_2)$ leads to the new KB P,Q, and the concurrent execution of $assert(\tau_1)$ and $assert(\tau_4)$ leads to the new KB P,K.

The sequential execution of first $assert(\tau_1)$ and then $assert(\tau_2)$ leads, instead, to the KB $P.\mathcal{Q}$, and the sequential execution of first $assert(\tau_2)$ and then $assert(\tau_1)$ leads to the KBs $P.\mathcal{Q}'$. The sequential execution of first $assert(\tau_1)$ and then $assert(\tau_3)$ leads to the KB $P.\mathcal{K}$, and the sequential execution of first $assert(\tau_3)$ and then $assert(\tau_2)$ leads to the KBs $P.\mathcal{K}'$.

Depending on the chosen underlying inference relation (see Sections 6.3.1 and 6.3.2), the KBs P,Q and P,K infer different conclusions from, respectively, $P.\mathcal{Q}$, $P.\mathcal{Q}'$ and $P.\mathcal{K}$, $P.\mathcal{K}'$.

Indeed, if the underlying the inference relation is $\vdash_\cap$ we have:

$$P,Q \vdash_\cap c \quad P.\mathcal{Q} \nvdash_\cap c \quad P.\mathcal{Q}' \nvdash_\cap c$$

since both $P.\mathcal{Q}$ and $P.\mathcal{Q}'$ have two refined models, one where c is true and another where c is false.

If the underlying inference relation is $\vdash_{Se}$ we have:

$$\begin{array}{lll} P,Q & \vdash_{Se} c & \\ P.Q & \vdash_{Se} c & \Leftrightarrow \quad c \in Se(\mathcal{RS}(P.Q)) \\ P.Q' & \vdash_{Se} c & \Leftrightarrow \quad c \in Se(\mathcal{RS}(P.Q')) \end{array}$$

where $Se(\mathcal{RS}(P.Q)$ and $Se(\mathcal{RS}(P.Q')$ are the selected refined models of, respectively $P.Q$ and Q'. The KB P,Q always infers c since it has one refined model where c is true, while $P.Q$ and $P.Q'$ infers c iff the selected refined model is the one where c is true.

Finally, if the underlying inference relation is $\vdash_{WF}$ we have:

$$\begin{array}{lllll} P,K & \vdash_{WF} not\, a & \wedge & P,K & \vdash_{WF} a \\ P.\mathcal{K} & \nvdash_{WF} not\, a & \wedge & P.\mathcal{K} & \vdash_{WF} a \\ P.\mathcal{K}' & \vdash_{WF} not\, a & \wedge & P.\mathcal{K}' & \nvdash_{WF} a \end{array}$$

In all the three cases above, none of the KBs obtained by sequentially executing the actions above is semantically equivalent to the one obtained by their concurrent executions. The example clarifies that, in order to capture the full expressivity of the DyLP paradigm, the choice of allowing the concurrent execution of basic actions is mandatory.

6.4.3 Examples of frequently used complex actions

We define here some complex actions whose usage occurs in the rest of the chapter.

Self killing rules

An interesting application of the action algebra of ERA is the possibility to write active rules that retract themselves after they have been triggered. Such a *self killing rule* is written as follows :

$$\textbf{On } E \ \ \textbf{If } Cond \ \textbf{Do } A \triangleright selfkillA(E,Cond,A).$$

where $E,Cond,A$ are the usual event, condition and action and $selfkillA(E,Cond,A)$ is the action defined in the following way:

$$selfkillA(E,Cond,A) \textbf{ is } retract(\ \textbf{On } E \ \ \textbf{If } Cond \ \textbf{Do } A \triangleright selfkillA(E,Cond,A)).$$

When event E occurs and $Cond$ is satisfied, then A is executed. If A succeeds, then action $selfkillA(E,Cond,A)$ is executed, and its effect is to retract the rule itself. Note that, by relying on action definitions, it is possible to write a *self referring rule* which is something usually impossible since a finite syntactic formula cannot contain itself as a sub formula .

A rule that retracts itself after executing an action is always only triggered *once*. In the following we will use the expression

$$\textbf{On } E \ \ \textbf{If } Cond \ \textbf{Once } A \triangleright selfkillA.$$

instead of

$$\textbf{On } E \ \ \textbf{If } Cond \ \textbf{Do } A \triangleright selfkillA(E, Cond, A).$$

Complex actions defined by raising events

One way for defining a complex actions a_{nam} is to rely on active rules by raising events. An action defined in this way has an action definition of the form

$$a_{nam} \textbf{ is } aE$$

where aE is an event name. The event aE triggers one or active more rules of the form

$$\textbf{On } aE \ \ \textbf{If } Cond \ \textbf{Do } a.$$

One useful usage of this kind of action within non ground ERA programs is when an action is intended to be executed on all its instances matching some conditions.

For instance, the action $send\_mailA(M, Class)$, sending an email M to all the student of a $Class$, is defined as follows.

$$send\_mailA(M, Class) \textbf{ is } raise(send\_mailE(M, Class)).$$

$$\textbf{On } send\_mailE(M, Class) \ \ \textbf{If } student(S, Class) \ \textbf{Do } mailA(M, S).$$

where the condition $student(S, Class)$ is satisfied iff S is a student of the class S.

When the action $send\_mailA(m, class)$ is executed [13] the event $send\_mailE(m, class)$ is raised and *all* the rules of the form

$$\textbf{On } send\_mailE(m, class) \ \ \textbf{If } student(s, class) \ \textbf{Do } mailA(m, s).$$

for which the condition $student(s, class)$ is satisfied are triggered and the corresponding action $mailA(m, s)$ is executed. We shall see an usage of this way of defining actions in Section 6.5.4.

6.5 Comparisons to other updates languages

The language ERA is a bridge between LP updates languages and ECA languages. It is therefore important to understand its relationship to these two categories of languages.

[13] Here the arguments M and $Class$ are instantiated.

We opted for a detailed and formal comparison with Evolp [ABLP02] and more informal comparisons to other existing LP update languages (more precisely with *LUPS* [APPP02], *EPI* [EFST01], *KUL* [Lei03], and *KABUL* [Lei03]). This choice was made in order to spare the reader to lengthy proofs of correctness of transformation from each of these languages into ERA, all of which quite similar amongst themselves, and not much different from the one of Evolp which is included in the corresponding formal comparison. Nevertheless, we illustrate how it is possible to encode programs in each of these other languages in ERA, and provide intuitions on how these encodings work.

6.5.1 Relationship with Evolp

As anticipated in Section 6.1, although using a slightly different syntax, ERA has been thought as an extensions of Evolp . As a proof we will show here how, despite of the different syntax, any Evolp rule can be directly translated into an ERA rule without altering the meaning of a program.

The main syntactic difference (apart from the algebraic structures on actions and events of ERA) between the two languages is the syntax of the "active rules" determining the actions to execute. The Evolp statements:

$$r1: \ assert(\tau) \leftarrow B. \qquad r2: \ not\, assert(\tau) \leftarrow B.$$

are not allowed in ERA where the unique rules whose heads are actions (resp. negation of actions) are active rules (resp. inhibition rules). While rules of the form $r2$ can be directly translated in inhibition rules simply by replacing the symbol $\leftarrow$ by **When** , the translation of rules of the form $r1$ is slightly less direct, since an active rule is always triggered by an event, a concept not clearly encoded in Evolp where (non rejected) rules of the form r_1 produces effects whenever their body is satisfied. However, the problem is solved by introducing an event *tickE* that occurs in every input program and by putting *tickE* as the triggering event of the active rule corresponding to $r1$. The name *tickE* denotes how the event occurs at any evolution step (like the tick of a clock).

Definition 60 . *For any rule Evolp rule* $\tau : H \leftarrow B_1, \ldots, B_n$ *let* τ^E *be the following ERA rule:*

$$\begin{array}{lll} i) & & \\ \tau^E & = H^E \leftarrow B_1^E, \ldots, B_n^E. \ for\ H \neq assert(\eta) & and\ H \neq not\, assert(\eta) \\ \tau^E & = \textbf{\textit{On}}\ tickE\ \textbf{\textit{If}}\ B_1^E, \ldots, B_n^E\ \textbf{\textit{Do}}\ assert(\eta^E). & for\ H = assert(\eta) \\ \tau^E & = \textbf{\textit{When}}\ B_1^E, \ldots, B_n^E\ \textbf{\textit{Do}}\ not\, assert(\eta^E). & for\ H = not\, assert(\eta). \end{array}$$

where, for any literal L and any rule η*:*

$$\begin{array}{rcl} L^E & = & L. \ for\ L \neq assert(\eta)\ and\ L \neq not\, assert(\eta) \\ assert(\eta)^E & = & assert(\eta^E) \\ (not\, assert(\eta))^E & = & not\, assert(\eta^E) \end{array}$$

where the rule η^E is obtained as in i).

Given any Evolp program P, we denote by P^E the ERA program obtained by replacing every rule τ with τ^E. Given any Evolp sequence $\mathcal{P} : P_1, \ldots, P_n$, we denote by $\mathcal{P}^E$ the ERA sequence obtained by replacing every program P_i with P^E. Given any sequence $\mathcal{E} : E_1, \ldots E_n$ of input programs in Evolp , we denoted by E_i^{ERA} the ERA input program $E_i^E \cup \{tickE\}$ and by $\mathcal{E}^{ERA}$ the sequence

$$(E_1^E \cup \{tickE\}), \ldots, (E_n \cup \{tickE\})$$

Given any pair $(\mathcal{P}, \mathcal{E})$ where $\mathcal{P}$ in an Evolp sequence and $\mathcal{E}$ is a sequence of Evolp input programs, the ERA equivalent of $(\mathcal{P}, \mathcal{E})$ *is the era system $(\mathcal{P}^E, \emptyset, \mathcal{E}^{ERA})$.*

The system $(\mathcal{P}^E, \emptyset, \mathcal{E}^{ERA})$ translates the Evolp sequence $\mathcal{P}$ with sequence of input programs $\mathcal{E}$. However, Evolp is a language conceived for reasoning about possible evolutions of a program, while ERA is a program conceived for executing actions. The natural question is then what is the semantical link between an Evolp sequence and its translation in ERA. The answer is that, if the underlying inference relation of ERA is the brave reasoning inference relation $\vdash_{Se\ e}$, there is a one-to-one correspondence between the possible evolutions of the original Evolp sequence and that of its translation. What determines which evolution is taken by the translated ERA program is determined by the choice of the selecting function *Se* associated to the inference relation $\vdash_{Se\ e}$.

Theorem 6.5.1 *Let $(\mathcal{P}, \mathcal{E}^n)$ be a pair where $\mathcal{P}$ is an Evolp sequence and $\mathcal{E}^n$ is a sequence of Evolp input programs both in the language $\mathcal{L}_{assert}$. Let $\mathcal{M} : M_1 \ldots M_n$ be any sequence of two valued interpretations over the language of $(\mathcal{P}, \mathcal{E}^n)$ and $\mathcal{P} \oplus \mathcal{T}^{n-1}$ be the trace of $\mathcal{M}$ given $(\mathcal{P}, \mathcal{E}^n)$. Then $\mathcal{M}$ is an evolving stable model of $(\mathcal{P}, \mathcal{E}^n)$ at state n if and only if there exists an inference relation $\vdash_{Se\ e}$ such that*[14]*:*

$$< (\mathcal{P}^E, \emptyset, \mathcal{E}^{n\ ERA}), \emptyset > \rightarrow^{i-1} < (\mathcal{P}^E \oplus \mathcal{T}^{i-1\ E}, \mathcal{E}^{i-1\ ERA}, (E_i^{ERA}, \ldots, E_n^{ERA})), \emptyset >$$

for any $i \leq n$ and

$$(\mathcal{P}^E \oplus \mathcal{T}^{i-1\ E}, \mathcal{E}^{i-1\ ERA}, E_i^{ERA}, \ldots, E_n^{ERA}) \vdash_{Se\ e} L^E$$

for any L in M_i with $i \leq n$.

proof Let $\mathcal{M}^j$, $\mathcal{E}^j$, be, respectively the evolving interpretation $M_1, \ldots, M_j$ and the sequence of input programs $E_1, \ldots, E_j$.

It is enough to prove that Then $\mathcal{M}^j$ is an evolving stable model of $(\mathcal{P}, \mathcal{E}^j)$ at state j if and only if:

$$< (\mathcal{P}^E, \emptyset, \mathcal{E}^{j\ ERA}), \emptyset > \rightarrow^{i-1} < (\mathcal{P}^E \oplus \mathcal{T}^{i-1\ E}, \mathcal{E}^{i-1\ ERA}, (E_i^{ERA}, \ldots, E_j^{ERA})), \emptyset >$$

[14] For the definition of the transition relation $\mapsto^i$ see Definition 58.

for any $i \leq j$ and

$$(\mathcal{P}^E \oplus \mathcal{T}^{i-1\ E}, \mathcal{E}^{i-1\ ERA}, E_i^{ERA}, \ldots, E_j^{ERA}) \vdash_{Se\ e} \ L^E$$

for any L in M_i with $i \leq n$.

As an intermediate step we prove that:

Lemma 6.5.1 *M_i is a refined stable model of $(\mathcal{P} \oplus \mathcal{T}^{j-1}) \uplus E_i$ iff*

$$(\mathcal{P}^E \oplus \mathcal{T}^{i-1\ E}, \mathcal{E}^{i-1\ ERA}, E_i^{ERA}, \ldots, E_j^{ERA}) \vdash_{Se\ e} \ L^E$$

for any L in M_i.

proof The two valued interpretation M_i is a refined stable model of $(\mathcal{P} \oplus \mathcal{T}^{i-1}) \uplus E_i$ iff

$$(\mathcal{P} \oplus \mathcal{T}^{i-1}) \uplus E_i \vdash_{Se\ e} \ L$$

for any $L \in M_i$ for some inference relation $\vdash_{Se\ e}$.

For any L in M_i, the inference

$$(\mathcal{P}^E \oplus \mathcal{T}^{i-1\ E}, \mathcal{E}^{i-1\ ERA}, E_i^{ERA}, \ldots, E_j^{ERA}) \vdash_{Se\ e} \ L^E$$

is equivalent to

$$(\mathcal{P}^E \oplus \mathcal{T}^{i-1\ E}) \uplus E_i^{ERA} \vdash_{Se} \ L^E$$

which, in turn, is equivalent to

$$((\mathcal{P}^E \oplus \mathcal{T}^{i-1\ E}) \uplus (E_i^{ERA})^R \vdash_{Se} \ L^E$$

where $((\mathcal{P}^E \oplus \mathcal{T}^{i-1\ E}) \uplus E_i^{ERA})^R$ is the DyLP obtained from $(\mathcal{P}^E \oplus \mathcal{T}^{i-1\ E}) \uplus E_i^{ERA}$ according to the steps illustrated in Section 6.3.2 (we recall that $E_i^{ERA} = E_i^E \cup \{tickE\}$)

According to the two translations ^{R} and ^{E} (whose composition will be denoted ^{RE}) there is a one-to-one correspondence between the rules of $(\mathcal{P} \oplus \mathcal{T}^{i-1}) \uplus E_i$ and $((\mathcal{P} \oplus \mathcal{T}^{i-1}) \uplus E_i \cup \{tickE\})^{RE}$.

Namely:

- For any rule $\tau : L \leftarrow B_1, \ldots, B_n$ in $\mathcal{P} \uplus E_1$ with $L \neq assert(\eta)$ and $\tau \neq not\ assert(\tau)$: the rule $\tau^E : L^E \leftarrow B_1^E, \ldots, B_n^E$ belongs to $((\mathcal{P} \oplus \mathcal{T}^{i-1}) \uplus E_i \cup \{tickE\})^{RE}$. .
- For any rule $assert(\eta) \leftarrow B_1, \ldots, B_n$ in $\mathcal{P} \uplus E_1$ with $L \neq assert(\eta)$ and $\tau \neq not\ assert(\tau)$: the rule $assert(\eta^{RE}) \leftarrow B_1^{RE}, \ldots, B_n^{RE}, tickE$ belongs to $((\mathcal{P} \oplus \mathcal{T}^{i-1}) \uplus E_i \cup \{tickE\})^{RE}$. .

- For any rule $not\,assert(\eta) \leftarrow B_1,\ldots,B_n$ in $\mathcal{P} \uplus E_1$ with $L \neq assert(\eta)$ and $\tau \neq not\,assert(\tau)$: the rule $not\,assert((\eta^{RE}) \leftarrow B_1^{RE},\ldots,B_n^{RE}$ belongs to $((\mathcal{P} \oplus \mathcal{T}^{i-1}) \uplus E_i \cup \{tickE\})^{RE}$. .

By the principle of partial evaluation (see Theorem A.2.1) we can delete the atom $tickE$ from the body of any rule in $((\mathcal{P} \oplus \mathcal{T}^{i-1}) \uplus E_i \cup \{tickE\})^{RE}$. without changing the semantics.

Again by the principle of substitution (see Theorem A.2.1) we obtain:

$$(\mathcal{P} \oplus \mathcal{T}^{i-1}) \uplus E_i \vdash_{Se} \ L \Leftrightarrow ((\mathcal{P} \oplus \mathcal{T}^{i-1}) \uplus E_i \cup \{tickE\})^{RE} \vdash_{Se} \ L^E$$

for any $L \in \mathcal{L}_{assert}$. Thus M_i is a refined stable model of $(\mathcal{P} \oplus \mathcal{T}^{i-1}) \uplus E_i$ iff

$$((\mathcal{P} \oplus \mathcal{T}^{i-1}) \uplus E_i \cup \{tickE\})^{RE} \vdash_{Se} \ L^E$$

as desired. ♢

We proceed now with the proof of the theorem by induction on j.

Basic step For $j = 1$ all we have to prove is that M_1 is an evolving stable model of $\mathcal{P}$ iff

$$(\mathcal{P}^E, \emptyset, E_1^{ERA}) \vdash_{Se\ e} \ L^E$$

for any $L \in M_i$.

By definition 46, M_1 is an evolving stable model of $(\mathcal{P}, E_1)$ iff M_1 is a refined stable model of $\mathcal{P} \uplus E_1$ i.e.

$$\mathcal{P} \uplus E_1 \vdash_{Se} L$$

for any $L \in M_1$. Hence we obtain the thesis as a particular case of Lemma 6.5.1.

Inductive step Let us assume that $\mathcal{M}^j$ is an evolving stable model of $(\mathcal{P}, \mathcal{E}^j)$ at state j if and only if:

$$< (\mathcal{P}^E, \emptyset, \mathcal{E}^{j\ ERA}), \emptyset > \rightarrow^{i-1} < (\mathcal{P}^E \oplus \mathcal{T}^{i-1\ E}, \mathcal{E}^{i-1\ ERA}, (E_i^{ERA}, \ldots, E_j^{ERA})), \emptyset >$$

for any $i \leq j$ and

$$(\mathcal{P}^E \oplus \mathcal{T}^{i-1\ E}, \mathcal{E}^{i-1\ ERA}, E_i^{ERA} \ldots E_j^{ERA}) \vdash_{Se\ e} \ L^E$$

for any L in M_i with $i \leq j$.

We have to prove that $\mathcal{M}$ is an evolving stable model of $(\mathcal{P}, \mathcal{E})$ at state $j+1$ if and only if:

$$< (\mathcal{P}^E, \emptyset, \mathcal{E}^{j\ ERA}), \emptyset > \rightarrow^{i-1} < (\mathcal{P}^E \oplus \mathcal{T}^{i-1\ E}, (E_i^{ERA}, \ldots, E_j^{ERA})), \emptyset >$$

for any $i \leq j+1$ and

$$(\mathcal{P}^E \oplus \mathcal{T}^{i-1\ E}, \mathcal{E}^{i-1\ ERA}, E_i^{ERA} \ldots E_j^{ERA}) \vdash_{Se\ e} \ L^E$$

for any L in M_i with $i \leq j+1$.

We divide the proof of the double implication in two parts.

$\Rightarrow$ Let us assume that:

$$< (\mathcal{P}^E, \emptyset, \mathcal{E}^{j+1\ ERA}), \emptyset > \rightarrow^{i-1} < (\mathcal{P}^E \oplus \mathcal{T}^{i-1\ E}, (E_i^{ERA}, \ldots, E_{j+1}^{ERA})), \emptyset >$$

for any $i \leq j+1$ and

$$(\mathcal{P}^E \oplus \mathcal{T}^{i-1\ E}, \mathcal{E}^{i-1\ ERA}, E_i^{ERA} \ldots E_j^{ERA}) \vdash_{Se\ e} \ L^E$$

for any L in M_i with $i \leq j+1$.

We have to prove that $\mathcal{M}^{j+1}$ is an evolving stable model of $(\mathcal{P}, \mathcal{E}^{j+1})$. By inductive hypothesis we already know that $\mathcal{M}^j$ is an evolving stable model of $(\mathcal{P}, \mathcal{E}^j)$ i.e. M_i is a refined stable model of $(\mathcal{P} \oplus \mathcal{T}^{i-1}) \uplus E_i$ for any $i \leq j$ it remains to prove that M_{j+1} is a refined stable model of $(\mathcal{P} \oplus \mathcal{T}^j) \uplus E_{j+1}$. This follows as a particular case of Lemma 6.5.1.

$\Leftarrow$ Assuming that $\mathcal{M}$ is an evolving stable model of $(\mathcal{P}, \mathcal{E})$ at state $j+1$, then, by inductive hypothesis:

$$< (\mathcal{P}^E, \emptyset, \mathcal{E}^{j\ ERA}), \emptyset > \rightarrow^{i-1} < (\mathcal{P}^E \oplus \mathcal{T}^{i-1\ E}, \mathcal{E}^{i-1\ ERA}, (E_i^{ERA}, \ldots, E_j^{ERA})), \emptyset >$$

Since the incoming input programs do not affect the computation of the transition, we can add the input program E_{j+1} in the tail of the sequence of input programs and conclude:

$$< (\mathcal{P}^E, \emptyset, \mathcal{E}^{j+1\ ERA}), \emptyset > \rightarrow^{i-1} < (\mathcal{P}^E \oplus \mathcal{T}^{i-1\ E}, \mathcal{E}^{i-1\ ERA}, (E_i^{ERA}, \ldots, E_{j+1}^{ERA})), \emptyset >$$

for any $i \leq j$.

It remains to prove it also for $i = j+1$. By definition of the transition relation $\rightarrow^n$ all we have to prove is:

$$< (\mathcal{P}^E \oplus \mathcal{T}^{j-1\ E}, \mathcal{E}^{j\ ERA}, E_{j+1}^{ERA}), \emptyset > \rightarrow^G < (\mathcal{P}^E \oplus \mathcal{T}^{j\ E}, \mathcal{E}^{j\ ERA} \oplus E_{j+1}^{ERA}, \emptyset), \emptyset >$$

for some set of actions G.

By the inductive hypothesis, taking $i = j$ we know that, for any $L \in \mathcal{L}_{assert}$:

$$(\mathcal{P}^E \oplus \mathcal{T}^{j-1\ E}, \mathcal{E}^{j-1\ ERA}, E_j^{ERA}) \vdash_{Se\ e} \quad L^E$$

iff $L \in M_j$.

In particular:

$$(\mathcal{P}^E \oplus \mathcal{T}^{j-1\ E}, \mathcal{E}^{j-1\ ERA}, E_j^{ERA}) \vdash_{Se\ e} \quad assert(\eta^E)$$

iff $assert(\eta) \in M_j$

Since the only actions executed by the considered ERA system are assertions, let G_j be the set of all the atoms of the form $assert(\eta^E)$, such that $assert(\eta)$ belongs to M_j. Then $\mathcal{T}_j^E$ is the set of rules η^E such that $assert(\eta^E)$ belongs to G_i. So, by the definition of the updating function $up/2$, the system transits into a state where the unique changes are to update the current program with T_j, and to consume one input program, i.e.

$$< (\mathcal{P}^E \oplus \mathcal{T}^{j-1\ E}, \mathcal{E}^{j\ ERA}, E_{j+1}^{ERA}), \emptyset > \rightarrow^{Gi} < (\mathcal{P}^E \oplus \mathcal{T}^{j\ E}, \mathcal{E}^{j\ ERA} \oplus E_{j+1}^{ERA}, \emptyset), \emptyset >$$

as desired.

It remains to prove that

$$(\mathcal{P}^E \oplus \mathcal{T}^{i-1\ E}, \mathcal{E}^{i-1\ ERA}, E_i^{ERA} \ldots E_{j+1}^{ERA}) \vdash_{Se\ e} \quad L^E$$

for any L in M_i with $i \leq j+1$. The assertion is already proved by inductive hypothesis for $i \leq j$. It remains to prove it for $i = j+1$, i.e. we have simply to prove:

$$(\mathcal{P}^E \oplus \mathcal{T}^{j\ E}, \mathcal{E}^{j\ ERA}, E_{j+1}^{ERA}) \vdash_{Se\ e} \quad L^E$$

for any L in M_{j+1}.

If $\mathcal{M}^{j+1}$ is an evolving stable model of $(\mathcal{P}, \mathcal{E}^{j+1}$ then M_{j+1} is a refined stable model of $(\mathcal{P} \oplus \mathcal{T}^j \uplus E_{j+1})$. Hence we obtain the thesis as a particular case of Lemma 6.5.1.

$\diamondsuit$

6.5.2 ERA as an action description language

As it comes out from the discussion of Section 6.5.1, every Evolp program can be directly translated in ERA. This is clearly also the case for EAPs that are Evolp macro programs. As it results from Section 5.3 every expression in an EAP is either a static rule, a dynamic rule, an inertial declaration or the initialization of a set of fluents.

A static rule has the form

$$F \leftarrow Body.$$

where F is a fluent literal and *Body* is a set of fluent literals. Thus, according to definition 60, static rules remain unchanged while being translated from EAPs to ERA. A dynamic rule is an expression of the form

$$\textbf{effect}(\tau) \leftarrow Cond. \qquad (\omega)$$

where τ is a static rule of the form

$$F \leftarrow Body$$

and *Cond* is a set of fluents. The dynamic rule (ω) above is a macro expression for the set of Evolp rules

$$\begin{aligned} F &\leftarrow Body,\ event(\tau). \\ assert(event(\tau)) &\leftarrow Cond. \\ assert(not\ event(\tau)) &\leftarrow event(\tau), not\ assert(event(\tau)) \end{aligned}$$

Thus, according to definition 60, the rule (ω) is translated by the following set of ERA rules:

$$\begin{array}{lll} F \leftarrow Body,\ event(\tau). & & \\ \textbf{On}\ tickE & \textbf{If}\ Cond & \textbf{Do}\ assert(event(\tau)). \\ \textbf{On}\ tickE & \textbf{If}\ event(\tau), not\ assert(event(\tau)) & \textbf{Do}\ assert(not\ event(\tau)). \end{array}$$

An inertial declaration is an expression of the form **inertial($\mathcal{K}$)** (where $\mathcal{K}$ is a set of fluents) and it stands for the set (where Q ranges over $\mathcal{K}$):

$$assert(prev(Q)) \leftarrow Q. \qquad assert(not\ prev(Q)) \leftarrow not\ Q.$$

And it is translated by the set of active rules:

$$\begin{array}{lll} \textbf{On}\ tickE & \textbf{If}\ prev(Q) & \textbf{Do}\ assert(Q). \\ \textbf{On}\ tickE & \textbf{If}\ not\ prev(Q) & \textbf{Do}\ assert(not\ Q). \end{array}$$

Finally, the initialization of a set of fluents is an expression of the form **initialize($\mathcal{F}$)**

(where F is a set of fluents) and it stands for the set of rules $Q \leftarrow prev(Q)$ for any fluent Q in $\mathcal{F}$. This set of rules remains unchanged when translated in ERA.

Since evolving action programs can be directly encoded in ERA, ERA itself can be used for encoding reactive systems (like agents) that *perform* actions but also (by EAPs) for describing (and thus *simulating*) the possible effects of these actions. It is hence possible to write simulations of the environment, in which the system acts, and also to run these simulations together with the reactive system, thus obtaining a simulation of the interaction system-environment.

6.5.3 Other updates languages

The semantics of a system written in either *LUPS* [APPP02], *EPI* [EFST01], *KUL* [Lei03], or *KABUL* [Lei03] can be viewed as a pair $< \mathcal{P}, \mathcal{S} >$ where $\mathcal{P}$ is a DyLP and $\mathcal{S}$ is set of statements. The semantics of an update language α is given by a transitions relation $\rightarrow^{\alpha}$ relating an initial pair $< \mathcal{P}, S >$ to a resulting pair $< \mathcal{P}', S' >$. Formally:

$$< \mathcal{P}, S > \rightarrow^{\alpha} < \mathcal{P}', S' >$$

The transition from $\mathcal{P}$ to $\mathcal{P}'$ if given by the effect of the executed commands which are determined by the statements in S whose preconditions are satisfied. These conditions depend on $\mathcal{P}$. Additionally, *EPI* and *KABUL* are also sensible to *external observations* (which can be assimilated to the events of ERA).

The mechanisms for determining the transition from S to S' depend on the specific language. First of all, in all languages but *EPI*, new statements can be externally added to S'. In the cases of *LUPS* and *KUL*, the statements may be either *volatile* or *persistent*. A volatile statement lasts for a single transition and then expires. Hence a volatile statement in S does not belong to S', unless added again from the outside. A persistent statement, instead, persists from one state to the successive ones unless deleted by another specific statement.

In the case of *EPI*, instead, all the statements are persistent and belong to the initial set of statements. Commands, instead, are volatile or persistent. A volatile command in a statement is executed every time the preconditions of the statement are satisfied, while a persistent command, once triggered by a statement, is executed at every state, even if the preconditions of the triggering statement are no longer satisfied.

Finally *KABUL*, like ERA and Evolp , allow commands to assert, retract and update statements by allowing statements (rather than simply inference rules) as arguments of commands. Hence the transition from S to S' is influenced also by the executed commands.

6.5.4 Updates language LUPS

The language LUPS [APPP02] has a special importance since it is historically the first LP updates language proposed.

The idea behind LUPS is that an initial program P is updated from the outside by resorting to updating statements. From the standpoint of ERA, we would say that the statements (active rules) of LUPS are only in the input programs, and not in the original program. However, some of this statements can be *permanent*, i.e. once introduced, they remain in the program.

The syntax of the LUPS statements is the following:

$$
\begin{array}{rrrr}
1 & \mathit{assert}\ \tau & \mathit{when} & B_1,\ldots,B_n. \\
2 & \mathit{retract}\ \tau & \mathit{when} & B_1,\ldots,B_n. \\
3 & \mathit{assert\_event}\ \tau & \mathit{when} & B_1,\ldots,B_n. \\
4 & \mathit{retract\_event}\ \tau & \mathit{when} & B_1,\ldots,B_n. \\
5 & \mathit{always}\ \tau & \mathit{when} & B_1,\ldots,B_n. \\
6 & \mathit{always\ assert\_event}\ \tau & \mathit{when} & B_1,\ldots,B_n. \\
7 & \mathit{cancel}\ \tau & \mathit{when} & B_1,\ldots,B_n.
\end{array}
$$

where τ is a LP rule and $B_1,\ldots,B_n$ is a conjunction of literals. The meaning of statements 1 and 2 is that of asserting (resp. retracting) the rule τ in the underlying $\mathcal{P}$ if the condition, $B_1,\ldots,B_n$ is satisfied *at the instant the statement is introduced* (in the ERA standpoint, in the state corresponding to the input program containing the statement).

The meaning of statements 3 and 4 is still that of asserting (resp. retracting) the rule τ if $B_1,\ldots,B_n$ is satisfied but this assertion (resp. retraction) lasts for *one state only* (in the state after the action is executed).

The meaning of statements 5 and 6 is the same of, respectively, statements 1 and 3, but the command is persistent, i.e. from that moment onwards, whenever the precondition $B_1,\ldots,B_n$ is satisfied, the command is executed. Finally the meaning of statement 7 is to delete any persistent statement with argument τ

The semantic of *LUPS* is given by translating the program into a DyLP. Two different semantics have been proposed for LUPS. The main difference lies in the concept of satisfaction of the preconditions of statements. In the original proposal (see [APPP02]) a precondition was satisfied if it was true in *at least* one dynamic stable model of the underlying *DyLP*. In [Lei01] a modified semantics was proposed where a command was executed iff the condition was true in the *all* the dynamic stable models.

Unlike in ERA and Evolp , in LUPS the statements themselves cannot be updated (apart from the possibility to make them permanent). It is not possible, for instance, to retract a permanent statement once introduced, nor to update its preconditions has it is done in ERA with inhibition rules. The reason is that the argument of a command

can be only an inference rule and not a command.

We will provide a hint (without formal proofs) of how it is possible to encode the behavior of the commands of *LUPS* in ERA by defining equivalent actions. The LUPS statement

$$Command\ \tau\ when\ Cond.$$

where τ is a logic programming rule , has an equivalent in the ERA active rule

$$\textbf{On}\ tickE\ \ \textbf{If}\ Cond\ \textbf{Do}\ trA(Commad(\tau)).$$

where $trA(Command(\tau))$ is the translation of the command, i.e. an *ERA* action whose execution is equivalent to the execution of $Command(\tau)$. The LUPS commands *assert* and *retract* have their equivalent in the homonymous ERA actions.

The action translating the command *assert_event* can be defined as:

$$assert\_event(H \leftarrow B)\ \textbf{is}\ \|(raise(eventE(H \leftarrow B)), assert(H \leftarrow B, eventE(H \leftarrow B)))$$

where $eventE(H \leftarrow B)$ is a new event. This action updates the knowledge base with the rule

$$H \leftarrow B, eventE(H \leftarrow B).$$

and raises the event $eventE(H \leftarrow B)$. So, in the state immediately after, the effect is that of updating the knowledge base with $H \leftarrow B$. However, the event $eventE(H \leftarrow B)$ immediately expires, and thus the body of the rule is no longer satisfied, until the event $eventE(H \leftarrow B)$ is raised again, i.e. until the same command is again executed. In that case, the rule

$$H \leftarrow B, eventE(H \leftarrow B).$$

is also part of a more recent update, hence, by virtue of Proposition 3.7.2 the first update can be ignored.

Encoding the command *retract_event* (i.e. non persistent retraction) is less straightforward, indeed we need to change the structure of the rules of the underlying DyLP, by replacing every rule $L \leftarrow H$ in the original program by the rule

$$L \leftarrow B, not\ (retractE(L \leftarrow B)).$$

where $retractE(L \leftarrow B)$ is a new literal.

Then, the action *retract_event* is defined in the following way:

$$retract\_event(H \leftarrow B)\ \textbf{is}\ raise(retractE(H \leftarrow B)$$

When this action is executed it raises the event $retractE(L \leftarrow B)$. This implies that

the body of the rule

$$L \leftarrow B, not\,(retractE(L \leftarrow B)).$$

is not satisfied and the result is equivalent to the deletion of the rule. However, the event $retractE(L \leftarrow B)$ immediately expires and hence the rule is active again in the following state unless (persistently or temporarily) retracted again.

The persistent statements 5 and 6 can be translated as the pair

$$\begin{array}{rl} & \textbf{On}\ tickE\ \ \textbf{If}\ B_1,\ldots,B_n\ \textbf{Do}\ assert(\tau). \\ assert(& \textbf{On}\ tickE\ \ \textbf{If}\ B_1,\ldots,B_n\ \textbf{Do}\ assert(\tau)\quad). \end{array}$$

and, respectively,

$$\begin{array}{rl} & \textbf{On}\ tickE\ \ \textbf{If}\ B_1,\ldots,B_n\ \textbf{Do}\ assert\_event(\tau). \\ assert(& \textbf{On}\ tickE\ \ \textbf{If}\ B_1,\ldots,B_n\ \textbf{Do}\ assert\_event(\tau)\quad). \end{array}$$

The first elements of the pair encode the volatile statement, while the *assert* actions make these statements persistent by asserting them as part of the program.

Finally, the statement

$$cancel(\tau)\ when\ Cond.$$

has the effect of deleting any persistent statements asserting τ (either a persistent or a volatile assertion). The definition of action $cancel(\tau)$ is as follows:

$$cancel(\tau)\ \textbf{is}\ raise(cancelE(\tau)).$$

Moreover, the program must have the active rule:

$$\begin{array}{rl} \textbf{On}\ cancelE(\tau)\ \ \textbf{Do}\ \|(& retract(\ \textbf{On}\ tickE\ \ \textbf{If}\ Cond\ \textbf{Do}\ assert(\tau)), \\ & retract(\ \textbf{On}\ tickE\ \ \textbf{If}\ Cond\ \textbf{Do}\ assert\_event(\tau))\quad). \end{array}$$

It is interesting to explore the behavior of these rules. The command *cancel* requires the retraction of *all* the persistent statements asserting τ (either as a persistent or volatile rule) despite of its precondition *Cond*. So, no specific precondition *Cond* is given as argument of the action *cancel*. According to the semantics of action definitions, provided in Section 6.3.3, when several action definitions are provided, the program non-deterministically executes *one* of the possible actions. Hence, the behavior of the following (wrong) definition of $cancel(\tau)$:

$$\begin{array}{rl} fake\_cancel(\tau)\ \textbf{is}\ \|(& retract(\ \textbf{On}\ tickE\ \ \textbf{If}\ Cond\ \textbf{Do}\ assert(\tau)), \\ & retract(\ \textbf{On}\ tickE\ \ \textbf{If}\ Cond\ \textbf{Do}\ assert\_event(\tau))\quad). \end{array}$$

would be to non-deterministically select *one* instance of the variable *Cond* and to ex-

ecute the corresponding instantiated action, i.e. it would retract one of the persistent statements whose head is $assert(\tau)$ or $assert\_event(\tau)$.

The effect of raising the event $cancelE(\tau)$ is, instead, that of triggering *all* the actions in the head of any active rule whose triggering event is $cancelE(\tau)$.

The encoding above illustrates how it is possible to translate *LUPS* commands and programs in ERA.

6.5.5 Updates language *KUL*

The language *KUL* [Lei03] is basically a revision of LUPS where the semantics of the language is revised in order to prevent some counterintuitive behavior. Moreover, the language extends the possibility of having persistent statement (denoted by the keyword *always*) not only to assert but also to retract rules, and introduces a *cancel_retract* command for deleting this kind of persistent statements. The behavior of persistent retraction statement can be simulated in ERA by applying the action *always* to statements of the form 2 and 4. Finally the statement

$$cancel\_retract(\tau)\ when\ Cond.$$

is translated by an action *always* applied to the statement:

$$cancel\_retract'(\tau)\ when\ Cond$$

where the definition of $cancel\_retract'(\tau)$ is:

$$cancel\_retract'(\tau)\ \textbf{is}\ raise(cancelRE(\tau)).$$

and the program has the active rule:

$$\textbf{On}\ cancelRE(\tau)\ \ \textbf{Do}\ \|(\ \ retract(\ \textbf{On}\ tickE\ \ \textbf{If}\ Cond\ \textbf{Do}\ retract(\tau)),\ \ retract(\ \textbf{On}\ tickE\ \ \textbf{If}\ Cond\ \textbf{Do}\ retract\_event(\tau))\ \ \ )\ .$$

Hence, as for *LUPS*, it is possible to encode *KUL* commands and programs in ERA.

6.5.6 Updates language *EPI*

The language *EPI* is also syntactically very close to LUPS. The main differences between the two languages is that the *EPI* statements are persistent by default and a statement of the form

$$Command(\tau)\ when\ Cond.$$

executes the command at *any state* where *Cond* is satisfied. In terms of *ERA*, the statements are part of the main program rather than part of the various input programs.

The commands of *EPI* coincide with those of *LUPS*. Since every command is persistent, the key word *always* (denoted, from now on, *EPI_always* to distinguish it from its LUPS homonymous) has a different meaning. Specifically: "from now on execute this command at every state." it is hence the command rather than the statement that is persistent. For instance, the statement

$$EPI\_always\ assert(\tau)\ when\ Cond.$$

means that the rule τ must be asserted at *every single state* after *Cond* has been verified.

This behavior can be programmed in ERA by defining *EPI_always* as an action whose argument is the command $Command(\tau)$. The effect of action $EPI\_always(Command(\tau))$ is to assert an active rule reiterating $Command(\tau)$ at every state. Formally:

$$EPI\_always(Command(\tau))\ \textbf{is}\ assert(\ \textbf{On}\ tickE\ \ \textbf{Do}\ Command(\tau)).$$

Hence, as for the other updates languages, it is possible to encode *EPI* in ERA.

6.5.7 Updates language *KABUL*

The updates language *KABUL* [Lei03] is the most complete (and complex) of all the existing update languages.

Syntactically *KABUL* extends *KUL* by introducing more commands and by allowing additional kind of preconditions in statements, like other commands, external observations (as in *EPI*) but also the presence or absence of rules. Like Evolp and ERA, *KABUL* allows the argument of a statement to be another statement rather than a rule and that of encoding *inhibition commands* blocking the execution of other commands.

The form of a statement in *KABUL* is the following [15]

$$\begin{aligned} Command(p)\ when\quad & L_i,\ldots,L_{n1},\\ & E:\ E_1,\ldots E_{n2}\\ & R:\ in(r_1),\ldots in(r_{n3})\\ & out(k_1),\ldots,out(k_{n4}) \end{aligned}$$

where p is either a rule τ, a statement S, or an *inhibition command*, the $L_i s$ are either literals or actions, the $E_i s$ are sets of external observations (or events) and the $r_i s$ and $k_i s$ are rules. Intuitively, the meaning is: "if $L_i,\ldots L_{n1}$ is satisfied, and the events (or external observations) $E_1,\ldots E_n$ occur and the rules $r_1,\ldots,r_{n3}$ belong to the current program, and the rules $k_1\ldots,k_{n4}$ do not belong to the current program, then execute $Command(p)$.

[15] In the original syntax of *KABUL* the command *when* was replaced by the notation $\Leftarrow$. For sake of simplicity, we opted here for a syntax more homogeneous with those of other updates languages.

The possible commands are the following.

$$
\begin{array}{l}
(always|konce)\ kassert(\tau)\\
(always|konce)\ kretract(\tau)\\
(always|konce)\ kassert\_event(\tau)\\
(always|konce)\ kretract\_event(\tau)\\
(always|konce)\ kassert(S)\\
(always|konce)\ kretract(S)\\
not\ kassert(\tau)\\
not\ kretract(\tau)\\
not\ kassert\_event(\tau)\\
not\ kretract\_event(\tau)\\
not\ kassert(S)\\
not\ kretract(S)
\end{array}
$$

where τ is any rule and S any statement. The commands with the keyword *not* are inhibition commands. The keyword *konce* introduces a new kind of persistent command whose meaning will be clarified below. We opted for the prefix *k* to distinguish *KABUL* commands from homonomous but slightly different *ERA* actions.

The effect of commands $(always)\ kassert\ (\_event)(\tau)$ and $(alaways)\ kretract(\_event)(\tau)$ is the same of *KUL*. The new keyword *konce* introduces a new kind of persistent command, whose meaning is very close to the complex action *once* already seen in Section 6.4.3. The output of statement

$$konce\ Command(p)\ when\ Cond.$$

is to immediately execute $Command(p)$ if condition *Cond* is true; otherwise the command remains by inertia in the set of statements until *Cond* is satisfied. At that moment, the statement is deleted from the set of statements.

The behavior of commands whose argument is a statement S is slightly different. The command $kassert(S)$ (resp. $kretract(S)$) adds (resp. to removes) the statement S to the new set S'. This addition (resp. retraction) is permanent if S is a persistent command; otherwise it lasts for one state only (as it happens for *assert_event* and *kretract_event* when the argument is a rule). The inhibition commands temporary block the effect of other statements.

There is a similarity between *KABUL*, Evolp and ERA regarding the possibility of manipulating statements by commands. There is also a similarity between *ERA* and *KABUL* (as is was with *EPI*) regarding the sensibility to external events (called observations in *KABUL*) as precondition of rules. Finally, the various command options of *KABUL* somehow recall the action algebra of *ERA*.

However, unlike *Evolp* and ERA, rules and statements are not treated in the same

way. Indeed, there is a difference in the behavior of $Command(S)$ and $Command(\tau)$ where S is a statement and τ is a rule. This difference is originated, as we have seen, by the way non persistent statements are treated.

As the reader may notice, although *KABUL* allows to apply several keywords (or *operators*) to commands, all these operators are *unary* i.e. *they do not allow to combine commands to obtain more complex ones*. In other words, also *KABUL* does not provide an algebra for combining commands (i.e. actions).

As we did with other languages, we provide a way on how *ERA* could replicate the statements of *KABUL*.

The volatile statement

$$\begin{aligned} Command(p)\ when \quad & L_i, \ldots L_{n1}, \\ & E:\ E_1, \ldots E_{n2} \\ & R:\ in(r_1), \ldots in(r_{n3}) \\ & out(k_1), \ldots, out(k_{n4}) \end{aligned}$$

is translated into the following active rule.

$$\begin{aligned} \textbf{On}\ \triangle(E_1, \ldots, E_{n2})\ \textbf{If}\quad & L_i, \ldots L_{n1} \\ & in(r_1), \ldots in(r_{n3}) \\ & not\ out(k_1), \ldots, not\ out(k_{n4}\ \textbf{Do} \\ & Command(p) \end{aligned}$$

where $in(\tau)$ is a new atom stating that rule τ belongs to the program.

The inhibition statement

$$\begin{aligned} not\,Command(p)\ when \quad & L_i, \ldots L_{n1}, \\ & E:\ E_1, \ldots E_{n2}, \\ & R:\ in(r_1), \ldots in(r_{n3}), \\ & out(k_1), \ldots, out(k_{n4}) \end{aligned}$$

is instead translated with the following inhibition rule.

$$\begin{aligned} \textbf{When}\ \triangle(E_1, \ldots, E_{n2}), \quad & L_i, \ldots L_{n1}, \\ & in(r_1), \ldots, in(r_{n3}), \\ & not\ out(k_1), \ldots, not\ out(k_{n4}), \\ & \textbf{Do}\ not\,Command(p) \end{aligned}$$

The atom $in(\tau)$ must be true iff the rule τ belongs to the *KB*. For this reason we initially add to the KB $\mathcal{P}$ the atom $in(\tau)$ for every rule τ appearing in $\mathcal{P}$. However, this is not enough, it is also necessary to guarantee that whenever a rule τ is asserted (resp. retracted) also the atom $in(\tau)$ is asserted (resp. retracted).

The commands $kassert(\tau)$ and $kretract(\tau)$ are then defined in such a way that those atoms are added (resp. retracted) together with rules:

Formally:

$$\begin{array}{rcl} kassert(\tau) & \textbf{is} & \|(assert(\tau), assert(in(\tau))). \\ kretract(\tau) & \textbf{is} & \|(assert(\tau), assert(in(\tau))). \\ kassert\_event(\tau) & \textbf{is} & \|(assert\_event(\tau), assert\_event(in(\tau))). \\ kretract\_event(\tau) & \textbf{is} & \|(retract\_event(\tau), retract\_event(in(\tau))). \end{array} \tag{6.13}$$

If, instead, the argument of the command is a statement. its translation is the following.

$$\begin{array}{rcl} kassert(S) & \textbf{is} & assert(\|(assert S')). \\ kretract(S) & \textbf{is} & assert(\|(S')). \\ kassert(\{S1, S2\}) & \textbf{is} & |(assert(S_1), assert(S_2)). \\ kretract(\{S1, S2\}) & \textbf{is} & |(retract(S_1), retract(S_2)). \end{array}$$

This definition is due to the fact that some statements (the non volatile ones) in *KABUL* are seen as single ERA active rule, while others (the persistent ones) are seen as a set of two different statements.

Indeed, the statements of the form:

$$\begin{array}{rl} always\ Command(p)\ when & L_i, \ldots L_{n1}, \\ & E:\ E_1, \ldots E_{n2} \\ & R:\ in(r_1), \ldots in(r_{n3}) \\ & out(k_1), \ldots, out(k_{n4}) \end{array}$$

are translated into the following pair of active rules.

$$\begin{array}{rll} t: & \textbf{On}\ \triangle(E_1, \ldots, E_{n2})\ \textbf{If} & L_i, \ldots L_{n1} \\ & & in(r_1), \ldots in(r_{n3}) \\ & & not\ out(k_1), \ldots, not\ out(k_{n4})\ \textbf{Do} \\ & & Command(p). \\ & \textbf{On}\ tickE\ \textbf{Do} & assert(t) \end{array}$$

This translation is analogous to the corresponding into statements in *LUPS*.

Finally the statements of the form:

$$\begin{array}{rl} konce\ Command(p)\ when & L_i, \ldots L_{n1}, \\ & E:\ E_1, \ldots E_{n2} \\ & R:\ in(r_1), \ldots in(r_{n3}) \\ & out(k_1), \ldots, out(k_{n4}) \end{array}$$

are translated into the following pair of active rules.

$$ll \ \textbf{On} \triangle(E_1,\ldots,E_{n2}) \quad \textbf{If } L_i,\ldots L_{n1} \ in(r_1),\ldots in(r_{n3}) \ not\,out(k_1),\ldots,not\,out(k_{n4} \ \textbf{Do} \ Command(p) \tag{6.14}$$

$$\textbf{On } tickE \ \textbf{If } not\,Command(p) \ \textbf{Do} \quad assert(t). \tag{6.15}$$

where t is the following active rule.

$$\textbf{On} \triangle(E_1,\ldots,E_{n2}) \ \textbf{If} \ L_i,\ldots L_{n1} \ in(r_1),\ldots in(r_{n3}) \ not\,out(k_1),\ldots,not\,out(k_{n4}) \ \textbf{Once} \ Command(p) \ )$$

Basically the precondition of the rule is tested by rule 6.14, in case it is satisfied the command is executed (and hence the precondition of rule 6.15 is not satisfied and the rule is not executed). Otherwise, the rule b asserts an active rule that uses the complex action *once* (see Section 6.4.3) to delete itself once it executed $Command(p)$.

The encoding above completes the series of translations of LP updates languages into ERA.

6.6 Comparisons to ECA formalisms and event languages

There exists several other proposals of ECA formalisms besides the ones we have compared to ERA. Most of them are related to active database systems like, for instance: AMIT [SJG04], Rule Core [Rul], JEDI [GNF98] and SQL triggers [WC96a]. Although these approaches implement most (but, as stated below, not all) of the functionalities of ERA, they are mainly procedural (do not have a declarative semantics). It is hence more problematic than in ERA to investigate the formal properties of programs written in these languages and establish formal results. For instance, it is hard to establish formal translations of one of these languages into another one (as we do, in Section 6.5.1, were a formal translation of Evolp into ERA is defined and its correctness is proved). Moreover, a formal declarative semantics greatly simplifies the understanding of the framework by the user. A declarative semantics is, for instance, among the features demanded in [Sem07] as a basic requirement of a Semantic Web services language. For this reason we focus our comparisons on declarative approaches. For an overall discussion on languages for programming active databases we refer to [AE03, WC96a].

Other examples of ECA formalisms are Semantic Web-oriented ECA languages like

Active XML [ABM+02] and XChange [BPS04]. The syntax of both languages is close to that of XML. Active XML is an extension of XML coping with active rules. It has a procedural semantics and does not support complex events. The language XChange is closer to ERA, since it has a LP-like semantics and allows to define active rules with complex actions and complex events.

However, neither XChange, nor any of the ECA languages listed above has the evolving capabilities showed by ERA. Also, none of these languages can be directly integrated with an action description language as in the case of ERA and EAPs.

Among ECA formalisms two languages deserve a closer comparison with ERA: DALI [CT04] and Statelog [LML98, LHA95, LMA96]. DALI is a LP-like languages and hence close in spirit wit ERA. DALI is, among the existing ECA languages, the one showing more similarities with ERA as discussed below. Statelog [LML98, LHA95, LMA96] is an ECA paradigm for representing ACID transactions describing a database transiting between states (indexed by natural numbers) as effect of external events. Besides ECA languages, it is also interesting to compare the event algebra of ERA with that of similar event algebras, as we do in Section 6.6.3.

6.6.1 The agent-oriented language DALI

DALI [CT04] is an agent-oriented ECA language that shares with ERA a logic programming based semantics and other similarities which make it worthwhile to closely compare the two paradigms.

The basic structure of DALI are active rules of the form:

$$E_1 \dots E_n :> Body.$$

where $E_1, \dots E_n$ are sequences of either internal or external events and $Body$ is a sequence $Obj_1, \dots, Obj_n$ of atoms where any Obj_i is an atom that may represent an action to be executed, a goal to be reached, or a condition to be tested within the KB of the agent. Formally:

$$Body ::= seq << Obj >> \qquad Obj ::= Atom \mid Action_A \mid Goal.$$

The intuitive meaning of an active rule of the form

$$E_1 \dots E_n :> Body.$$

is: "when the events $E_1 \dots E_n$ occurs, do $Body$ provided that the conditions in $Body$ are satisfied". Hence active rules in DALI are ECA rule where conditions and the actions may be interleaved. An external event occurs when some information comes from the

outside, while internal events E_i are raised by rules of the form:

$$E_i : - \ Condition.$$

where *Condition* is a sequence of atoms that may represent the beliefs of the agent, and events occurred in the past (called “past evens") or any condition to be tested within the KB of the agent. Formally:

$$Condition ::= seq << C_i >> \quad C_i ::= Atom \,|E_P\,|Belief$$

The truth value of atoms occurring in the *Body* of active rule is established according to logic programming inference rules of the form $Atom : -Atom_1, \ldots, Atom_n.$ where $Atom_i$ are new atoms.

Actions represent activities to be executed. An action may have referring “action rules", i.e. logic programming-like expressions specifying the preconditions for the execution of that actions which have the form

$$Action_A : -Precondition$$

where *Precondition* is a sequence of atoms. Finally, a goal atom $Goal_G$ represents a desirable achievement. A goal may specify actions and subgoals to be undertaken by rules of the form:

$$Goal_G :> Body$$

where *Body* is a sequence of actions or new (sub goals) atoms. The intuitive meaning of the rule above is “to achieve $Goal_G$ one have to achieve all the subgoals and execute all the actions in *Body*"

At a semantic level, DALI has an “evolutionary semantics" that defines the semantics of the language by successive transformation of the language into definite logic programs. Common traits of ERA and DALI are:

- The notion of internal and external events, and a memory of events occurred in the past,
- The fundamental construct of ECA rules, although in DALI actions and conditions may be interleaved,
- An underlying logic programming semantics, knowledge representation and inference system,
- The possibility to define complex actions, achieved in ERA by the action algebra and action definitions and in DALI by goals.

The main differences, at a general level, between DALI and ERA is that the former is totally focused on a logic programming-like rule approach, while ERA mixes a (dy-

namic) logic programming syntax and semantics for inference and reactive behavior with an algebraic approach to events and actions.

In some cases, the algebraic approach grants to ERA more expressivity w.r.t. DALI. For instance, although both languages have a notion of past events, DALI has a limited capability of combining events occurring in different instants. In particular, it is not clear how to replicate in DALI the complex event $A(e_1, e_2, e_3)$ whose intuitive meaning is "event e_1 occurred in the past, event e_3 occurs now, and event e_2 did not occur in the meanwhile" nor how to represent a negative event *not e* i.e. the "non occurrence" of event e. Regarding the underlying logic programming framework, DALI relies on the least Herbrand model semantics of definite logic programs, while ERA relies on DyLPs either the refined or well founded semantics. Hence, unlike ERA, DALI does not handle default negation nor negation in the head of rules.

Another relevant point of comparison are the evolution capabilities of the two languages. The semantics of DALI is called " evolutionary semantics". According to this semantics, the meaning of a program is given by the least Herbrand model of the a definite logic program obtained by the original DALI program, via a program transformation. The transformed program changes from state to state according to the occurred events and executed actions. Although this approach recall the semantics of ERA (and also that of logic programming updates languages) these is a substantial difference. In ERA inference and active rules are updated as a result of the evolution of the programs while inference and active rules in DALI do not change as a result of the evolution of the program. Hence, the evolution of the transformed programs does not correspond to an evolution of the knowledge (inference rules) and behavior (active rules) of the system. As a matter of fact, the evolutionary semantics of DALI and the underlying DyLP framework of ERA and updates languages are orthogonal as stated by the authors of DALI in [CT04] So, the capabilities of self evolution of *ERA*, i.e. assertion, retraction and inhibition of inference and active rules, has no equivalent in DALI.

On the other hand, DALI has many interesting features not yet developed (or, at lest, not yet fully developed) in ERA, particularly regarding its agent-oriented nature. For instance, DALI implements specific mechanism for communication among agents, an agent architecture close in spirit to the BDI (Belief, Desire, Intention) [Rao96] approach, and plug-in planning functionalities. As such, the two languages target, for now, different applications. The integration of the features above listed in ERA are a ground of investigation for future developments of the language.

Statelog

Statelog [LML98, LHA95, LMA96] is an ECA paradigm for representing ACID transactions describing a database transiting between states (indexed by natural numbers) as effect of external events. Statelog is an extension of Datalog, where literals have an extra argument which is the state to which the truth value of the literal is referred. A

first version of Statelog is called *Flat Statelog* and does not consider the possibility to express nested transactions. The main construct of Statelog are *progressive Statelog rules* i.e. rules of the form

$$[S_0]H \leftarrow [S_1]B_1, \ldots, [S_n]B_n$$

where H is an atom, the various $B_i s$ are literals and $S_i s$ are the indexes of the state at which the truth value of the corresponding literal must be evaluated and $S_0 \geq S_i$ for each i. A *Statelog program is a finite set of progressive Statelog rules*. The considered Statelog programs are *state stratified* [LHA95], a generalization of the notion of local stratification [AB94]. The considered semantics of a program is its unique perfect model semantics [AB94].

An event in Statelog is simply an atom true at some given state S. External events are special atoms that are added to the current database $\mathcal{D}[S]$ at some state S corresponding to the instant when the external event occurs. An internal event e is an atom which is true at some state S by the effect of a Statelog rule of the form $[S]e \leftarrow B$. Composite events are obtained by Statelog rules linking events occurring at the same or at different states.

Internal actions are special atoms that have the effect, when inferred at some state S of asserting or deleting facts from the database at the state $S+1$. External actions are also special atoms having effects on the external environment.

Within Flat Statelog, given an initial database $\mathcal{D}$ at the initial state n_i which incorporates all the facts as well as a list of external events, a semantic is given for computing the perfect model $M[n_{i+1}]$ at the successive state induced by the Statelog program P and from that the successive state $M[n_{i+2}]$ and so on, until the sequence of state stabilizes reaching a fix point where $M[n_{i+n}] = M[n_{i+n+1}]$. Such states are called *final states* and denote the end of a computation induced by an external event. Notes that, despite of their names, there can be several final states during the lifespan of a system. Each time a new external event occurs the system starts to change until it reaches a new final state and so on.

The execution of external action is delayed until the system reaches a final state.

Unlike the (simpler) Flat version, Statelog provides the possibility to specify complex transactions as head of rules (those transactions are called *nested transactions* or *procedures*).

A procedure π is a subprogram of the Statelog program. The language allows the declaration of a procedure π by the syntax:

$$proc\ \pi\ \{H_1 \leftarrow B_1\ \ldots\ H_n \leftarrow B_n\}$$

Every program has a procedure *main* representing the main body of the program. A procedure can have other (sub) procedures as heads of its rules while only the procedure *main* admits external events in the body of its rules, i.e. external events cannot be

processed within nested transactions.

The computation of a procedure is isolated from the rest of the program. This is achieved by beginning a sequence of states where only the actions that appears as heads of rules in the procedures are executed. If a procedure is aborted all its actions are undone.

Besides actions and external events, there are other special atoms in the language called *control relations* whose meaning is to handle the execution of the sub procedures. Namely the control relation $[S.\pi.0]BOT$ (Begin Of Transaction) begins the execution of a procedure π at a state S starting from the state with a subsequence of states the first one being $(S.\pi.0)$ where only the actions of π are executed and $[S]EOT$ (End Of Transaction) terminates the execution of a procedure, $[S]running$ denotes that a procedure is running while *commit* and *abort* causes the current procedure to end either commit or abort. Finally, relations $commit(\pi)$ and $abort(\pi)$ states that the procedure π was, respectively, either committed or aborted.

A first evident difference between Statelog and ERA is that *Statelog* is expressively designed for defining ad executing actions, while this subject is, by now, not touched in ERA. Moreover, *Statelog* has a more procedural view than ERA. First of all states are part of the syntax (they are arguments of predicates) in Statelog while leaved as an implicit concept in ERA. Indexing of states is used for encoding complex events in Statelog, a purpose achieved in ERA by defining an event algebra. The control of the execution transactions is handled by special purpose atoms in Statelog while ERA has no explicit concept of transaction. Another important difference is that Statelog uses sub-programs for specifying complex (trans)actions while ERA resort to an action algebra.

Given the difference of the basic ideas, it is not easy to compare the expressivity of the two approaches, mainly because of the use of states as arguments of literals and control relations in Statelog.

Above all, as the other paradigms presented here, Statelog does not consider the possibility to update rules but only facts.

6.6.2 Process algebras

The action algebra introduced in ERA is inspired by works on process algebras like CCS [Mil89], CSP [Hoa85] and TCC [SJG96]. Not surprisingly, there are several similarities between the action algebra of ERA and process algebras. The common basic choice is to define an algebra of operators for building complex operations (either processes or actions) by elementary ones. Also many of the operators in ERA, like sequential and parallel execution, the IF operator and action definitions, are inherited by process algebras. Another common choice is to rely on transition systems for defining the effect of an operation.

However, there are also important differences marking the algebra of ERA as some-

thing different from a classical process algebra. These differences come from the specific features of ERA, in particular: the capability to execute several actions concurrently. Regarding parallel execution, process algebras usually rely on the notion of *interleaving* which means that the parallel execution of two or more processes amounts to chose and execute a basic operation of one process at each transition, allowing to switch from the execution of a process to that of another one.

Within ERA, instead, a set of actions may be truly concurrently executed by executing, in a single step, the basic operations of each action. This difference is substantial since, as shown in Section 6.4.2, the execution of a set of actions is not, in general, semantically equivalent to the sequential execution of its individual actions.

Another important difference is the way non executable actions are treated. Usually, within process algebras, when a process cannot be executed (i.e. it has no applicable transition) its execution is halted, but this is not considered as a failure and the process is not discarded. Within ERA, instead, non executable actions are discarded. The reason to this difference is that, within process algebras, this behavior of *storing* non executable process is used for opening processes awaiting for some condition to become true (for instance, the incoming of an external input). The approach of ERA is different since actions are supposed to be triggered by active rules, that are hence in charge for reacting to changes and external inputs. What is missing in process algebras (and coped by ERA) is a clear notion of event and the possibility of defining complex events and program reactions to such events, together with a notion of knowledge representation and reasoning and evolving capabilities analogous to the ones of ERA.

6.6.3 Snoop and other event languages

The event algebra of ERA is basically a fragment of the Snoop event algebra [AC03, Ada02]. Snoop is a framework for event specification and detection. The main issues marking the difference between Snoop and the event algebra proposed in Section 6.3.2 are:

- Cumulative events: (Non ground) events that occur more than once in a given period marked by two other events, and whose final output are the values of the parameters of the occurred events. For instance, the cumulative event $A*(E_1, E_2(X), E3)$ occurs with output $X_1, \ldots X_n$ when E_3 occurs and between E_1 and E_3 the events $E_2(X_1), \ldots E_2(X_n)$ occur.

- Periodic events: Events periodically occurring in a given period marked by two other events. For instance, the periodical event $p(e_1, [1hr], e_3)$ occurs every hour starting from the occurrence of e_1 until e_2 occurs.

Since the work in this thesis is more focused on evolution and algebras of operators, we leave to future development the issue of extending the language with the full expressivity of Snoop. Another possibility would be to develop an alternative version

of ERA which is orthogonal to the choice of the underlying event specification and detection framework. In this respect, it would be possible to consider also alternative choices of the basic event framework such as, for instance, COMPOSE[GJO92], ADAM [PDG92] and ACOOD [Ber91].

6.7 Conclusions and open issues

We identified a set of desirable features for an ECA language whose goal is to program knowledge bases with reactive behavior. Namely: a declarative semantics, the capability to express complex events and actions in a compositional way, and that of receiving external updates, and performing self updates to data, inference rules and active rules. We also pointed on the possibility to define transactions and that of describing the effects of actions as additional interesting features. We argued that Evolp does not match all these features, nor does any known ECA paradigm or LP updates language. In particular Evolp does not provide facilities for combining events and actions. Hence, we defined the logic programming ECA language ERA, which, like Evolp , is based on the semantics of DyLPs, but it also incorporates active rules and algebras for events and actions. We also provided the ERA with a declarative semantics based on the refined semantics of DyLPs (for inferring conclusions) and a transition system (for the execution of actions).

We showed that ERA extends Evolp in the sense that any Evolp program has a direct translation in ERA. This also allows for the use the action description language EAP described of Chapter 5, as a part of the ERA formalism. We then compared ERA with other LP update formalisms illustrating how to simulate these languages with ERA.

In Section 6.4.1 we briefly discuss the possibility that actions could be non executable, i.e. that their execution may actually *fail*. Within ERA, non executable (failed) actions are simply discarded. Moreover, when executing a complex action, if one of its sub actions is not executable the whole actions results as non executable and its execution is interrupted, i.e. the action fail. ERA has no specific means for treating failure such as, for instance, recovering from the failure of an action by executing another action in its place (a form of alternative execution) or to reverse a partially executed complex action. Recovering from a failure is a key issue when executing *transactions*. A transaction is: *a sequence of information exchange and related work (such as database updating) that is treated as a unit for the purposes of satisfying a request and for ensuring database integrity.* [Sea]. Transactions are the standard way to perform operations on database systems , operations involving transfers of money or other resources and, in general, all those operations supporting some kind of reversion in case of failure. A crucial aspect for any kind of transaction is to support mechanisms for reversing partially executed failed transactions. In order to apply ERA to database and resource transfer applica-

tions, the most important open issue is to extend it with transactional capabilities and specific ways to treat the failure of (trans)actions.

An introductive discussion to the topic of a *Transaction ERA* is among the subjects of Chapter 7, and is subject of current work.

Chapter 7

Conclusions

The goal of the thesis has been the development of a logic programming based framework for programming AI applications provided with reasoning, reactive and evolving capabilities. To achieve this goal, we first established theoretical foundations by defining the semantics of a logic programming paradigm for dealing with updates of knowledge, i.e. dynamic logic programs. These basic results were then used for addressing the issues of reasoning about the effect of actions, programming reactive behavior and executing actions.

Dynamic logic programs are sequences of logic programs where the first program is seen as the initial knowledge base and the others are subsequent updates. Within DyLPs, rules can have either positive or negative literals in their heads. Conflicts among rules with complementary heads are bound to occurs. According to the principle of causal rejection, whenever a conflict occurs among rules in different updating programs, the rules in less recent updates are rejected. Most of the existing semantics for DyLPs in the literature are extensions of the classical stable model semantics, taking into account the causal rejection principle. These semantics are strictly related to each other and coincide on large classes of programs. However none of them was completely satisfactory. In particular none of them complies with the principle of immunity to tautologies, i.e. for all of them there are examples where the addition of a tautological rule of the form $L \leftarrow Body$ where L occurs in $Body$ changes the semantics of the considered DyLP.

The first (sub)goal of the thesis was to define a suitable extension to DyLPs of the stable model semantics of generalized logic programs. The first step in this direction was the definition of a new principle, called the refined extension principle, that should be satisfied by any stable model-like semantics for DyLPs based on causal rejection. It turns out that the violation of the principle of immunity to tautology is a special case of violation of the refined extension principle. We then defined a semantics for DyLPs that extends the stable model semantics and complies with both the causal rejection and the refined extension principle. This semantics is a refinement of the other semantics for DyLPs based on causal rejection, in the sense that any model according to the new

semantics is also a model according to the others. For this reason the new semantics was named refined stable model semantics for DyLPs.

Another important property of the semantics is that of extending in a natural way the well supported semantics to the case of DyLPs. The well supported semantics is an alternative and equivalent characterization of the stable model semantics of normal logic programs. We extended the definition of well supported model to DyLPs, and proved that the set of well supported models of a DyLP coincides with its set of models according to the refined semantics.

Yet one more achievement was that of extending the refined-well supported semantics from DyLPs to multidimensional dynamic logic programs. MDyLPs are a generalization of DyLPs. While DyLPs are sequences of logic programs, MDyLPs are partially ordered multisets of programs. We extended the notion of well supported model to MDyLPs and found an equivalent fixpoint characterization.

After having defined a proper extension of the stable model semantics to DyLPs, the next step was to define a suitable extension of the well founded semantics to DyLPs. In the case of normal logic programs, the well founded semantics is a skeptical polynomial approximation of the stable model semantics, in the sense that the well founded model of a program is a subset of any of its stable models. We extended the well founded semantics to DyLPs by providing an alternative fixpoint definition and shown the relationships between the refined and the well founded semantics for DyLPs extends the one between the stable model and well founded semantics of normal logic programs, i.e. the well founded model of a DyLP is a subset of any of its refined models. Moreover, the complexity of the computation of the well founded semantics of a DyLP is, as in the case of normal logic programs, polynomial w.r.t. the number of rules of the system.

After establishing these basic results, we faced the issue of how to use DyLPs for representing and reasoning about the effect of actions. To face this issue we used the already existing logic program updates language Evolp. Evolp is a language that is capable of specifying the evolution of a DyLP by asserting new rules. The evolution of a program is also influenced by external inputs in the form of Evolp programs. We defined an action description language as a macro language built on top of Evolp. The new language, named evolving action programs, enables to describe the effect of actions on the environment by rules. We proved that the existing action description languages $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ can be modularly translated into EAPs hence showing that EAPs are at least as expressive as these languages. Moreover, we showed the capability of EAPs to handle changes in the environment and the way it is affected by actions.

Having provided a tool for reasoning about the effect of actions we were still short off showing how to specify the execution of actions. To face this issue we defined an Event-Condition-Action language with inference logic rules, with active rules specifying what actions to execute when a given event occurs, with an event and an action al-

gebra for defining complex events and actions in terms of simpler ones, with inhibition rules preventing actions from being executed and, finally, with self evolving features (i.e. the capability to autonomously assert inference, active and inhibition rules). The event algebra allows to combine events, possibly occurring in difference instants, in order to define new ones. The action algebra allows to specify the execution of several basic actions like the assertion and retraction of rules, the execution of external actions having effects on the external environment, or to raise an internal event that is then detected by the system. The algebra also allows to specify whether several actions must be sequentially or concurrently executed, or also to specify conditions determining the execution of one action instead of another one. Both for events and for actions it is possible to give a name to a complex combination of events (resp. actions) and hence to use the action (resp. event) name in other expressions.

The semantics of the new language, called ERA (after Evolving Reactive Algebraic Programs) is based, for the inference and reactive part, on the (either refined or well founded) semantics of DyLPs. This is achieved by transforming any ERA program into a DyLP where active and inhibition rules are turned into logic programming rules with, respectively, positive and negative literals in the head. Regarding actions, their semantics is given by transition rules describing the effects of actions on the system. We showed than that every Evolp (and hence EAPs) program has a straightforward translation into ERA, and hence the latter can be considered as an extension of the former.

Summarizing, the original contribution to the initial required features are

Reasoning

- The stable model and the well founded semantics for normal logic programs have been extended to the case of dynamic logic programs, thus obtaining the refined semantics for dynamic and multidimensional dynamic logic programs and the well founded semantics for dynamic logic programs. Both the semantics allow for reasoning about knowledge which is constantly updated. The possible conflicts between older and newer rules are automatically solved by the developed logic framework

  While the refined semantics allows for more complex, NP-hard, form of reasoning requiring the global consistency of the knowledge base, the well founded semantics is a skeptical polynomial approximation of the refined one, allowing to quickly process data (although at the cost of loosing some inference power) and capable to deal with inconsistent knowledge bases

- The logic programming update language Evolp (provided with an underline refined semantics) has been used for defining an action description formalism of EAPs. EAPs can be used for representing the effects of actions and predicting the possible evolutions of the world given an initial state and a

sequence of performed actions

Reactivity The Event-Condition-Action, dynamic logic programming based, language ERA provides ways to define active rules specifying how and under which conditions a system should react to a incoming event by executing actions. Events and actions can be either basic or complex ones. Complex events and actions are defined by basic ones by means of, respectively, an event and an action algebra. The conditions are conjunctions of literals that must be evaluated according to facts and LP inference rules in the KB of the system.

Evolution

- The capability to automatically handle updates of rules of the underlying DyLP framework provides the basis for enabling evolution of the framework
- Evolving action programs can autonomously integrate updates to the very rules describing the effects of actions in order to handle changes in the described environment
- The ERA language allows the user to specify conditions under which active and inference rules, event and actions definitions can be asserted and retracted. It is also possible to specify exceptions to active rules by asserting inhibition rules. The user does not need to take care of solving the possible conflicts among older and newer rules, since this task is automatically solved by the underlying DyLP framework

The three features above, are not stand-alone achievements. Indeed, the inference system of ERA may incorporate either the refined or the well founded semantics for DyLPs. Moreover, evolving action programs can be simulated by ERA. Hence, ERA incorporates the features of reasoning, reactive and evolving capabilities as required by the initial goal of the thesis.

7.1 Current and future work

As just argued, in our opinion the thesis achieved the goal of defining an LP-based language for reasoning and reacting to events by executing actions. Of course this does not close the research area of reactivity in knowledge bases, and some issues are left open in this thesis. Some of them are related to possible extensions of the language to deal with new problems and application areas. Some others are just about further studies related to intermediate results developed in the thesis, that were not pursued since they were not crucial for the main goals of the work.

To start it is possible to further study the basic semantics in order for providing either further properties of the well supported semantics for DyLPs or further refinements to match new principles and properties.

Also, the well founded semantics is not defined for the case of multidimensional dynamic logic programs. This is so in this thesis, since it was not a central issue for the goal of defining a language for both reasoning about and executing actions. But this extension could be particularly profitable for Semantic-Web and distributed data base related applications, enabling to merge and update data and knowledge from different sources, solving possible contradictions by assigning priorities on most reliable sources, and then querying the resulting KB according to its well founded, possibly paraconsistent, model.

A possible way to define a well founded semantics is by an alternating fixpoint operator obtained by combining the anti-monotonous operators of Definition 36 that extend the dynamic and refined semantics to the multidimensional case. Further investigation to understand the formal properties and detect possible drawbacks of this approach are in order.

Another topic left open is that of defining action query languages for EAPs, capable of dealing with the issues of prediction, postdiction and planning both with complete and incomplete knowledge.

Currently, the theoretical framework developed in the thesis have been implemented up to the point of the refined and well founded semantics for DyLPs. It still remains to implement the refined semantics for MDyLPs and the ERA-EAPs language for reasoning about and executing actions

ERA was established as a language for reasoning and executing actions. The basic language was defined and compared to extant approaches. This, of course, does not preclude extensions of the language, to deal with further issues. One such extension is that of allowing multidimensional updates capabilities. Regarding this, it could be possible to take advantage on the development related to the multidimensional updates langauge $KABUL^m$

Also, the event algebra of ERA could be extended with more general classes of complex events, in order to obtain at least the same expressivity of event specification languages such as Snoop An alternative solution could be to elaborate a version of the language parametrical w.r.t. the underlying event specification language.

Possibly the most relevant extension of ERA would to provide the language with specific features for defining and executing transactions. Transactions are the standard way to perform operations on database systems, operations involving transfers of money or other resources and, in general, all those operations supporting some kind of rollback in case of failure. Indeed, this is a subject of current work, and it is profitable to spend here some more words on how it could be faced and solved.

To match the requirement of preserving the integrity of data, transactions usually

satisfy the *ACID properties (Atomicity, Consistency, Isolation and Durability)*. Atomicity requires the transaction to be executed as a unit, i.e. either all the operations involved are committed or none is. To achieve Atomicity, the standard way to handle the failure of an ACID transaction is by a *rollback*, i.e. by restoring the state of the system before the execution of the failed transaction[1]

Consistency requires that a transaction either creates a new state where the consistency of the data is preserved or fails without changing the data. Isolation requires that a transaction in process and not yet committed must remain isolated from any other transaction, i.e. to execute a set of transactions T is equivalent to execute one-by-one all the transactions of T in some order. Durability requires that, even in the event of a failure and system restart, the data is available in the state it was when the last transaction committed. Durability is usually achieved by low-level software management and is hence outside the scope of this study.

Although ACID properties are essential features for transactions performing internal changes in database systems, they imposes very strict demands [HK87] that are not always suitable for performing transitions involving system networks like, for instance, Web Sources, as argued in [TSI02].

The reason is that Atomicity and Isolation require conditions that cannot be guaranteed by loosely-coupled networks, where the single nodes have a high degree of independence, and communications are slow and fail frequently. Atomicity requires all the steps of the transactions to rollback when the transaction fails, while Isolation requires that the data involved in a transaction is unaccessible until the transaction either commits of fails. Transactions for which Atomicity cannot be guaranteed are those involving activities in the real world like, for instance, dispensing money from an automated teller machine or sending an email. There is no possibility to undo these actions, often called *real* [HK87].

Moreover, some transactions may involve iterated information exchange between different actors such as data bases, web services, human agents and so on, potentially lasting for hours or days. This transactions are called *long running transactions* [TSI02] or also *sagas* [DT87]). In this scenario, to demand for a complete lock of data (demanded to comply with Isolation) until the transaction is executed is too restrictive. We refer to transactions for which the demands of the ACID properties are too restrictive as to *irreversible transactions*.

An irreversible transaction is usually formed by sub transactions that are required to obey to the ACID properties and irreversible actions like messages or real actions. Since when an irreversible transaction fails it is note possible to rollback to the initial state, another mechanism for handling failure is required. The usual approach (see, for instance, [HK87], [DT87], [CGV+02]) is to resort to another operation, called *compensation* [HK87], leading into a state that is considered semantically equivalent to the initial

[1] This is usually implemented by recording in a stack all the basic operations executed by a transactions and, in case of failure, to *undo* the operations in reverse order [HGM02].

(for this reason a compensation is also known as *semantical rollback*)

Note that a rollback of an ACID transaction is just a form of compensation. In general, an irreversible transaction T is defined and implemented as a combination of ACID sub transactions and, possibly, external iterations involving receiving external inputs and performing external actions. To execute T is, in general, equivalent to execute a sequence of sub-transactions $T_1, \ldots, T_n$. Each sub transaction T_i has a compensation C_i. A correct execution of T implies either to execute

$$T_1, \ldots, T_n$$

or, in case a failure to start the execution of the various compensations starting from the last committed sub transaction T_i, i.e. to execute

$$T_1, \ldots, T_i, C_i, \ldots C_1$$

This notion of transactions and compensation activities was first introduced in the seminal paper [DT87].

Similarly to actions as defined in ERA, transactions are basic or complex operations that permanently change the state of the system. The first notion required by transactions that is formally missing in ERA is that of *failure*. However, the notion of non executable actions can be seen as a prototype for the notion of failure of a transaction. Even more important is to specify ways to *handle a failure*. Indeed, in ERA, there is no notion of rollback and it is hence not possible to reverse a (trans)actions, and hence Atomicity cannot be accomplished. Even Isolation is not preserved since in ERA actions are concurrently executed and, as shown in Section 6.4.2, the concurrent execution of actions is, in general, not equivalent to any sequential execution of those actions. Thus, it is not possible to implement ACID transaction in ERA.

As for rollback, there is also not a clear notion of compensation in ERA. Finally, in several applications it is useful to have the possibility to specify an alternative way to execute a transaction if the main stream of the transaction fails.

Both ACID and irreversible transactions are actions with specific functionalities for handling failure: alternative execution and either rollback (for ACID transactions) or compensation (for irreversible transactions). The most natural way to define irreversible and ACID transaction in ERA could hence be to extend the action algebra with new operators implementing failure handling functionalities. In this extended algebra of *Transaction ERA*, a transaction may either be an ACID transaction or an irreversible transaction.

Such an extension of ERA, in which we are currently working, will result in a unique declarative language, capable of executing both ACID and irreversible transactions, together with evolving capabilities.

Bibliography

[AB94] K. R. Apt and R. N. Bol. Logic programming and negation: A survey. *The Journal of Logic Programming*, 19 & 20:9–72, May 1994.

[ABB04] José Júlio Alferes, Federico Banti, and Antonio Brogi. From logic programs updates to action description updates. In *CLIMA V*, pages 52–77, 2004.

[ABB06] José Júlio Alferes, Federico Banti, and Antonio Brogi. An event-condition-action logic programming language. In Michael Fisher, Wiebe van der Hoek, Boris Konev, and Alexei Lisitsa, editors, *Logics in Artificial Intelligence, 10th European Conference, JELIA 2006*, pages 29–42. Springer, 2006.

[ABBL04] J. J. Alferes, F. Banti, A. Brogi, and J. A. Leite. Semantics for dynamic logic programming: a principled based approach. In *Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-7)*, volume 1730 of *LNAI*, Berlin, 2004. Springer.

[ABBL05] J. J. Alferes, F. Banti, A. Brogi, and J. A. Leite. The refined extension principle for semantics of dynamic logic programming. *Studia Logica*, 79(1), 2005.

[ABLP02] J. J. Alferes, A. Brogi, J. A. Leite, and L. M. Pereira. Evolving logic programs. In S. Flesca, S. Greco, N. Leone, and G. Ianni, editors, *JELIA'02*, LNAI, 2002.

[ABM+02] S. Abiteboul, O. Benjelloun, I. Manolescu, T. Milo, and R. Weber. Active xml: Peer-to-peer data and web services integration. *citeseer.ist.psu.edu/article/abiteboul02active.html*, 2002.

[AC03] Raman Adaikkalavan and Sharma Chakravarthy. Snoopib: Interval-based event specification and detection for active databases. In *ADBIS*, pages 190–204, 2003.

[Ada02] Raman Adaikkalavan. Snoop event specification formalization algorithms and implementation using an interval bases semantics. Master's thesis, The Univeristy of Texas at Arlington, July 2002.

[AE03] Juan M. Ale and Mauricio Minuto Espil. Active rules and active databases: Concepts and application. In *Effective Databases for Text & Document Management*, pages 234–261. 2003.

[ALP+98a] J. J. Alferes, J. A. Leite, L. M. Pereira, H. Przymusinska, and T. C. Przymusinski. Dynamic logic programming. In J. L. Freire, M. Falaschi, and M. Vialres-Ferro, editors, *Proceedings of the 1998 Joint Conference on Declarative Programming (AGP-98)*, pages 393–408, La Coruña, Spain, 1998.

[ALP+98b] J. J. Alferes, J. A. Leite, L. M. Pereira, H. Przymusinska, and T. C. Przymusinski. Dynamic logic programming. In A. Cohn, L. Schubert, and S. Shapiro, editors, *Proceedings of the 6th International Conference on Principles of Knowledge Representation and Reasoning (KR-98)*, pages 98–111, San Francisco, 1998. Morgan Kaufmann Publishers.

[ALP+00a] J. J. Alferes, J. A. Leite, L. M. Pereira, H. Przymusinska, and T. C. Przymusinski. Dynamic updates of non-monotonic knowledge bases. *Journal of Logic Programming*, 45(1–3), 2000.

[ALP+00b] J. J. Alferes, J. A. Leite, L. M. Pereira, H. Przymusinska, and T. C. Przymusinski. Dynamic updates of non-monotonic knowledge bases. *The Journal of Logic Programming*, 45(1–3):43–70, September/October 2000.

[AP96] J. J. Alferes and L. M. Pereira. *Reasoning with logic programming*, volume 1111 of *Lecture Notes in Artificial Intelligence and Lecture Notes in Computer Science*. Springer-Verlag Inc., New York, NY, USA, 1996.

[APP+99] J. J. Alferes, L. M. Pereira, T. Przymusinski, H. Przymusinska, and P. Quaresma. Preliminary exploration on actions as updates. In M. C. Meo and M. V. Ferro, editors, *Proceedings of the 1999 Joint Conference on Declarative Programming (AGP-99)*, 1999.

[APPP02] J. J. Alferes, L. M. Pereira, H. Przymusinska, and T. Przymusinski. LUPS: A language for updating logic programs. *Artificial Intelligence*, 132(1 & 2), 2002.

[Apt96] K. R. Apt. *From logic programming to Prolog*. Prentice-Hall, Inc., Upper Saddle River, NJ, 1996.

[BAB04] F. Banti, J. J. Alferes, and A. Brogi. The well founded semantics for dynamic logic programs. In Christian Lemaître, editor, *IBERAMIA-9*, LNAI, 2004.

[BAB05] Federico Banti, José Júlio Alferes, and Antonio Brogi. Operational semantics for dylps. In *EPIA*, pages 43–54, 2005.

[Ban05a] F. Banti. An implementation of the well founded semantics for dylps, 2005. Available at http://centria.di.fct.unl.pt/∼ banti/Implementation.htm.,.

[Ban05b] F. Banti. An implementation of the well founded semantics for dylps, 2005. Available at http://centria.di.fct.unl.pt/∼ banti/Implementation.htm.,.

[Ber91] M. Berndtsson. Acood: An approach to an active object oriented dbms, 1991.

[BFL99] F. Buccafurri, W. Faber, and N. Leone. Disjunctive logic programs with inheritance. In D. De Schreye, editor, *ICLP-99*, Cambridge, November 1999. MIT Press.

[BG97] C. Baral and M. Gelfond. Reasoning about effects of concurrent actions. *Journal of Logic Programming*, 31:85–118, 1997.

[BGP97] C. Baral, M. Gelfond, and Alessandro Provetti. Representing actions: Laws, observations and hypotheses. *Journal of Logic Programming*, 31, April–June 1997.

[BL96] Chitta Baral and Jorge Lobo. Formal characterization of active databases. In *Logic in Databases*, pages 175–195, 1996.

[BPS04] F. Bry, P. Patranjan, and S. Schaffert. Xcerpt and xchange - logic programming languages for querying and evolution on the web. In *ICLP*, pages 450–451, 2004.

[BW90] J. C. M. Baeten and W. P. Weijland. *Process algebra*. Cambridge University Press, New York, NY, USA, 1990.

[CGV+02] Mandy Chessell, Catherine Griffin, David Vines, Michael J. Butler, Carla Ferreira, and Peter Henderson. Extending the concept of transaction compensation. *IBM Systems Journal*, 41(4):743–758, 2002.

[CL03] Jan Carlson and Björn Lisper. An interval-based algebra for restricted event detection. In *FORMATS*, pages 121–133, 2003.

[CT04] Stefania Costantini and Arianna Tocchio. The dali logic programming agent-oriented language. In *JELIA*, pages 685–688, 2004.

[Dix95a] J. Dix. A classification theory of semantics of normal logic programs I: Strong properties. *Fundamenta Mathematicae*, 22(3):227–255, 1995.

[Dix95b] J. Dix. A classification theory of semantics of normal logic programs II: Weak properties. *Fundamenta Mathematicae*, 22(3):257–288, 1995.

[DLV00] DLV. The DLV project - a disjunctive datalog system (and more), 2000. Available at http://www.dbai.tuwien.ac.at/proj/dlv/ .

[DP96] C. V. Damásio and L. M. Pereira. Default negation in the heads: why not? In R. Dyckhoff, H. Herre, and P. Schroeder-Heister, editors, *Int. Ws. Extensions of Logic Programming (ELP-96)*, volume 1050 of *LNAI*, pages 103–117. Springer Verlag, November 1996.

[DP98] C. V. Damásio and L. M. Pereira. A survey on paraconsistent semantics for extended logic programas. In D. M. Gabbay and Ph. Smets, editors, *Handbook of Defeasible Reasoning and Uncertainty Management Systems*, volume 2, pages 241–320. Kluwer Academic Publishers, 1998.

[DPP97] 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, *Nonmonotonic Extensions of Logic Programming*, LNAI 1216, pages 1–36. Springer, Berlin, 1997.

[DT87] Umeshwar Dayal and Irving L. Traiger, editors. *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, California, May 27-29, 1987*. ACM Press, 1987.

[ea00] T. Eiter et al. Planning under incomplete information. In *CL'2000*, 2000.

[EFST01] T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. A framework for declarative update specifications in logic programs. In Bernhard Nebel, editor, *Proceedings of the seventeenth International Conference on Artificial Intelligence (IJCAI-01)*, pages 649–654, San Francisco, CA, 2001. Morgan Kaufmann Publishers, Inc.

[EFST02a] T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. On properties of semantics based on causal rejection. *Theory and Practice of Logic Programming*, 2:711–767, 2002.

[EFST02b] T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. On properties of update sequences based on causal rejection. *Theory and Practice of Logic Programming*, 2002. to appear.

[EK76] M. Van Emden and R. Kowalski. The semantics of predicate logic as a programming language. *Journal of ACM*, 4(23):733–742, 1976.

[FAA03] F.Banti, J. J. Alferes, and A.Brogi. A principled semantics for logic program updates. In *Non monotonic reasoning, actions, changes NRAC'03*, 2003. To appear.

[Fag94] François Fages. Consistency of Clark's completion and existence of stable models. *Journal of Methods of Logic in Computer Science*, 1:51–60, 1994.

[Gär92] P. Gärdenfors, editor. *Belief Revision*, number 29 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.

[Gel92] A. Van Gelder. The alternating fixpoint of logic programs with negation. *Journal of Computer and System Sciences*, 1992.

[GJO92] N. Gehani, H. V. Jagadish, and O.Shmueli. pages 81–90. 1992.

[GL88] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *5th International Conference on Logic Programming*. MIT Press, 1988.

[GL90] M. Gelfond and V. Lifschitz. Logic programs with classical negation. In Warren and Szeredi, editors, *7th International Conference on Logic Programming*, pages 579–597. MIT Press, 1990.

[GL93] M. Gelfond and V. Lifschitz. Representing actions and change by logic programs. *Journal of Logic Programming*, 17:301–322, 1993.

[GL98a] M. Gelfond and V. Lifschitz. Action languages. *Electronic Transactions on AI*, 16, 1998.

[GL98b] E. Giunchiglia and V. Lifschitz. An action language based on causal explanation: Preliminary report. In *AAAI'98*, pages 623–630, 1998.

[GLL+97] E. Giunchiglia, J. Lee, V. Lifschitz, N. Mc Cain, and H. Turner. Representing actions in logic programs and default theories: a situation calculus approach. *Journal of Logic Programming*, 31:245–298, 1997.

[GLL+03] E. Giunchiglia, J. Lee, V. Lifschitz, N. McCain, and H. Turner. Nonmonotonic causal theories. *Artificial Intelligence*, 2003.

[GM99] Paul Gastin and Michael W. Mislove. A truly concurrent semantics for a simple parallel programming language. In *CSL*, pages 515–529, 1999.

[GNF98] G.Cugola, E. D. Nitto, and A. Fuggetta. Exploiting an event-based infrastructure to develop complex distributed systems. In *20th Int. Conf. on Software Enginieering*, 1998.

[GRS91] 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.

[HGM02] J. Widom H. García-Molina, J. D. Ullman. *Database Systems: The Complete Book*. Prentice-Hall, 2002.

[Hit03] Pascal Hitzler. Towards a systematic account of different logic programming semantics. In Andreas Günter, Rudolf Kruse, and Bernd Neumann, editors, *Proceedings of the 26th German Conference on Artificial Intelligence, KI2003, Hamburg, September 2003*, volume 2821 of *Lecture Notes in Artificial Intelligence*, pages 355–369. Springer, Berlin, 2003.

[HK87] Garcia-Molina H. and Salem K. *Readings in database systems*, pages 290–300. San Francisco, California, 1987.

[Hoa85] C.A.R. Hoare. *Communication and Concurrency*. Prentice-Hall, 1985.

[Hom04] M. Homola. Dynamic logic programming: Various semantics are equal on acyclic programs. In J. Leite and P. Torroni, editors, *5th Int. Ws. On Comptational Logic In Multi-Agent Systems (CLIMA V)*, pages 227–242. Pre-Proceedings, 2004. ISBN: 972-9119-37-6.

[HS05] Pascal Hitzler and Sibylle Schwarz. Level mapping characterizations of selector-generated models for logic programs. In *Proceedings of the 19th Workshop on (Constraint) Logic Programming, W(C)LP 2005, Ulm, Germany*, 2005. To appear.

[HW05] Pascal Hitzler and Matthias Wendt. A uniform approach to logic rogramming semantics. *Theory and Practice of Logic Programming*, 5(1-2):123–159, 2005.

[KM91] H. Katsuno and A. O. Mendelzon. On the difference between updating a knowledge base and revising it. In J. Allen, Richard Fikes, and E. Sandewall, editors, *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning*, pages 387–394, San Mateo, CA, USA, April 1991. Morgan Kaufmann Publishers.

[KS86] R. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4(1):67–95, 1986.

[LAP00] J. A. Leite, J. J. Alferes, and L. M. Pereira. Dynamic logic programming with multiple dimensions. In L. Garcia and M. Chiara Meo, editors, *Procs. of the APPIA-GULP-PRODE'00 Joint Conference on Declarative Programming AGP'00, La Habana, Cuba*, 2000.

[LAP01] J. A. Leite, J. J. Alferes, and L. M. Pereira. Multi-dimensional dynamic knowledge representation. In T. Eiter, M. Truszczynski, and W. Faber, editors, *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-01)*, volume 2173 of *LNAI*, pages 365–378, Berlin, 2001. Springer.

[Lei97] J. A. Leite. Logic program updates. Master's thesis, Dept. de Informática, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, November 1997.

[Lei01] J. A. Leite. A modified semantics for LUPS. In P. Brazdil and A. Jorge, editors, *Progress in Artificial Intelligence, Proceedings of the 10th Portuguese International Conference on Artificial Intelligence (EPIA01)*, volume 2258 of *LNAI*, pages 261–275, Berlin, 2001. Springer.

[Lei03] J. A. Leite. *Evolving Knowledge Bases*, volume 81 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2003.

[LHA95] B. Ludasher, U. Haman, and L.Lausen Adaikkalav. A logical framework for active rules. In *Proc. of the 7-th international Conference on Management of Data (COMAD)*. Mc Graw Hill, 1995.

[Lif99] V. Lifschitz. *The Logic Programming Paradigm: a 25-Year Perspective*, chapter Action languages, answer sets and planning, pages 357–373. Springer Verlag, 1999.

[LL03] J. Lee and V. Lifschitz. Describing additive fluents in action language C+. In William Nebel, Bernhard; Rich, Charles; Swartout, editor, *Proc. IJCAI-03*, pages 1079–1084, Cambridge, MA, 2003. To Appear.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming*. Springer–Verlag, 2 edition, 1987.

[LMA96] G. Lausen, W. May, and L.Lausen Adaikkalav. Nested transactions in a logical language for active rules. In *International Workshop on Logic in Databases*. Springe, 1996.

[LML98] Bertram Ludäscher, Wolfgang May, and Georg Lausen. Nested transactions in a logical language for activerules. Number 1154 in LNCS, pages 196–222. Springer, 1998.

[LP97] J. A. Leite and L. M. Pereira. Generalizing updates: from models to programs. In *LPKR'97: workshop on Logic Programming and Knowledge Representation*, 1997.

[LP98] J. A. Leite and L. M. Pereira. Iterated logic program updates. In J. Jaffar, editor, *Proceedings of the 1998 Joint International Conference and Symposium on Logic Programming (JICSLP-98)*, pages 265–278, Cambridge, 1998. MIT Press.

[LPR98] H. Levesque, F. Pirri, and R. Reiter. Foundations for the situation calculus. *Linköping Electronic Articles in Computer and Information Science*, 03, 1998.

[LS06] João Leite and Luís Soares. Adding evolving abilities to a multi-agent system. In *CLIMA VII*, pages 246–265, 2006.

[LW92] V. Lifschitz and T. Woo. Answer sets in general non-monotonic reasoning (preliminary report). In B. Nebel, C. Rich, and W. Swartout, editors, *KR-92*, 1992.

[MC07] M.Osorio and V. Cuevas. Updates in answer set programming: An approach based on basic structural properties. *Theory and Practice of Logic Programming*, 7, 2007.

[McC59] J. McCarthy. Programs with commons sense. In *Proceedings of Teddington Conference on The Mechanization of Thought Process*, pages 75–91, 1959.

[McC88] J. McCarthy. *Mathematical logic in artificial intelligence*, pages 297–311. Daedalus, 1988.

[MH69] J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*. Edinburgh University Press, 1969.

[Mila] R. Milner. *Handbook of Theoretical Computer Science, volume B: Formal Models and Semantics*, chapter Operational and algebraic semantics of concurrent processes.

[Milb] R. Milner. *Theoretical Computer Science*, chapter Calculi for synchrony and asynchrony.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[MT91] V. Marek and M. Truszczyński. Autoepistemic logic. *Journal of the Association for Computing Machinery*, 38(3):588–619, 1991.

[PDG92] N. Paton, O. Diaz, and P. Gray. pages 81–90. 1992.

[PP88] H. Przymusinska and T. Przymusinski. Weakly perfect model semantics. In R. Kowalski and K. A. Bowen, editors, *5th International Conference on Logic Programming*, pages 1106–1122. MIT Press, 1988.

[Prz90] T. Przymusinski. Extended stable semantics for normal and disjunctive programs. In Warren and Szeredi, editors, *7th International Conference on Logic Programming*, pages 459–477. MIT Press, 1990.

[QR99] P. Quaresma and I. P. Rodrigues. A collaborative legal information retrieval system using dynamic logic programming. In *Proceedings of the Seventh International Conference on Artificial Intelligence and Law (ICAIL-99)*, ACM SIGART, pages 190–191, N.Y., 1999. ACM Press.

[Rao96] A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In W. van der Velde and J. W. Perram, editors, *Agents Breaking Away (LNAI 1038)*, pages 42–55. Springer-Verlag: Heidelberg, Germany, 1996.

[Rei87] R. Reiter. Nonmonotonic Reasoning. *Annual Review Comput. Science*, 2:147–187, 1987.

[Rew] The rewerse project. *http://rewerse.net/*.

[RF89] A. S. Rao and N. Y. Foo. Formal theories of belief revision. In Hector J. Levesque Ronald J. Brachman and R. Reiter, editors, *Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning*, pages 369–380, Toronto, Canada, May 1989. Morgan Kaufmann.

[RN95] S. Russel and P. Norvig. *Artificial Intelligence A Modern Approach*, page 4. Artificial Intelligence. Prentice Hall, 1995.

[RT88] K. Ross and R. Topor. Inferring negative information from disjunctive databases. *Automated Reasoning*, 4:397–424, 1988.

[Rul] Rulecore. *http://www.rulecore.com*.

[sap04] Sapphire healthcare information system. http://www.intranexus.com/sapphire.htm, 2004.

[SCM+99] S.Abiteboul, C.Culet, L. Mignet, B.Amann, T.Milo, and A. Eyal. Active views for electronic commerce. In *25th Very Large Data Bases Coference Proceedings*, 1999.

[Sea] SearchCIO.com. transaction.

[Sef00] J. Sefranek. A kripkean semantics for dynamic logic programming. In *Logic for Programming and Automated Reasoning(LPAR'2000)*. Springer Verlag, LNAI, 2000.

[Sem07] Semantic web services language requirements. http://www.daml.org/services/swsl/requirements/swsl-requirements.shtml#sec-general, 2007.

[SH87] D. McDermott S. Hanks. Nonmonotonic logic and temporal projection. *Artificial Intelligence*, 33:379–412, (1987).

[SI99] C. Sakama and K. Inoue. Updating extended logic programs through abduction. In M. Gelfond, N. Leone, and G. Pfeifer, editors, *LPNMR-99*, Berlin, 1999. Springer.

[SJG96] Vijay A. Saraswat, Radha Jagadeesan, and Vineet Gupta. Timed default concurrent constraint programming. *Journal of Symbolic Computation*, 22(5/6), 1996.

[SJG04] Vijay A. Saraswat, Radha Jagadeesan, and Vineet Gupta. Amit - the situation manager. *The International Journal on Very Large Data Bases archive*, 13, 2004.

[SMO00] SMODELS. The SMODELS system, 2000. Available at `http://www.tcs.hut.fi/Software/smodels/` .

[Tar55] A. Tarski. A lattice–theoretic fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[TSI02] Mikalsen T., Tai S., and Rouvellou I. Transactional attitudes: reliable composition of autonomous web services. In *Dependable Systems and Networks Conference*, 2002.

[W3C04] W3c semantic web health care and life sciences interest group. http://www.w3.org/2001/sw/hcls/, 2004.

[WC96a] J. Widom and S. Ceri, editors. *Active Database Systems – Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann Publishers, 1996.

[WC96b] Jennifer Widom and Stefano Ceri. *Active Database Systems: Triggers and Rules For Advanced Database Processing.* Morgan Kaufmann, 1996.

[ZF98] Y. Zhang and N. Y. Foo. Updating logic programs. In Henri Prade, editor, *Proceedings of the 13th European Conference on Artificial Intelligence (ECAI-98)*, 1998.

Appendix A

Proofs of theorems from Chapter 3

A.1 Properties of the least model

Through the proofs of theorems and propositions we use several properties of the least model of a definite logic program.

Property A.1.1 *Let P, U and R be definite logic programs, such that, for any rule $\tau \in U$, $B(\tau) \not\subseteq least(P)$, and $H(R) \subseteq least(P)$ then:*

$$least(P \cup U \cup R) = least(P)$$

i.e., $P \cup R \cup U$ is a syntactic extension of P.

proof The *least* operator is monotonous, hence $least(P) \subseteq least(P \cup U \cup R)$. We prove that $least(P)$ is a model of $P \cup U \cup R$. First, $least(P)$ is the least model of P. Moreover, it is a model of U, since, by hypothesis, the body of any rule in U is not satisfied. Finally it is a model of R, since, by hypothesis, the head of any rule in R is in $least(P)$. Since $least(P \cup R \cup U)$ is the least model of $P \cup R \cup U$ then

$$least(P \cup R \cup U) \subseteq least(P)$$

$\diamondsuit$

Since for any tautology τ either $H(R) \in least(P)$ or $B(\tau) \not\subseteq least(P)$ it immediately follows:

Corollary A.1.1 *Let P be a definite logic program and $\mathcal{E}$ be a set of tautologies. Then $least(P \cup \mathcal{E}) = P$.*

Property A.1.2 *Let P be a definite logic program and Ta a program whose rules are tautologies i.e. rules of the form: $A \leftarrow B$ where $A \in B$. Then $P \cup Ta$ is a syntactic extension of P.*

proof Let U be the program consisting of all rules r' in Ta such that $H(r') \notin least(P)$ and R be the program obtained considering all rules r such that $H(r) \in least(P)$. Note that, since for all rules, $r \in Ta$ $H(r) \in B(r)$, the rules of U are such that $B(r) \not\subseteq least(P)$. From this and the definition of R, it follows that P, U and R satisfies the hypothesis of Property A.1.1. Hence, by the thesis of such Property it follows immediately that $P \cup Ta$ is a syntactic extension of P. ♢

Property A.1.3 *Let P be any definite logic program, and τ a definite rules. Let us suppose: $B(\tau) \not\subseteq least(P)$, then: $least(P \cup \{\tau\}) = least(P)$*

The idea of this property is that rules whose body is not satisfies are not relevant for the computation of the least Herbrand model.
proof We know that $least(P \cup \{\tau\}) \supseteq least(P)$. Since $least(P)$ does not support τ, then by minimality: $least(P \cup \{\tau\}) \subseteq least(P)$

♢

Property A.1.4 *Let P be any definite logic program, τ is a rule of P. Let τ_1 be a rule of P such that: $B(\tau') \subseteq least(P \setminus \{\tau\})$ and $hd(\tau) = hd(\tau')$, then:*

$$least(P) = least(P \setminus \{\tau\})$$

proof We know that $L = least(P \setminus \tau) \subseteq least(P)$. Since L is a model of τ' and it satisfies the body of such rule, it follows that $hd(\tau) \in L$, then $L \supseteq least(P)$

♢

Property A.1.5 *Let P be any definite logic program and A an atom. Then A belongs to the least Herbrand model of P iff there exists $P' \subseteq P$ such that P' is a finite set of rules and $A \in least(P')$*

proof

Let us suppose $A \in least(P)$ and let n be the least natural number such that $A \in T_P^n(\emptyset)$. We prove the by induction on n that the property is true for each $A \in T_P^n(\emptyset)$ and for each n.

If $n = 0$ there is nothing to prove. Let us suppose the property proved for n, we prove it for $n+1$. We know that

$$\forall A \in T_P^n(\emptyset)\ it\ exists\ a\ finite\ program\ P' :\ P' \subseteq P \wedge\ A \in least(P')$$

Let $A \in T_P^{n+1}(\emptyset)$. By definition of the T_P operator there exists a rule $\tau :\ A \leftarrow B$ such that $B \subseteq T_P$. Then for each literal $B_i \in B$ there exists, by inductive hypothesis, a finite subprogram P_i of P such that $B_i \in least(P_i)$. Consider the program $P' = \bigcup P_i \cup \{\tau\}$. P' is

the union of a finite number of finite subprograms of P, then it is a finite subprogram of P. Moreover $least(P') \models B$ and $\tau \in P'$. Then $A \in P'$

$\diamondsuit$

A.2 The principle of partial evaluation

In some of the proof we will refer to the *principle of partial evaluation*. This principle is one of the properties listed in [Dix95b] and it is satisfied by most of the existing semantics. Intuitively it asserts that given a program with a rule of the form $a \leftarrow, B$ where c is an atom and B a conjunction of literals, it is possible to substitute the rule above with the set of rules of the form $a \leftarrow B_i, B$ for any rule of the form $c \leftarrow B_i$ in the program without changing the semantics of the program itself. Given a logic program P, let *Sem* be a function mapping a logic program P to a set of interpretations. Any semantics is univocally determined (and univocally determines) such a function. For instance, the function *STABLE*, associated to the stable model semantics, maps a program P to the set of its stable models. The function *LEAST*, associated to the least Herbrand model semantics, maps a program P into single-element set formed by its least Herbrand model i.e. $LEAST(P) = \{least(P)\}$. Finally the function WFS, associated to the well founded semantics, maps a program P into single-element set formed by its well founded model. We can now formalize the principle of partial evaluation.

Definition 61 *Let P be a logic program, Sem a function associated to a semantics and let c be an atom. Let*

$$c \leftarrow B_1, \ldots, c \leftarrow B_n$$

be the set of rules with head c in P. Let P' be program obtained by replacing any rule of the form $a \leftarrow c, B$ with the set of rules

$$\begin{array}{c} a \leftarrow B_1, B. \\ \vdots \\ a \leftarrow B_n, B. \end{array}$$

The semantics associated to Sem satisfies the principle of partial evaluation then:

$$Sem(P) = Sem(P')$$

Most of the known semantics for LPs satisfy this principle.

Theorem A.2.1 *Then the stable and the well founded semantics for normal logic programs satisfies the principle of partial evaluation and so it does the least Herbrand model semantics.*

For an extended dissertation on the subject and formal proofs see [Dix95b]

A.3 Proofs of theorems and propositions

Proof of Proposition 3.3.1: Let $M \in Sem(P \cup \{\tau\})$, then, by definition:

$$M = least(P \cup \{\tau\} \cup Assumtions(P \cup \{\tau\}, M))$$

Using Property A.1.2 we obtain

$$M = least(P \cup Assumtions(P \cup \{\tau\}, M))$$

Hence $P \cup \{\tau\}$ is a refined extension of P. Since the semantics Sem complies with the refined extension principle, this implies $M \in Sem(P)$. Thus $Sem(P \cup \{\tau\}) \subseteq Sem(P)$ as desired. ♢

Proof of Proposition 3.3.2: Recall that, in the stable model semantics, independently of P, the set $Assumptions(P,M)$ is M^-. We simply have to prove that, if M is a stable model of $P \cup E$ and $P \cup E$ is an extension of P w.r.t. M, then M is a stable model of P. If M is a stable model of $P \cup E$, then, by definition of extension

$$M = least(P \cup M^- \cup E) = least(P \cup M^-)$$

This implies M is a stable model of P. ♢

Proof of Proposition 3.3.3: Let $\mathcal{P}$ and $\mathcal{E}$ be dynamic logic programs and Sem a semantics for DLPs that complies with the refined extension principle. Let M be in $Sem(\mathcal{P} \cup \mathcal{E})$. Then:

$$M = least(\rho(\mathcal{P} \cup \mathcal{E}) - Rej(\mathcal{P} \cup \mathcal{E}, M) \cup Assumption(\mathcal{P} \cup \mathcal{E}, M))$$

Splitting the multiset $\rho(\mathcal{P} \cup \mathcal{E}) - Rej(\mathcal{P} \cup \mathcal{E}, M)$ in two parts, namely $\rho(\mathcal{P}) - Rej(\mathcal{P} \cup \mathcal{E}, M)$ and $\rho(\mathcal{E}) - Rej(\mathcal{P} \cup \mathcal{E}, M)$ we obtain

$$M = least(\rho(\mathcal{P}) - Rej(\mathcal{P} \cup \mathcal{E}, M) \cup Assumption(\mathcal{P} \cup \mathcal{E}, M) \cup \mathcal{E}')$$

where $\mathcal{E}' = \rho(\mathcal{E}) - Rej(\mathcal{P} \cup \mathcal{E}, M)$ which means all the rules in $\mathcal{E}'$ are tautologies. Since the addition of tautologies does not change the least model (cf. Property A.1.2) it follows that M is also the least model of

$$\rho(\mathcal{P}) - Rej(\mathcal{P} \cup \mathcal{E}, M) \cup Assumption(\mathcal{P} \cup \mathcal{E}, M)$$

Since Sem complies with the refined extension principle, from such principle it follows that $M \in Sem(\mathcal{P})$. ♢

Proof of Theorem 3.4.1 : Let $\mathcal{P}$ and $\mathcal{E}$ be the two dynamic logic programs $P_1, \ldots, P_n$

and $E_1, \ldots, E_n$ such that $\mathcal{P} \cup \mathcal{E}$ is a refined extension of $\mathcal{P}$ and M is a refined stable model of $\mathcal{P} \cup \mathcal{E}$ i.e.

$$M = least(\rho(\mathcal{P} \cup \mathcal{E}) - Rej(\mathcal{P} \cup \mathcal{E}, M) \cup Assumption(\mathcal{P} \cup \mathcal{E}, M))$$

We have to prove that M is also a refined stable model of $\mathcal{P}$ i.e.

$$M = least(Generator^R(\mathcal{P}, M))$$

Since M is a refined stable model of $\mathcal{P} \cup \mathcal{E}$ then, by definition, M is the least model of the definite logic program: $\mathcal{G}en \cup \mathcal{R}es$ where $\mathcal{G}en$ is the program

$$(\rho(\mathcal{P}) - Rej^S(\mathcal{P}^{E_i}, M)M) \cup Def(\mathcal{P} \cup \mathcal{E}, M)$$

and $\mathcal{R}es$ the program $\rho(\mathcal{E}) - Rej^S(\mathcal{P}^{E_i}, M)M$. From the hypothesis we know that $\mathcal{G}en \cup \mathcal{R}es$ is a syntactic extension of $\mathcal{G}en$. By the definition of syntactic extension we derive that

$$M = least(\mathcal{G}en)$$

If a rule τ belongs to $Rej^R(M)$, then it also belongs to $Rej^S(\mathcal{P}^{E_i}, M)M$, because there will be anyway a rule rejecting it, hence

$$Rej^R(M) \subseteq Rej^S(\mathcal{P}^{E_i}, M)M$$

Furthermore, if $not\, A$ belongs to $Def(\mathcal{P} \cup \mathcal{E}, M)$, there is no rule in $\mathcal{P}$ with head A and true body, then $not\, A \in Def(\mathcal{P}, M)$ and hence for any A, $not\, A \in Def(\mathcal{P} \cup \mathcal{E}, M)$ implies $not\, A \in Def(\mathcal{P}, M)$ i.e.

$$Def(\mathcal{P} \cup \mathcal{E}, M) \subseteq Def(\mathcal{P}, M)$$

It follows that

$$Def(\mathcal{P} \cup \mathcal{E}, M) \quad \subseteq \quad least(Generator^R(\mathcal{P}, M)) \tag{A.1}$$

Moreover, since

$$\rho(\mathcal{P}) - Rej^S(\mathcal{P}^{E_i}, M)M \subseteq \rho(\mathcal{P}) - Rej^R(\mathcal{P}, M) \tag{A.2}$$

by definition of $\mathcal{G}en$ and the inclusions A.1 and A.2 we obtain

$$\mathcal{G}en \subseteq Generator^R(\mathcal{P}, M)$$

We prove that $Generator^R(\mathcal{P}, M) = \mathcal{G}en \cup U \cup R$ where R and U are definite logic programs such that $\mathcal{G}en$, R and U satisfies the hypothesis of Property A.1.1. This, by such

Property, implies:

$$least(Generator^R(\mathcal{P},M)) = least(\mathcal{G}en) = M$$

and prove the thesis.

We define R in the following way

$$\{\tau \notin \mathcal{G}en \wedge H(\tau) \in M \wedge (\tau \in Def(\mathcal{P},M) \vee \tau \in \rho(\mathcal{P}) - Rej^R(M))\}$$

And U in the following way

$$\{\tau \notin \mathcal{G}en \wedge\ H(\tau) \notin\ M \wedge\ \tau \mid \tau \in \rho(\mathcal{P}) - Rej^R(M)\}$$

By definition of $Generator^R(\mathcal{P},M)$ it follows

$$Generator^R(\mathcal{P},M) = \mathcal{G}en \cup U \cup R$$

and, by definition of R it follows $H(R) \subseteq least(\mathcal{G}en)$. It remains to prove that $M \not\models B(\tau)$ for any $\tau \in U$. Let L be the head of τ. Since, by definition of U, $H(\tau) \notin M$, there are two possibilities.

- L is the positive literal A and $not\, A \in\ Def(\mathcal{P} \cup \mathcal{E},M)$. Then there does not exist any rule in $\rho(\mathcal{P})$ whose head is A and whose body is satisfied by M, then $M\ \not\models B(\tau)$.
- There exists a rule η in $\rho(\mathcal{P}) - Rej^S(\mathcal{P}^{E_i},M)$ such that $H(\eta) = not\, L$ and $M \models B(\eta)$. Let us suppose, by contradiction, $M \models B(\tau)$ and hence $M \models B(\tau)$. Let i be the index such that $\tau \in P_i$. Then there exists P_j such that $\eta\ \in\ P_j$ and $i < j$, otherwise η would not belong to $\rho(\mathcal{P}) - Rej^S(\mathcal{P}^{E_i},M)$. Then τ is rejected by η and hence: $\tau\ \notin\ \rho(\mathcal{P}) - Rej^R(M)$ against hypothesis.

We proved that $\mathcal{G}en$, U and R satisfy the hypothesis of Property A.1.1 then, by such Property: $least(Generator^R(\mathcal{P},M)) = least(\mathcal{G}en) = M$

$\diamondsuit$

Proof of Theorem 3.4.2 :

Let $\mathcal{P}$ be any DLP and $\mathcal{E}$ any DLP whose unique rules are tautologies and M a refined stable model of $\mathcal{P}$. We have to prove that M is also a refined stable model of $\mathcal{P} \cup \mathcal{E}$.

By definition, M is a refined stable model of $\mathcal{P} \cup \mathcal{E}$ iff

$$M = least(Generator^R(\mathcal{P} \cup \mathcal{E},M))$$

and

$$Generator^R(\mathcal{P} \cup \mathcal{E},M)\ \ =\ \ \rho(\mathcal{P}) - Rej^S(\mathcal{P}^{E_i},M)M\ \cup$$

$$Def(\mathcal{P}\cup\mathcal{E},M) \quad \cup \quad \rho(\mathcal{E}) - Rej^S(\mathcal{P}^{E_i},M)M$$

First, we find a simpler program G such that

$$least(Generator^R(\mathcal{P}\cup\mathcal{E},M)) = least(G)$$

Since

$$\rho(\mathcal{E}) - Rej^S(\mathcal{P}^{E_i},M)M$$

is a tautological program, by Property A.1.2 we obtain

$$\begin{aligned} &least(Generator^R(\mathcal{P}\cup\mathcal{E},M)) = \\ &least\left(\rho(\mathcal{P}) - Rej^S(\mathcal{P}^{E_i},M)M \cup Def(\mathcal{P}\cup\mathcal{E},M)\right) \end{aligned}$$

Moreover, we prove that $Def(\mathcal{P}\cup\mathcal{E},M) = Def(\mathcal{P},M)$. Clearly

$$Def(\mathcal{P}\cup\mathcal{E},M) \subseteq Def(\mathcal{P},M)$$

Viceversa, suppose $not\,A \in Def(\mathcal{P},M)$. Then $A \notin M$. For each rule τ in $\rho(\mathcal{E})$ such that $H(\tau) = A$, since τ is a tautology, $A \in B(\tau)$ which implies $M \not\models B(\tau)$. Hence, it follows that $A \in Def(\mathcal{P}\cup\mathcal{E},M)$. It follows that

$$Def(\mathcal{P}\cup\mathcal{E},M) = Def(\mathcal{P},M)$$

as stated. Hence we define the program G as

$$G = \rho(\mathcal{P}) - Rej^S(\mathcal{P}^{E_i},M)M \cup Def(\mathcal{P},M)$$

By what previously said it follows that:

$$least(Generator^R(\mathcal{P}\cup\mathcal{E},M)) = least(G)$$

Hence, if we prove $least(G) = M$ we prove the thesis. By

$$\rho(\mathcal{P}) - Rej^S(\mathcal{P}^{E_i},M)M \subseteq \rho(\mathcal{P}) - Rej^R(M)$$

It follows

$$G \subseteq \rho(\mathcal{P}) - Rej^R(M) \cup Def(\mathcal{P},M)$$

Then clearly $least(G) \subseteq M$.

It remains to prove the opposite inclusion. Let us suppose, by contradiction, there exists a literal L such that $L \in M$ but $L \notin least(G)$. This means there exists a rule $\tau \in Generator^R(\mathcal{P},M)$ such that $\tau \notin G$ and $H(\tau) = L$. This means there exists a tautology η such that $H(\eta) = not\,L$ and $M \models B(\eta)$. The last condition implies, since

η is a tautology, that $not\, L \in M$, or equivalently $L \notin M$ against hypothesis. Then $least(G) = M$ and hence M is a refined stable model of $\mathcal{P} \cup \mathcal{E}$. ♢

Proof of Theorem 3.5.1 :

Let $\mathcal{P}$ be any DLP and M a refined stable model of $\mathcal{P}$. We have to prove that M is a dynamic stable model. We remember here the definition of rejected rules and default assumptions for the the dynamic stable model semantics.

$$\begin{aligned} Rej(\mathcal{P},M) &= \{r \mid r \in P_i, \exists r' \in P_j, i < j, r \bowtie r', M \vdash B(r')\} \\ Def(\mathcal{P},M) &= \{not\, A \mid \nexists r \in \rho(\mathcal{P}), H(r) = A, M \vdash B(r)\} \end{aligned}$$

We know that M is a refined stable model of $\mathcal{P}$ i.e.

$$M = least(\rho(\mathcal{P}) - Rej^R(M) \cup Def(\mathcal{P},M))$$

We have to prove that

$$M = least(\rho(\mathcal{P}) - Rej(\mathcal{P},M) \cup Def(\mathcal{P},M))$$

Following the outline of the proof of Theorem 3.4.1 We prove that

$$Generator^R(\mathcal{P},M) \cup R \cup U = Generator(\mathcal{P},M)$$

where R and U are definite logic programs such that $Generator^R(\mathcal{P},M)$, R and U satisfies the hypothesis of Property A.1.1. This, by such Property, implies:

$$least(Generator^R(\mathcal{P},M)) = M$$

and prove the thesis. By the two definitions of rejected rules we obtain

$$\rho(\mathcal{P}) - Rej^R(M) \subseteq \rho(\mathcal{P}) - Rej(\mathcal{P},M)$$

We define R in the following way

$$\{\tau \mid \tau \notin \rho(\mathcal{P}) - Rej^R(M) \ \wedge\ \tau \in \rho(\mathcal{P}) - Rej(\mathcal{P},M) \wedge\ H(\tau) \in M\}$$

And U in the following way

$$\{\tau \mid \tau \in \rho(\mathcal{P}) - Rej(\mathcal{P},M) \ \wedge\ \tau \notin Generator^R(\mathcal{P},M) \wedge\ H(\tau) \notin M\}$$

By definition of $Generator^R(\mathcal{P},M)$ it follows

$$Generator^R(\mathcal{P},M) = Generator(\mathcal{P},M) \cup U \cup R$$

and, by definition of R it follows $H(R) \subseteq least(Generator(\mathcal{P},M))$.

It remains to prove that $M \not\models B(\tau)$ for any $\tau \in U$. Let L be the head of τ. By definition of U it follows that $L \notin M$. There are two possibilities.

- L is the positive literal A and $not\, A \in Def(\mathcal{P},M))$. Then it does not exists any rule in $\rho(\mathcal{P})$ whose head is A and whose body is satisfied by M, then $M \not\models B(\tau)$.

- There exists a rule $\eta \in \rho(\mathcal{P}) - Rej^R(M)$ such that $H(\eta) = not\, L$ and $M \models B(\eta)$. Let us suppose, by contradiction, $M \models B(\tau)$, which implies that $M \models B(\tau)$. Let i be the index such that $\tau \in P_i$. Then, there exists P_j such that $\eta \in P_j$ and $i < j$ otherwise η would not belong to $\rho(\mathcal{P}) - Rej^R(M)$. Then, by definition, $\tau \notin \rho(\mathcal{P}) - Rej(\mathcal{P},M)$, against hypothesis.

We proved that $Generator^R(\mathcal{P},M)$, U and R satisfies the hypothesis of Property A.1.1. Hence we obtain

$$least(Generator(\mathcal{P},M)) = least(Generator^R(\mathcal{P},M) \cup U \cup R)$$

By Property A.1.1: $least(Generator^R(\mathcal{P},M)) = least(Generator(\mathcal{P},M))$. This concludes the proof.

$\diamondsuit$

Proof of Proposition 3.7.2:

In the rest of the proof we denote with $\mathcal{P}Q$ the program $\mathcal{P} \oplus Q$ and with $\mathcal{P}Q'$ the program $\mathcal{P}' \oplus Q$.

From $B(\tau) \subseteq B(\gamma)$ it follows that, for any interpretation I:

$$I \models B(\gamma) \;\Rightarrow\; I \models B(\tau)$$

We distinguish the cases when $hd(\tau) = hd(\gamma)$ and $hd(\tau) \bowtie hd(\gamma)$.

If $hd(\tau) = hd(\gamma)$ we prove that, for any interpretation I:

$$\Gamma^R_{\mathcal{P}Q}(I) = \Gamma^R_{\mathcal{P}Q'}(I)$$

from which the thesis immediately follows.

From the implication above it follows by definition of $Default$

$$Default(\mathcal{P}Q,I) \;=\; Default(\mathcal{P}Q',I)$$

Moreover, since γ belongs to a less recent update than τ, any rule rejected by γ is also rejected by τ. Hence, if $\gamma \in Rej^R(\mathcal{P}Q,I)$ then:

$$\rho(\mathcal{PQ}) \setminus Rej^R(\mathcal{PQ},I) \quad = \quad \rho\left(\mathcal{PQ}'\right) \setminus Rej^R(\mathcal{PQ}',I)$$

and hence, by definition of Γ^R, it follows $\Gamma^R_{\mathcal{PQ}}(I) = \Gamma^R_{\mathcal{PQ}'}(I)$.

If

$$\gamma \notin Rej^R(\mathcal{PQ},I)$$

then:

$$\rho(\mathcal{PQ}) \setminus Rej^R(\mathcal{PQ},I) \quad = \quad \rho\left(\mathcal{PQ}'\right) \setminus Rej^R(\mathcal{PQ}',I) \cup \{\gamma\}$$

Since γ belongs to a less recent update than τ, also τ is not rejected i.e. it belongs to $\rho(\mathcal{PQ}') \setminus Rej^R(\mathcal{PQ}',I)$. If

$$\Gamma^R_{\mathcal{PQ}'}(I) \not\models B(\gamma)$$

then, by Property A.1.1

$$\Gamma^R_{\mathcal{PQ}}(I) = least(\rho(\mathcal{PQ}) \setminus Rej^R(\mathcal{PQ},I)) = least(\rho\left(\mathcal{PQ}'\right) \setminus Rej^R(\mathcal{PQ}',I) \cup \{\gamma\} = \Gamma^R_{\mathcal{PQ}'}(I)$$

as desired.

If

$$\Gamma^R_{\mathcal{PQ}'}(I) \models B(\gamma)$$

then

$$\Gamma^R_{\mathcal{PQ}'}(I) \models B(\tau)$$

hence, since $\Gamma^R_{\mathcal{PQ}'}(I)$ is a model of τ, then $hd(\gamma) \in \Gamma^R_{\mathcal{PQ}'}(I)$ and hence, by Property A.1.1

$$\Gamma^R_{\mathcal{PQ}}(I) = least(\rho(\mathcal{PQ}) \setminus Rej^R(\mathcal{PQ},I)) = least(\rho\left(\mathcal{PQ}'\right) \setminus Rej^R(\mathcal{PQ}',I) \cup \{\gamma\} = \Gamma^R_{\mathcal{PQ}'}(I)$$

as desired.

If $hd(\tau) \bowtie hd(\gamma)$ let I be a two valued interpretation. We will prove that I is a well supported mode of $\mathcal{PQ}$ iff it is also a well supported mode of $\mathcal{PQ}'$.

Let ℓ be any level mapping. By $B(\tau) \subseteq B(\gamma)$ we obtain:

$$\ell(B(\tau)) \leq \ell(B(\gamma))$$

If

$$I \not\models B(\gamma) \ \vee \ \ell(hd(\gamma)) \not> \ell(B(\gamma))$$

then γ is not relevant in order to determine weather I is a well supported model or not. Hence I is a well supported mode of $\mathcal{PQ}$ (and ℓ the relative level mapping)

iff it is also a well supported mode of $\mathcal{P}Q'$ (and ℓ the relative level mapping) as desired.

It remains to examine the case when

$$I \models B(\gamma) \ \wedge \ \ell(hd(\gamma)) > \ell(B(\gamma))$$

In this case we have

$$I \models B(\gamma) \ \Rightarrow \ I \models B(\tau)$$

and

$$\ell(hd(\tau)) = \ell(hd(\gamma)) > \ell(B(\gamma)) \geq \ell(B(\tau))$$

Hence, by Definition 28, $\gamma \in Rej_l(\mathcal{P}Q,I)$. Moreover, $hd(\tau) \in I$ iff $\tau \in Rej_l(\mathcal{P}Q,I)$ which is equivalent to $\tau \in Rej_l(\mathcal{P}Q',I)$ i.e. γ it is not relevant to determine weather I and ℓ satisfy condition *i*) and *ii*) of Definition 29 i.e. it is not relevant to determine weather I is well supported model of $\mathcal{P}Q$ and $\mathcal{P}Q'$. Hence I is a well supported model of $\mathcal{P}Q$ iff it is a well supported model of $\mathcal{P}Q'$ as desired.

$\diamondsuit$

Proof of Proposition 3.7.3:

In the rest of the proof we denote with $\mathcal{P}Q$ the program $\mathcal{P} \oplus Q$ and with $\mathcal{P}Q'$ the program $\mathcal{P}' \oplus Q$. We prove that, for any interpretation I:

$$\Gamma^R_{\mathcal{P}Q}(I) = \Gamma^R_{\mathcal{P}Q'}(I)$$

from which the thesis immediately follows. By definition of $Default$:

$$Default(\mathcal{P}Q,I) \quad = \quad Default(\mathcal{P}Q',I)$$

Since there exists no conflicting rule with in any P_j with $j \leq i$, no rule is rejected by τ.

If $not\, A \notin Default(\mathcal{P}Q',I)$ then there exists a rule with head A and true body in I in some P_j. Since any rule with head A is, by hypothesis, in a more recent update than i, then that rule also rejects τ. Hence

$$\rho(\mathcal{P}Q) \setminus Rej^R(\mathcal{P}Q,I) \quad = \quad \rho\left(\mathcal{P}Q'\right) \setminus Rej^R(\mathcal{P}Q',I)$$

and hence, by definition of Γ^R, it follows $\Gamma^R_{\mathcal{P}Q}(I) = \Gamma^R_{\mathcal{P}Q'}(I)$ as desired.

If $not\, A \in Default(\mathcal{P}Q',I)$ then, by Property A.1.1:

$$\Gamma^R_{\mathcal{P}Q}(I) = least(\rho(\mathcal{P}Q) \setminus Rej^R(\mathcal{P}Q,I)) = least(\rho\left(\mathcal{P}Q'\right) \setminus Rej^R(\mathcal{P}Q',I) \cup \{\gamma\} = \Gamma^R_{\mathcal{P}Q'}(I)$$

as desired.

♢

Lemma A.3.1 *Let $\mathcal{MP}$ be any multidimensional dynamic logic program in the language $\mathcal{L}$, n an index, M a well-supported model of $\mathcal{MP}$ at P_n and $L \in M$. Then, either $L \in Default(M)$ or there exists a rule $L \leftarrow B$ in some P_i such that the fact $safe(L,i)$ belongs to $Safe^R(\mathcal{MP}^n, M)$, $M \models B$ and $\ell(L) > ell(B)$.*

proof We proceed by cases.

- Let $L = not\, A$ for some atom A and there exists no rule $A \leftarrow B$ such that $M \models B$. Then, by definition, $M \in Default(M)$ and the thesis is satisfied.

- Let now $L = A$ and there exists no rule with head $not\, A$ whose body is true in M. Then, by *ii*) there exists a rule $\tau^i : A \leftarrow B$, in some P_i, such that $M \models B$ and $\ell(A) > \ell(B)$. Clearly $safe(L,i) \in Safe^R(L,i)$ since there is no rule that threatens τ^i and the thesis is satisfied.

- Finally, let us suppose there exists a rule with head $not\, L$ and true body in M. Let α be a maximum index st. such a rule η^α exists in P_α. Since $L \in M$ then M is not a model of η. This means $\eta^\alpha \in Rej_\ell(M)$ i.e. there exists a rule $\tau^\beta : L \leftarrow B$ in P_β : $\alpha \prec \beta$, such that $M \models B$ and $l(L) > l(B)$. There exists no rule with head $not\, L$ and true body in P_β or any successive update, otherwise the maximality of α would be violated, hence there exists no rule that threatens τ^β, which means $\tau^\beta \in Safe^R(\mathcal{MP}^n, M)$ as desired.

♢

We are now ready to prove the equivalence Theorem 3.9.1.

Proof of Theorem 3.9.1: In the following we will use the notation $GS(M)$ for $\Sigma(\mathcal{MP}^n) \cup Safe^R(\mathcal{MP}^n, I) \cup Rj(\mathcal{MP}^n) \cup Default(\mathcal{MP}^n, I)$.

Let us first suppose M is a RMSM of $\mathcal{P}$, i.e. $M = \Gamma^R(M) = least(GS(M))|_{\mathcal{L}}$. We have to prove that M is a well-supported model.

For this we have to prove conditions *i*) and *ii*) of Definition 32. We define the level mapping ℓ as follows: Let T_{GS} be the immediate consequence operator of the *definite* logic program $GS(M)$ and T^m_{GS} be the m^{th} application of T_{GS} to the empty set . If *not A* is in $Default(M)$ then $\ell(A) = 1$, otherwise $\ell(A)$ is the minimum n such that either A or *not A* is in T^m_{GS}. Formally:

$$\ell(A) = min(\{n : A \in T^m_{GS} \text{ or } not\ A \in T^m_{GS}\})$$

- Thirst of all we prove condition *ii*) We proceed by induction on the value of $\ell(L)$.

Basic step. Let A be any atom in M, then $\ell(A) > 1$, since any rule whose head is A contains the literal $safe(A,i)$ for some index i in its body. Hence the thesis is trivially satisfied.

Inductive step. Let $\ell(A) = m+1$ and suppose $ii)$ proved for any A st. $\ell(A) \leq m$. Since $A \in least(GS(M))$, there exists a rule $A \leftarrow B, safe(A,i)$ in $GS(M)$ such that $\tau^i : A \leftarrow B$ belongs to P_i, $B \subseteq T^m_{GS}$, $safe(A,i) \in T^m_{GS}$ and finally, $\ell(A) > \ell(B)$ and M satisfies B. Since $safe(A,i)$ is true in T^m_{GS}, then, either 1): τ^i is not threatened by any rule or 2): all the rules that threatens τ^i are rejected.

1. In the former case, by definition, there exists no rule in a later update that can reject τ^i, hence, τ^i is also a rule of $\rho(P) \setminus Rej_\ell(M)$ and condition $ii)$ is satisfied.
2. In the latter case, let η^j be a threat for τ^i that belongs to an update j. Then, there exists a rule τ^k in an update P_k with $j \prec k$ with head A and true body such that $A > safe(A,i) > \ell(B(\tau^k))$. Let us consider k to be a maximum index such that such a τ^k belongs to k. It remains to prove that $\tau^k \in \rho(P) \setminus Rej_\ell(M)$.

   By contradiction. Let us suppose, there exists $\eta^\beta \in P_\beta$ with $i \prec k \prec \beta$ that threatens τ^k. Then η would threaten τ^i as well, i.e. the atom $rej(A,\beta)$ is in the body of the rule $safe(A,i) \leftarrow cond^R(\mathcal{MP}^n, A, i)$. By the maximality condition imposed on k, there exists no rule in any subsequent update that rejects η^β, hence $rej(A,\beta)$ is not in $least(GS(M))$ and then $safe(A,i)$ does not belong to T^m_{GS} against what previously said.

   Hence τ^k is not threatened by any rule in a subsequent update. Then, by definition $\tau^k \in \rho(P) \setminus Rej_\ell(M)$ and condition $ii)$ is satisfied..

- We prove now condition $i)$ i.e. M is a model of $\rho(P) \setminus Rej_\ell(M)$. It is sufficient to prove that if τ^i is a rule in any P_i such that M is not a model of τ^i, then $\tau^i \in Rej_\ell(M)$.

  If $\tau^i \in P_i$ is such a rule and L is its head, then in $M \models B(\tau^i)$ and $L \notin M$. This means, since M is two valued, $not\ L \in M$. Then there exists a rule $not\, L \leftarrow B, safe(not\, L, j)$ such that $\eta^j : L \leftarrow B \in P_j$, $\ell(L) > \ell(B)$, $\ell(L) > safe(not\, L, j)$ and $\Gamma^R(M) \models B, safe(not\, L, j)$. If $i \prec j$ then , by definition, $\tau^i \in Rej_\ell(M)$ as desired. If $i \not\prec j$, then τ^i threatens eta^j, hence $rej(L,i) \in least(GS(M))$, which means there exits a rule η^k with head L in some update P_k with $i \prec k$ such that $M = \Gamma^R(M) \models B(\eta^k)$ and $\ell(L) > \ell(B(\eta^k))$. Then, by definition, $\tau^i \in Rej_\ell(M)$ as desired.

Let now M be a WS model of $\mathcal{MP}$. First of all we notice that $Default(M) \subseteq M$. Indeed if $not\ A \in Default(M)$ then it does not exist in $\rho(P)$ a rule $A \leftarrow B$ such that $M \models B$ hence, by definition A is not in M and then $not\ A \in M$. We prove that

$$M = \Gamma^R(M)$$

- We start proving $L \in M \Rightarrow L \in \Gamma^R(M)$. We proceed by induction on the level of L.

  Basic step. Let $\ell(L)$ be minimal. If $L \in Default(M)$, then, since $Defalut(M) \subseteq GS(M)$, this implies that $L \in least(GS(M))$. If $L \notin Default(M)$, then, by Lemma A.3.1, there exists a rule $\tau^i : L \leftarrow B$ with $l(L) > l(B)$ in some P_i. By minimality of $\ell(L)$ τ^i can only be the fact L. Hence the rule $L \leftarrow safe(L,i)$ is in $GS(M)$. We prove that $safe(L,i) \in least(GS(M))$ and hence $L \in \Gamma^R(M)$. If τ^i is not threatened by any rule, then the fact $safe(L,i) \leftarrow$ belongs to $GS(M)$. Otherwise, let $rej(not\, L, j)$ be in $cond^R(L,i,M)$. This means there exists a rule η^j in some update $P_j : i \not\prec j$ with head $not\, L$ and true body in M. Since $not\, L \notin M$, η^j belongs to $Rej_\ell(M)$. By definition of Rej_l and the minimality of $\ell(L)$, we deduce that the fact $L \leftarrow$ belongs to P_k with $j \prec k$. This implies that the fact $rej(not\, L, j) \leftarrow$ belongs to $GS(M)$, i.e. $rej(not\, L, j) \in least(GS(M))$. Repeating this reasoning for all the $rej(not\, L, j)$ in $cond^R(L,i,M)$ we obtain that $least(GS(M)) \models cond^R(L,i,M)$ and hence, since the rule $safe(L,i) \leftarrow cond^R(L,i,M)$ belongs to $GS(M)$ we obtain $safe(L,i) \in least(GS(M))$ and then $L \in \Gamma^R(M)$ as desired.

  Inductive step. Let us suppose we proved that $L \in M \Rightarrow L \in \Gamma^R(M)$ for all L such that $l(L) \leq m$, we prove it also for $l(L) = m+1$. If $L \in Default(M)$, then $L \in \Gamma^R(M)$. If $L \notin Default(M)$, then, by Lemma A.3.1, there exists a rule $\tau^i : L \leftarrow B$ with $l(L) > l(B)$ in some P_i. Hence, the rule $L \leftarrow B, safe(L,i)$ is in $GS(M)$. By inductive hypothesis we know $\Gamma^R(M) \models B$. It remains to prove that $least(GS(M)) \models safe(L,i)$. If τ^i is not threatened by any rule, then $safe(L,i)$ belongs to $GS(M)$. Otherwise, let $rej(not\, L, j)$ be in $cond^R(L,i,M)$. This means there exists a rule η^j in some update $P_j : i \not\prec j$ with head $not\, L$ and true body in M. Since $not\, L \notin M$, η^j belongs to $Rej_\ell(M)$. Hence a rule $\tau^k : L \leftarrow B'$ such that $M \models B'$ and $\ell(L) > \ell(B')$ belongs to some P_k with $j \prec k$. This means that the rule $rej(not\, L, j) \leftarrow B'$ belongs to $GS(M)$. Moreover, by inductive hypothesis and by what previously said, we conclude $B\prime \subseteq \Gamma^R(M)$ and hence $rej(not\, L, i)$ belongs to $least(GS(M))$. Repeating this reasoning for all the $rej(not\, L, j)$ in $cond^R(L,i,M)$ we obtain that $least(GS(M)) \models cond^R(L,i,M)$ and hence, since the rule $safe(L,i) \leftarrow cond^R(L,i,M)$ belongs to $GS(M)$ we obtain $safe(L,i) \in least(GS(M))$ and then $L \in \Gamma^R(M)$ as desired.

- We have now to prove that, for any $L \in \Gamma^R(M)$ it also holds $L \in M$. We proceed by induction proving that $T^m_{GS}|_{\mathcal{L}} \subseteq M$. Since $least(T^m_{GS}) = \bigcup T^m_{GS}$, we obtain then the desired result

  Basic step. The condition is trivially true since $T^0_{GS} = \emptyset$.

  Inductive step. Let us suppose $T^m_{GS}|\mathcal{L} \subseteq M$. We want to prove $T^{m+1}_{GS}|\mathcal{L} \subseteq M$.

  Since M is a WS model, then $L \in Default(M) \Rightarrow L \in M$.

  Let us suppose instead there exists a rule $L \leftarrow B, safe(L,i)$ in $GS(M)$, B is true in T^m_{GS} and $safe(L,i) \in T^m_{GS}$. Let i be a maximum index st. such a rule belongs

to $G(M)$. This means the rule $\tau^i : L \leftarrow B$ belongs to P_i. Moreover, by inductive hypothesis, it follows that $M \models B$. It remains to prove that τ^i does not belong to $Rej_\ell(M)$.

By contradiction. Let us suppose there exists a rule η^j with head $not\, L$ such that $M \models B(\eta^j)$ and $i \prec j$. Then η^j threatens τ^i, hence since $safe(L,i)$ belongs to T^m_{GS}, this implies $rej(not\, L, j) \in T^m_{GS}$. This is possible only if a rule τ^k with head L exists, in some update P_k with $i \prec j \prec k$ such that $T^m_{GS} \models B(\tau^k)$. By inductive hypothesis, $M \models B(\tau^k)$. This violates the maximality of i.

Then $\tau^i \notin Rej_\ell(M)$ and then M is a model of τ^i, which means, since its body is true, that $L \in M$ as desired.

♢

Proof of Lemma 3.10.1: We have to prove the two inclusions

$$a)\ \Gamma_{(\mathcal{MP},n)}(I) \subseteq \Gamma^D_{(\mathcal{MP},n)}(I) \qquad b)\ \Gamma^D_{(\mathcal{MP},n)}(I) \subseteq \Gamma_{(\mathcal{MP},n)}(I) \tag{A.3}$$

- First we prove inclusion a). This is inclusion is nearly straightforward. Let τ^i be a rule of P_i not in $Rej(Multi^n, I)$ This means there is no rule that threatens τ^i. Hence the fact $safe(L,i) \leftarrow$ belongs to $Safe(L,i,I)$. Since $\Gamma^D_{(\mathcal{MP},n)}(I)$ is a model of $Safe(L,i,I)$ and a model of any rule $L \leftarrow B, safe(L,i)$ such that $L \leftarrow B$ belongs to P_i, then $\Gamma^D_{(\mathcal{MP},n)}(I)$ is a model of any rule $L \leftarrow B$ that belongs to $\rho(P) \setminus Rej(I)$. Since $\Gamma^D_{(\mathcal{MP},n)}(I)$ is also, by definition, a model of $Default(I)$ we conclude that it is a model of.

  $$\rho(\mathcal{MP}^n) \cup Rej(\mathcal{MP}^n, M) \cup Default(\mathcal{MP}^n, I)$$

  Hence $\Gamma_{(\mathcal{MP},n)}(I)$, being the least model of such program, is a subset of $\Gamma^D_{(\mathcal{MP},n)}(I)$.

- We prove now inclusion b) Let $G(M)$ be the definite program.

  $$\Sigma(\mathcal{MP}^n) \cup Safe(\mathcal{MP}^n, I) \cup Rj(\mathcal{MP}^n)\ \cup Default(\mathcal{MP}^n, I)$$

  and T_G the immediate consequences operator associated to $G(M)$. Moreover, let T^m_G be the $m-th$ application of T_G to the empty set. Then $\Gamma^D_{(\mathcal{MP},n)}(I) = \bigcup T^m_G | \mathcal{L}$ We prove by induction that $T^m_G \subseteq \Gamma^D_{(\mathcal{MP},n)}(I)$ for each natural number m.

  Basic step. $T^0_G = \emptyset$, hence the inclusion is trivially true.

  Inductive step. Let us suppose $T^m_G \subseteq \Gamma^D_{(\mathcal{MP},n)}(I)$. We have to prove $T^{m+1}_G \subseteq \Gamma^D_{(\mathcal{MP},n)}(I)$. Let $L \in T^m_G$. Then, either $L \in Default(M)$, and hence $L \in \Gamma^D_{(\mathcal{MP},n)}(I)$, or there exists a rule $\tau^{i\prime} : L \leftarrow B, safe(L,i)$ in $\Sigma(\mathcal{MP}^n)$ such that: $\tau^i : L \leftarrow B \in P_i$,

$T_G^m \models B$ and $safe(L,i) \in T_G^m$. This implies, by inductive hypothesis, that $\Gamma^D_{(\mathcal{MP},n)}(I) \models B$. Let us consider a maximum i st. such a $\tau^{i'}$ belongs to P_i. If we prove that the fact $safe(L,i) \leftarrow$ belongs to $Safe(\mathcal{MP}^n, I)$, this means there exists no rule an a later update that threatens τ^i. This is equivalent to say that $\tau^i \in \rho(P) \setminus Rej(M)$. Then $\Gamma^D_{(\mathcal{MP},n)}(I)$ is a model of τ^i and, since $\Gamma^D_{(\mathcal{MP},n)}(I) \models B$, we conclude $L \in \Gamma^D_{(\mathcal{MP},n)}(I)$ It remains hence to prove that, indeed, $safe(L,i) \leftarrow$ belongs to $Safe(\mathcal{MP}^n, I)$.

By contradiction. Let us suppose $safe(L,i) \leftarrow$ does not belong to $Safe(\mathcal{MP}^n, I)$. This is equivalent to say that $cond(L,i) \neq \emptyset$. Let then $rej(not\, L, j)$ be an element of $cond(L,i)$. Then $i \prec j$. Since $safe(L,i) \in T_G^m$, there exists a rule with head $rej(not\, L, i)$ and true body in I that belongs to $Defeat(\mathcal{MP}^n, I)$. This implies there exists a rule $\tau^k : L \leftarrow B'$ in some P_k, $i \prec j \prec k$ such that $T_G^m \models B'$. This violates the maximality of i

This concludes the proof.

$\diamondsuit$

Proof of Theorem 3.11.1:

First of all we partially evaluate $\mathcal{P}^{Tk}$ (see Theorem A.2.1). Let $\mathcal{P}_U^{Tk}$ be the program obtained by substituting the rules of the form

$$rej(\overline{L}, i) \leftarrow rej(\overline{L}, j)$$

with the set of rules $rej(\overline{L}, i) \leftarrow \overline{B}$. for each rule τ in any P_j of the form $not\, L \leftarrow B$ such that $i \leq j \leq k$. Since the latter is a partial evaluation of the former (see Theorem A.2.1), $\mathcal{P}^{Tk}$ and $\mathcal{P}_U^{Tk}$ have the same stable models.

We start by proving some technical results on M^T.

Lemma A.3.2 *For any atom A of $\mathcal{L}$: if M^T is a stable model of $\mathcal{P}_U^{Tk}$, then:*

$$not\, A \in M^T \;\Leftrightarrow\; A^- \in M^T$$

proof Since M^T is a two valued interpretation:

$$not\, A \in M^T \Leftrightarrow\; A \notin M^T$$

hence we have simply to prove that:

$$A^- \in M^T \;\Leftrightarrow\; A \notin M^T$$

If $A^- \in M^T$ then, either $rejA_0^- \notin M^T$ or there exist a rewritten rule in $\mathcal{P}_U^{Tk}$ of the form $A^- \leftarrow \overline{B}, not\, rej(A^-, i)$ with true body in M^T.

In the first case this implies that there exists no rule of the form $A \leftarrow B$ in any P_i with $i \leq k$ such that $\overline{B}$ is true in M^T i.e. there exists no rewritten rule of the form $A \leftarrow \overline{B}, not\ rej(A^-,i)$ with $\overline{B}$ true in M^T and hence A does not belong to M^T since M^T is a supported model.

In the second case let $A^- \leftarrow \overline{B}, not\ rej(A^-,i)$ be the rewritten rule with head A and true body in M^T with the greatest i. Then, for any $j \leq i$, the clause $rej(A,j) \leftarrow \overline{B}$ belongs to $\mathcal{P}_U^{Tk}$ and hence $rej(A,j)$ belongs to M^T. This implies that the body of any rewritten rule $A \leftarrow \overline{B}, not\ rej(A^-,j)$ with $j \leq i$ is false. On the other side, since for any rewritten rule $A \leftarrow \overline{B}, not\ rej(A^-,j)$ with $j > i$, the rule $rej(A^-,i) \leftarrow \overline{B}$ belongs to $\mathcal{P}_U^{Tk}$ and the atom $rej(A^-,i)$ is false in M^T, it follows that $\overline{B}$ is false in M^T for such a rule. Hence, A is not supported and hence it is false in M^T.

On the other side, if $A^- \notin M^T$ then, by the constraint

$$u \leftarrow not\ u,\ not\ A,\ not\ A^-$$

it follows that $A \in M^T$ and hence we obtain the thesis. ♢

From the lemma above we immediately obtain the following corollary

Corollary A.3.1 *Let M^T be a stable model of $\mathcal{P}_U^{Tk}$. For any conjunction B of literals in $\mathcal{L}$:*

$$M^T \models B \ \Leftrightarrow\ M^T \models \overline{B}$$

Lemma A.3.3 *Let M^T be an interpretation over the language of $\mathcal{P}_1^{Tk}$ and M its restriction to the language $\mathcal{L}$ i.e. $M = M^T|_{\mathcal{L}}$ and let $M|_{\overline{\mathcal{L}}}$ be the set of atoms obtained by restricting M^T to the set of atoms $\overline{\mathcal{L}}$. Let $\mathbf{Rj}(\mathcal{P}^k)$ be the set of rejection rules of $\mathcal{P}_1^{Tk}$. Moreover, let us assume that M^T and M satisfies conditions 3.11.1. Then the following equivalences hold:*

$$\begin{aligned} M^T - \mathcal{P}_1^{Tk} &= \overline{\rho\left(\mathcal{P}^k\right) \setminus Rej^R\left(\mathcal{P}^k, M\right)} \cup \overline{Def\left(\mathcal{P}^k, M\right)} \cup \mathbf{Rj}(\mathcal{P}^k) \\ least(M^T - \mathcal{P}_1^{Tk})|_{\overline{\mathcal{L}}} &= least\left(\overline{\rho\left(\mathcal{P}^k\right) \setminus Rej^R\left(\mathcal{P}^k, M\right)} \cup \overline{Def\left(\mathcal{P}^k, M\right)}\right) \end{aligned} \tag{A.4}$$

proof

The program $M^T - \mathcal{P}_U^{Tk}$ is formed by the reduce of **default**, **rewritten** and **rejection** rules.

The **default** rules, have the form $A^- \leftarrow not\ rej(A^-,0)$ where A is an atom of $\mathcal{L}$. The second of conditions 3.11.1 imposes that: $rej(A^-,0) \in M^T$ iff $not\ A$ belongs to $Default(P^k,M)$. Hence the fact A^- belongs to the M^T-reduce of set of default rules iff $not\ A \in Default(P^k,M)$ i.e. this set of rules is equal to $\overline{Default(P^k,M)}$.

By definition, the set of **rewritten** rules is the set of rules of the form

$$\overline{L} \leftarrow \overline{B},\ not\ rej(\overline{L},i).$$

For each rule $L \leftarrow B$ in P_i with $i \leq k$. By the third of conditions 3.11.1, $rej(\overline{L}, i) \in M^T$ iff $L \leftarrow B$ belongs to $Rej^R(\mathcal{P}^k, M)$ and hence the rule $\overline{L} \leftarrow \overline{B}$ belongs to the M^T-reduce of the set of rewritten rules iff $L \leftarrow B$ belongs to the set $\rho\left(\mathcal{P}^k\right) \setminus Rej^R\left(\mathcal{P}^k, M\right)$ i.e. the M^T-reduce of the set of rewritten rules is equal to $\overline{\rho\left(\mathcal{P}^k\right) \setminus Rej^R\left(\mathcal{P}^k, M\right)}$.

Finally, the set of **rejection** rules $\mathbf{Rj}(\mathcal{P}^k)$ is a set of rules of the form $rej(\overline{L}, i) \leftarrow \overline{B}$.

The literals of the form $rej(\overline{L}, i)$ do not belong to $\overline{\mathcal{L}}$ nor, as we have seen, occurs in the rules of the M^T-reduce of rejection and default rules, hence this rules are not relevant for determining $M^T|_{\overline{\mathcal{L}}}$ and can be ignored for this purpose.

The body of the rules of are conjunction of atoms, and hence the M^T-reduce of $\mathbf{Rj}(\mathcal{P}^k)$ is $\mathbf{Rj}(\mathcal{P}^k)$ itself. Hence we obtain the equivalences A.4.

$\diamondsuit$

To complete the proof it remains to to prove that any refined model M is the restriction to $\mathcal{L}$ of a stable model M^T of $\mathcal{P}_1^{Tk}$ and viceversa.

Let M^T be a stable model of $\mathcal{P}_1^{Tk}$. In order to use Lemma A.3.3, we start by proving that M^T satisfies conditions 3.11.1. The first of such conditions is guaranteed by Lemma A.3.2. Let $M = M^T|_{\mathcal{L}}$ be the restriction to $\mathcal{L}$ of M^T.

By definition:

$$Default(\mathcal{P}^k, M) = \{not\ A : \nexists\, A \leftarrow B.\ \in \mathcal{P}^k : M \models B\}$$

By definition of $\mathcal{P}_U^{Tk}$, if $not\ A\ \in\ Default(\mathcal{P}^k, M)$ then there does not exist in $\mathcal{P}_1^{Tk}$ a rule $rej(A^-, 0) \leftarrow \overline{B}$ such that $M^T \models \overline{B}$. This implies, by Corollary A.3.1, that:

$$not\ A\ \in\ Default(\mathcal{P}^k, M) \Leftrightarrow rej(A^-, 0)\ \notin\ M^T$$

Hence the second of conditions 3.11.1 is satisfied.

By definition

$$Rej^R(\mathcal{P}^k, M) = \{\tau \in P_i : \exists\, \eta \in P_j,\ i \leq j \leq k :\ \tau \bowtie \eta\ \wedge\ M \models B(\eta)\}$$

Then, by definition of $\mathcal{P}_U^{Tk}$ and by Corollary A.3.1, $\tau\ \in Rej^R(\mathcal{P}^k, M) \wedge \tau\ \in\ P_i$ if and only if there exists in $\mathcal{P}^{Tk}$ a rule

$$rej(\overline{hd(\tau)}, i) \leftarrow \overline{B}.\quad such\ that\ M^T \models \overline{B}$$

Then:

$$\tau\ \in\ Rej^R(M)^k \wedge \tau\ \in\ P_i\ \Leftrightarrow\ rej(\overline{hd(\tau)}, i)\ \in\ M^T$$

Hence the third of conditions 3.11.1 is satisfied.

It remains to prove that if M^T is be a stable model of $\mathcal{P}_U^{Tk}$, then $M = M^T|_{\mathcal{L}}$ is a refined model of $\mathcal{P}^k$. By Definition 9 and equivalence A.4 we obtain the following equivalence.

$$M^T|_{\overline{\mathcal{L}}} = least\left(\overline{\rho\left(\mathcal{P}^k\right) \setminus Rej^R\left(\mathcal{P}^k, M\right)} \cup \overline{Def\left(\mathcal{P}^k, M\right)}\right) \tag{A.5}$$

By Definition 27, M is a refined model of $\mathcal{P}^k$ iff

$$M = least\left(\rho\left(\mathcal{P}^k\right) \setminus Rej^R\left(\mathcal{P}^k, M\right) \cup Def\left(\mathcal{P}^k, M\right)\right)$$

The second elements of the equivalences above, are equal previous a syntactical substitution of any literals $not\,A$ with a literal A^-. Hence, By Lemma A.3.2 we obtain that M is a refined model of $\mathcal{P}^k$.

Let M be a refined model of $\mathcal{P}$ at P_k. We have to prove that there exists an extension of M^T of M such that M^T is a stable model of $\mathcal{P}_U^{Tk}$.

We define M^T as the two valued interpretation obtained by adding to M exactly those literals needed for satisfying conditions 3.11.1 i.e.

- A literal A^- for each $not\,A \in M$.
- A literal $rej(A^-, 0)$ for each $not\,A \in Def(\mathcal{P}^k, M)$.
- A literal $rej(\overline{L}, i)$ for each $\tau \in P_i \wedge \tau \in Rej^R(\mathcal{P}^k, M)$.

We have to prove the following equivalence:

$$M^T = least(M^T - \mathcal{P}_1^{Tk})$$

By construction, M^T satisfies conditions 3.11.1. Hence, by Lemma A.3.3 and by Definition 27 we obtain:

$$\begin{aligned} M^T - \mathcal{P}_1^{Tk} &= \overline{\rho\left(\mathcal{P}^k\right) \setminus Rej^R\left(\mathcal{P}^k, M\right)} \cup \overline{Def\left(\mathcal{P}^k, M\right)} \cup \mathbf{Rj}(\mathcal{P}^k) \\ least(M^T - \mathcal{P}_1^{Tk})|_{\overline{\mathcal{L}}} &= least\left(\overline{\rho\left(\mathcal{P}^k\right) \setminus Rej^R\left(\mathcal{P}^k, M\right)} \cup \overline{Def\left(\mathcal{P}^k, M\right)}\right) \\ M &= least\left(\rho\left(\mathcal{P}^k\right) \setminus Rej^R\left(\mathcal{P}^k, M\right) \cup Def\left(\mathcal{P}^k, M\right)\right) \end{aligned}$$

By the equivalences above and the first of conditions 3.11.1 we obtain:

$$M^T_{\overline{\mathcal{L}}} = least\left(\overline{\rho\left(\mathcal{P}^k\right) \setminus Rej^R\left(\mathcal{P}^k, M\right)} \cup \overline{Def\left(\mathcal{P}^k, M\right)}\right)$$

Since the heads of the rules of $\mathbf{Rj}(\mathcal{P}^k)$ do not occurs in any of the bodies of the rules in $M^T - \mathcal{P}_1^{Tk}$, and given the equivalence above, it is sufficient to iterate once the immediate-consequence operator of $T_{\mathbf{Rj}(\mathcal{P}^k)}$ over $M^T_{\overline{\mathcal{L}}}$ and to have the atoms of $M^T_{\overline{\mathcal{L}}}$ itself to obtain the least model of $M^T - \mathcal{P}_1^{Tk}$ i.e.

$$least(M^T - \mathcal{P}_1^{Tk}) = M^T_{\overline{\mathcal{L}}} \cup T_{\mathbf{Rj}(\mathcal{P}^k)}(M^T_{\overline{\mathcal{L}}}) \tag{A.6}$$

By construction, an atom of the form $rej(A^-,0)$ belongs to M^T iff $not\,A \notin Def(\mathcal{P}^k,M)$ iff there exists a rule $A \leftarrow B$ in some P_i with $i \leq k$ such that: $M \models B$ which is equivalent to say that there exists a rule $rej(A^-,0) \leftarrow \overline{B}$ in $\mathbf{Rj}(\mathcal{P}^k)$ and $M^T \models \overline{B}$ which is equivalent to say that $rej(A^-,0)$ belong to $T_{\mathbf{Rj}(\mathcal{P}^k)}(M^T_{\overline{\mathcal{L}}})$.

By construction, a literal of the form $rej(\overline{L},i)$ with $1 \leq i \leq k$ belongs to M^T iff there exists a rule τ with head L in some P_i with $i \leq k$ and a rule $not\,L \leftarrow B$ in some P_j with $i \leq k$ such that: $M \models B$ which is equivalent to say that there exists a rule $rej(\overline{L},i) \leftarrow \overline{B}$ in $\mathbf{Rj}(\mathcal{P}^k)$ and $M^T \models \overline{B}$ which is equivalent to say that $rej(A^-,0)$ belongs to $T_{\mathbf{Rj}(\mathcal{P}^k)}(M^T_{\overline{\mathcal{L}}})$. Hence the atoms of the form $rej(\overline{L},i)$ in M^T are exactly those of $T_{\mathbf{Rj}(\mathcal{P}^k)}(M^T_{\overline{\mathcal{L}}})$. Hence, by equivalence A.6 we conclude:

$$least(M^T - \mathcal{P}_1^{Tk}) = M^T$$

$\diamondsuit$

Proof of Theorem 3.12.1: For simplicity we will use the notation P^T for the transformed equivalent. We have to prove that a refined stable model of $\mathcal{MP}$ at P_n is a the restriction to $\mathcal{L}$ of a stable model of P^T and, on the other side that, for any stable model M^* of P^T is the there exists a refined stable model of M such that $M^*M \equiv |_{\mathcal{L}}M$.

Let then M be a refined stable model of $\mathcal{MP}$ at P_i. Let M^- be the set $\{A^- : A \notin M\}$. We define M^* in the following way. Let

$$\begin{aligned}
M^+ &= \{A \in \mathcal{L} \cap M\} \\
M^- &= \{A^- : A \notin M\} \\
M^{Inf} &= \{inf(L,i) :\ L \leftarrow B \in \mathcal{MP}^n,\ M \models B\} \\
M^{OK} &= \{ok(L,i) :\ inf(not\,L,i) \notin M^{Inf}\} \cup \{ok(L,i) :\ \exists inf(L,j) \in M^{Inf},\ i \prec j\} \\
M^{Safe} &= \{safe(L,i) :\ \forall j : i \not\prec j \preceq n\ ok(L,i) \in M^{OK}\} \\
M^{**} &= M^+ \cup M^- \cup M^{Inf} \cup M^{OK} \cup M^{Safe}
\end{aligned}$$

We define M^* as least interpretation in $\mathcal{L}^T$ containing M^{**}

We have to prove that M^* is a well-supported model of P^T according to Definition 11. First of all we prove that M^* is a model of P^T, which is equivalent to say that M^* is a model of each set of rules $DefR(\mathcal{MP}^n)$, $Inf(\mathcal{MP}^n)$, $Prog(\mathcal{MP}^n)$, $OK(\mathcal{MP}^n)$ and $Cons(\mathcal{MP}^n)$.

- We prove M^* is a model of $DefR(\mathcal{MP}^n)$. Let

$$A^- \leftarrow not\ \ inf(A,i_1),\ldots,not\, inf(A,i_n).$$

be a rule in$DefR(\mathcal{MP}^n)$ and suppose M^* satisfies $not\ \ inf(A,i_1),\ldots,not\, inf(A,i_n)$, then by definition of M^*, $inf(A,i_1)$ does not belong to M^{Inf} and hence, by definition of M^{Inf}, there exists no rule $A \leftarrow B$ in any P_i, $i \preceq n$ such that $M \models B$. This implies, by definition that $not\, A \in Defaut(\mathcal{MP}^n, M)$ and hence $not\, A \in M$ which finally implies $A^- \in M^- \subseteq M^*$. Hence M^* is a model of any rule

$$A^- \leftarrow not\ \ inf(A,i_1),\ldots,not\, inf(A,i_n).$$

in $DefR(\mathcal{MP}^n)$.

- We prove M^* is a model of $Inf(\mathcal{MP}^n)$. Let $Inf(\mathcal{MP}^n) \leftarrow \overline{B}$ be any rule of $Inf(\mathcal{MP}^n)$ and let us suppose M^* satisfies $\overline{B}$. Then, by definition of M^+ and M^-, we know M satisfies B and hence, by definition of M^{Inf}, $inf(L,i) \in M^*$. Hence M^* is a model of any rule

$$Inf(\mathcal{MP}^n) \leftarrow \overline{B}$$

of $Inf(\mathcal{MP}^n)$.

- We prove M^* is a model of $OK(\mathcal{MP}^n)$. Let

$$ok(L,i) \leftarrow \overline{B}$$

be any rule in $OK(\mathcal{MP}^n)$ such that $L \leftarrow B$ is a rule in any P_j with $i \not\prec j$ and $j \preceq n$. Let us suppose that M^* satisfies $\overline{B}$. Then $M \models B$ and hence, by definition, $ok(L,i) \in M^{OK} \subseteq M^*$. Let

$$ok(L,i) \leftarrow not\, inf(not\, L,i)$$

be any rule in $OK(\mathcal{MP}^n)$. Let us suppose $M* \models not\, inf(not\, L,i)$, then $inf(not\, L,i) \notin M^{Inf}$, and hence by definition of M^{OK}, $ok(L,i) \in M^{OK} \subseteq M^*$. Let finally

$$safe(L,i) \leftarrow ok(L,j_1),\ldots ok(L,j_m)$$

be a rule in $OK(\mathcal{MP})$. Let us suppose $M^* \models ok(L,j_1),\ldots ok(L,j_m)$, then, by definition of M^{Safe}, $safe(L,i) \in M^{Safe} \subseteq M^*$. Hence M^* is a model of any rule in $OK(\mathcal{MP}^n)$.

Then we have to prove condition *ii*) of Definition 11. Let $\mathcal{L}^\Gamma$ be the language of the definite program

$$S = \Sigma(\mathcal{MP}^n) \cup Safe^R(\mathcal{MP}^n, I) \cup Rj(\mathcal{MP}^n) \cup Default(\mathcal{MP}^n, I)$$

and let M^Γ be the least Herbrand model of such program. By Definition 37 we know $M = M^\Gamma|_\mathcal{L}$. By Theorem 3.9.1 we know there exists a level mapping ℓ^Γ such that for any atom A in M^Γ, there exists a rule $A \leftarrow B$ in S such that $S \models A$ and $\ell^\Gamma(A) > \ell^\Gamma(B)$. We define the level mapping ℓ over $\mathcal{L}^T$ in the following way

- for each atom $A \in \mathcal{L}$, $\ell(A) = \ell^\Gamma(A) + 1$
- for each atom A^- such that $A \in \mathcal{L}$, $\ell(A^-) = \ell^\Gamma(not\, A) + 1$.
- for each atom $safe(L,i)$ in $\mathcal{L}^T$, $\ell(safe(L,i)) = \ell^\Gamma(safe(L,i)) + 1$
- for each atom $ok(L,i)$ in $\mathcal{L}^T$ such that $rej(not\, L,i) \in least(S)$, $\ell(ok(L,i)) = \ell^\Gamma(rej(not\, L,i)) + 1$, otherwise $\ell(ok(L,i)) = 0$.
- for each atom $inf(L,i)$ such that $inf(L,i) \in M^*$ select a rule[1] $inf(L,i) \leftarrow B$ in P^T such that $M^* \models B$. Then $\ell(inf(L,i)) = \ell(B) + 1$ Otherwise $\ell(inf(not\, L,i)) = 0$

In order to prove condition *ii*) of Definition 11 we have to prove that, for any atom A in M^*, there exists a rule

$$A \leftarrow A_1 \ldots A_k, not\, B_1, \ldots not\, B_h$$

in P^T such that

$$M^* \models A_1 \ldots A_k, not\, B_1, \ldots not\, B_h$$

and $\ell(A) > \ell(A_i)$ for all As. We proceed by cases

- for each atom $inf(L,i)$ such that $inf(L,i) \in M^*$ there exists, by definition of ℓ a rule $inf(L,i) \leftarrow B$ in P^T such that $M^* \models B$ and $\ell(inf(L,i)) = \ell(B) + 1 > \ell(B)$.
- Let $ok(L,i)$ be an atom in M^*, by definition of M^*, either there exists a rule $ok(L,i) \leftarrow not\, inf(L,i)$ such that $inf(L,i) \notin M^{Inf}$, and hence $M^* \models not\, inf(L,i)$, and hence condition *ii*) is satisfied or there exists a node j, $i \prec j \preceq n$ such that $Inf(not\, L,i)$ belongs to M^{Inf}. This implies there exists a rule $rej(L,i) \leftarrow B$ in $Rj(\mathcal{MP}^n)$ such that $M \models B$, hence $rej(L,i) \in least(S)$. This implies there exists a rule $rej(L,i) \leftarrow B'$ in $Rj(\mathcal{MP}^n)$ such that $M \models B'$ and $\ell^\Gamma(rej(L,i)) > \ell^\Gamma(B')$, which implies the a rule $ok(L,i) \leftarrow \overline{B'}$ belongs to P^T and $\ell(ok(L,i)) > \ell(\overline{B'})$. Hence condition *ii*) is satisfied.
- Let $safe(L,i)$ be an atom in M^*, by definition of M^*, $safe(L,i) \in M^{Safe}$, that is to say that for all $j:\ i \not\prec j \preceq n$, the atom $ok(L,i)$ belongs to M^{OK} } Moreover, by definition of P^T, we know the rule

$$\tau:\ safe(L,i) \leftarrow ok(L,j_1), \ldots ok(L,j_m)$$

[1] Such a rule always exists by definition of M^{Inf}.

for each j_s such that $i \not\prec j_s$ and $j_s \preceq n$ belongs to P^T, and, by what previously said, $M \models ok(L,j_1),\dots ok(L,j_m)$. Hence τ is a rule with head $safe(L,i)$ and true body in M^*. It remains to prove $\ell(safe(L,i)) > \ell(B(\tau))$. Let $ok(L,j)$ be an atom in $B(\tau)$ such that $rej(L,i) \notin cond^R(\mathcal{MP}^n,M,L,i)$. Then, by definition of M^{Inf}, $inf(L,i) \notin M^{Inf}$ and hence there $\ell(ok(L,i)) = 0$, Since, by definition of ℓ, $\ell(safe(L,i)) \geq 1$, it remains to prove $\ell(safe(L,i)) > \ell(ok(L,j))$ for all (L,j) such that $rej(not\, L,j) cond^R(\mathcal{MP}^n,M,L,i))$.

We know the rule $safe(L,i) \leftarrow cond^R(\mathcal{MP}^n,M,L,i)$ belongs to $Safe^R(\mathcal{MP}^n,M)$. We prove now that $least(S)$ satisfies the body of such rule and hence $safe(L,i) \in least(S)$. Indeed, let us suppose $rej(not\, L,j) \in cond^R(\mathcal{MP}^n,M,L,i)$, this implies there exists a rule $not\, L \leftarrow B$ in P_j such that $M \models B$, hence $inf(not\, L,j) \in M^{Inf}$, which means, since $ok(L,j) \in M^*$ there exists a rule $ok(L,i) \leftarrow \overline{B'}$ such that $M \models \overline{B'}$ which implies there exists a rule $L \leftarrow B'$ in some P_k, $j \prec k \preceq n$ such that $M \models B'$ and hence the rule $rej(not\, L,j) \leftarrow B\prime$ belongs to S and $S \models B\prime$. Since $least(S)$ is a model of S this implies $rej(L,i) \in S$. This is true for any $rej(not\, L,j)$ in $cond^R(\mathcal{MP}^n,M,L,i)$, hence $least(S) \models cond^R(\mathcal{MP}^n,M,L,i)$, which means $safe(L,i) \in least(S)$. Finally this implies, by definition of ℓ^Γ that $\ell^\Gamma(safe(L,i)) > \ell^\Gamma(cond^R(\mathcal{MP}^n,M,L,i))$ and this implies, by definition of ℓ, that $\ell(safe(L,i)) > \ell(ok(L,j))$ for all (L,j) such that $rej(not\, L,j) cond^R(\mathcal{MP}^n,M,L,i))$. Hence $\ell(safe(L,i)) > \ell(B(\tau))$ as desired.

- let A be a atom in $\mathcal{L} \cap M^*$, then there exists a rule $A \leftarrow B, safe(L,i)$ in $\Sigma(\mathcal{MP}^n)$ such that $M \models B$, $least(S) \models safe(L,i)$, $\ell^\Gamma(A) > \ell^\Gamma(B)$ and $\ell^\Gamma(A) > \ell^\Gamma(safe(L,i))$. Then the rule $A \leftarrow \overline{B}, safe(L,i)$ belongs to P^T, $M^* \models \overline{B}, safe(L,i)$ and, by definition of ℓ, $\ell(A) > \ell(\overline{B})$ and $\ell(A) > \ell(safe(L,i))$ as required.

- Let finally A^- be a atom in $\mathcal{L} \cap M^*$. If $not\, A \in Default(\mathcal{MP}^n,M)$, then M^* satisfies the body of the rule

$$A^- \leftarrow not\ \ inf(A,i_1),\dots,not\, inf(A,i_n).$$

and hence condition *ii*) is satisfied. Otherwise there is a rule $not\, A \leftarrow B, safe(L,i)$ in $\Sigma(\mathcal{MP}^n)$ such that $M \models B$, $least(S) \models safe(L,i)$, $\ell^\Gamma(A) > \ell^\Gamma(B)$ and $\ell^\Gamma(A) > \ell^\Gamma(safe(L,i))$. Then the rule $A^- \leftarrow \overline{B}, safe(L,i)$ belongs to P^T, $M^* \models \overline{B}, safe(L,i)$ and, by definition of ℓ, $\ell(A^-) > \ell(\overline{B})$ and $\ell(A^-) > \ell(safe(L,i))$ as required.

Hence condition *ii*) of Definition 11 is satisfied and hence M^* is a well-supported model of P^T.

Let now M^* be a well-supported model of P^T, we have to prove that $M = M^*|_{\mathcal{L}} \Gamma^R_{(\mathcal{MP},n)}(M)$ Again, let

$$S = \Sigma(\mathcal{MP}^n) \cup Safe^R(\mathcal{MP}^n,I) \cup Rj(\mathcal{MP}^n) \cup Default(\mathcal{MP}^n,I)$$

We have to prove that M is the restriction to $\mathcal{L}$ of $least(S)$. Let

$$\begin{aligned} M^{Rj} &= \{rej(L,i) | ok(L,i) \leftarrow \overline{B} \in P^T \ \wedge M^* \models \overline{B}\} \\ M^{\Gamma} &= M \cup M^{Rj} \end{aligned} \tag{A.7}$$

We prove that $least(S) = M^{\Gamma}$ and hence $M = \Gamma^R_{(\mathcal{MP},n)}(M)$.

According to Theorem 3.9.1, we have to prove that M^{Γ} satisfies every rule in S and there exists a level mapping ℓ such that for every atom A in M, there exists a rule $A \leftarrow B$ st. $M^{\Gamma} \models B$ and $\ell(A) > \ell(B)$.

First of all we prove that M^{Γ} satisfies every rule in S. We proceed by cases.

- Let $not\, A$ be an atom in $Default(\mathcal{MP}^n, M)$. Then there exists no rule $A \leftarrow B$ in any P_i, $i \preceq n$ such that $M \models B$ and hence $inf(A,i)$ does not belong to M^* for any i. Since M^* is a model of the rule

  $$A^- \leftarrow not\ \ inf(A,i_1), \ldots, not\, inf(A,i_n).$$

  where $not\, inf(A,i_1), \ldots, not\, inf(A,i_n)$ is the conjunction of all $not\, inf(A,i_s)$ such that $i_s \leq n$. then A^- belongs to M, hence by the rule $\leftarrow A, A^-$ it follows that A does not belong to M and hence $not\, A \in M \subseteq M^{\Gamma}$ as required

- Let $rej(not\, ,L) \leftarrow B$ be a rule in S. The rule $rej(L,i) | ok(L,i) \leftarrow \overline{B}$ belongs to P^T. If $M \models B$, then, by definition of M^{Γ}, $rej(L,i) \in M^{\Gamma}$ as required.

- Let $L \in \mathcal{L}$ and $L \leftarrow B, safe(L,i)$ be as rule in S. Then the rule $L \leftarrow \overline{B}, safe(L,i)$ belongs to P^T. If $M^{\Gamma} \models B, safe(L,i)$, then $M^* \models \overline{B}, safe(L,i)$ and, since M^* is a model of P^T, we obtain that $L \in M^*$ and hence $L \in M^{\Gamma}$ as required.

- Finally, let $safe(L,i) \leftarrow cond^R(\mathcal{MP}^n, M, L, i)$ be a rule in S and let M^{Γ} satisfies $cond^R(\mathcal{MP}^n, M, L, i)$, we have to prove that $safe(L,i)$ belongs to M^{Γ}. in order o prove this, it is sufficient to prove that $ok(L,j)$ belongs to M^* for all j such that $i \not\prec j \preceq n$. Indeed, since the rule

  $$safe(L,i) \leftarrow ok(L,j_1), \ldots ok(L,j_m)$$

  belongs to P^T and M^* is a model of P^T, it would follow that $safe(L,i)$ belongs to M^* and hence to M^{Γ}.

  By the definition of M^{Γ} is follows that $ok(L,j)$ belongs to M^* for each $rej(not\, L, j)$ in M^{Γ}. It remains to prove that $ok(L,j)$ in case $i \not/prec j$ and $rej(not\, L, j) not \in cond^R(\mathcal{MP}^n, M, L, i)$. If this is the case, then there exists no rule in P_j with head $not\, L$ and true body in M and hence there exists no rule in P^T with head $inf(not\, L, j)$ and true body in M^*. Since M^* is well-supported, this means $inf(not\, L, j)$ is not in M, i.e. $M^* \models not\, inf(not\, L, j)$ and hence, since the rule

$ok(L,j) \leftarrow not\, inf(not\, ,j)$ is in P^T it follows that $ok(L,j)$ belongs to M^*. By what previously said, it follows that $safe(L,i)$ belongs to M^Γ as required.

It remains to prove condition *ii*) of Theorem 3.9.1, i.e. there exists a level mapping ℓ such that, for any atom A in M^Γ, there exists a rule $A \leftarrow B$ in S such that $M^\Gamma models B$ and $\ell(A) > \ell(B)$. Since $M*$ is a well-supported model of P^T, there exists a level mapping ℓ^* such that, for any atom A in M^* there exists a rule $A \leftarrow B$ in P^T such that $M^* \models B$ and $\ell^*(A) > \ell*(B)$. We define ℓ in the following way.

- For any atom $rej(L,i)$, $\ell(rej(L,i)) = \ell^*(ok(L,i))$.
- For any atom $A \in \mathcal{L}$, $\ell(not\, A) = \ell^*(A^-)$
- Otherwise $\ell(A) = ell*(A)$.

We prove, by cases, that ℓ satisfies the condition *ii*) above.

- Let the atom $rej(L,i)$ belong to M^Γ. Then $ok(not\, L,i)$ belongs to M^*, which implies there is a rule $ok(not\, L,i) \leftarrow \overline{B}$ in P^T with $M^* \models \overline{B}$ and $\ell*(ok(not\, L,i)) > \ell^*(\overline{B})$. This implies that the rule $rej(L,i) \leftarrow B$ belongs to S. By $M^* \models B$ it follows $M^\Gamma \models B$ Finally, since $\ell(rej(L,i)) = \ell^*(ok(L,i))$ and $\ell(B) = \ell^*(\overline{B})$, we obtain $\ell(rej(L,i)) > \ell(B)$ as required.
- Let the atom A such that A is a literal over $\mathcal{L}$ belong to M^Γ. Then there exists a rule $A \leftarrow \overline{B}, safe(A,i)$ in P^T such that $M^* \models \overline{B}, safe(A,i)$, and $\ell^*(A) > \ell^*(\overline{B}, safe(A,i))$ Then the rule $A \leftarrow B, safe(A,i)$ belongs to S, $M^\Gamma \models B, safe(A,i)$, and, since $\ell(safe(A,i)) = \ell^*(safe(A,i))$, $\ell(A) = \ell^*(A))$ and $\ell(B) = \ell^*(\overline{B})$, we conclude that $\ell(A) > \ell(B, safe(A,i))$ as required.
- Let the atom $not\, A$, such that $A \in \mathcal{L}$ belong to $Default(M)$. Then the fact $not\, A \leftarrow$ belongs to S and hence condition *ii*) is clearly satisfied.
- Let the atom $not\, A$ such that $not\, A$ is a literal over $\mathcal{L}$ belong to M^Γ. Hence $A^- \in M^*$. Let us suppose further $not\, A \notin Default(M)$ (the other case had been treated above). Then, for some i such that $i \prec n$, there exists a rule $inf(L,i) \leftarrow \overline{B}$ in P^T such that $M^* \models \overline{B}$ and hence $inf(A,i)$ belongs to M^*. Then the rule

$$A^- \leftarrow not\ \ inf(A,i_1), \ldots, not\, inf(A,i_n).$$

has a false body. Then, since $A^- \in M^*$, there exists a rule $\overline{A} \leftarrow \overline{B}, safe(not\, A,i)$ in P^T such that $M^* \models \overline{B}, safe(not\, A,i)$, and $\ell^*(\overline{A}) > \ell^*(\overline{B}, safe(not\, A,i))$ Then the rule $not\, A \leftarrow B, safe(not\, A,i)$ belongs to S, $M^\Gamma \models B, safe(not\, A,i)$, and, since $\ell(safe(A,i)) = \ell^*(safe(A,i))$, $\ell(not\, A) = \ell^*(\overline{A}))$ and $\ell(B) = \ell^*(\overline{B})$, we conclude that $\ell(not\, A) > \ell(B, safe(not\, A,i))$ as required.

- Finally, let the atom $safe(L,i)$ belong to M^Γ. Then $safe(L,i)$ also belongs to M^* and hence the rule

$$safe(L,i) \leftarrow ok(L,j_1),\ldots ok(L,j_m)$$

in P^T has a true body in M^* and $\ell^*(safe(L,i)) > \ell^*(ok(L,j_1),\ldots ok(L,j_m))$.

The rule $safe(L,i) \leftarrow cond^R(\mathcal{MP}^n,M,L,i)$ belongs to S. Since, for any L and i, $\ell(rej(not\,L,i)) = \ell^*(ok(L,j))$ and $\ell(safe(L,i)) = \ell^*(safe(L,i))$ we conclude

$$\ell(safe(L,i)) > \ell(cond^R(\mathcal{MP}^n,M,L,i))$$

It remains to prove that $M^\Gamma \models cond^R(\mathcal{MP}^n,M,L,i)$. If the atom $rej(not\,L,j)$ belongs to $cond^R(\mathcal{MP}^n,M,L,i)$, then there exists a rule $not\,L \leftarrow B$ in P_j such that $M \models B$, hence the rule $inf(L,i) \leftarrow \overline{B}$ belongs to P^T and $M^* \models \overline{B}$. This implies $inf(not\,L,j) \in M^*$ and hence the body of the rule $ok(L,j) \leftarrow not\,inf(not\,L,j)$ in P^T is not satisfied. Hence there has to exist a rule $ok(L,j) \leftarrow \overline{B'}$ in P^T such that $M^* \models \overline{B'}$, which implies, by definition of M^Γ, that $rej(not\,L,j) \in M^\Gamma$. Hence $M^\Gamma \models cond^R(\mathcal{MP}^n,M,L,i)$ as required.

Since we have proved that condition *ii*) of Theorem 3.9.1 is satisfied we have proved that M^Γ is the least Herbrand model of S and hence M is a refined model of $\mathcal{MP}$ at P_n. $\diamondsuit$

Appendix B

Proofs of theorems from Chapter 4

Proof of Lemma 4.3.1:

Let $\mathcal{P}$ be any Dynamic Logic Program and X, Y be two sets of literals, and $X \subseteq Y$, then $B(\tau) \subseteq X$ implies $B(\tau) \subseteq Y$ for each τ in P; then it follows immediately:

$$\begin{aligned}
Default(Y) &= \{not\ A\ such\ that\ \nexists \tau \in P : A = hd(\tau) \wedge B(\tau) \subseteq Y\} \subseteq \\
Default(X) &= \{not\ A\ such\ that\ \nexists \tau \in P : A = hd(\tau) \wedge B(\tau) \subseteq X\} \\
Rej(X) &= \{\tau \in P_i\ such\ that\ \nexists \eta \in P_j\ i < j \wedge \tau \bowtie \eta \wedge B(\tau) \subseteq X\} \subseteq \\
Rej(Y) &= \{\tau \in P_i\ such\ that\ \nexists \eta \in P_j\ i < j \wedge \tau \bowtie \eta \wedge B(\tau) \subseteq Y\} \\
Rej^R(X) &= \{\tau \in P_i\ such\ that\ \nexists \eta \in P_j\ i \leq j \wedge \tau \bowtie \eta \wedge B(\tau) \subseteq X\} \subseteq \\
Rej^R(Y) &= \{\tau \in P_i\ such\ that\ \nexists \eta \in P_j\ i \leq j \wedge \tau \bowtie \eta \wedge B(\tau) \subseteq Y\}
\end{aligned}$$

Hence it follows immediately:

$$\begin{aligned}
Generator(Y) &= \rho(\mathcal{P}) \setminus Rej(Y) \cup Default(Y) \subseteq \\
Generator(X) &= \rho(\mathcal{P}) \setminus Rej(X) \cup Default(X)
\end{aligned}$$

and:

$$\begin{aligned}
Generator^R(Y) &= \rho(\mathcal{P}) \setminus Rej^R(Y) \cup Default(Y) \subseteq \\
Generator(X) &= \rho(\mathcal{P}) \setminus Rej^R(X) \cup Default(X)
\end{aligned}$$

Hence, in the end:

$$\Gamma(Y) = least(Generator(Y)) \subseteq \Gamma(X) = least(Generator(X))$$

and:

$$\Gamma^R(Y) = least(Generator^R(Y)) \subseteq \Gamma^R(X) = least(Generator^R(X))$$

♢

Proof of Lemma 4.3.2: Let $\mathcal{P}$ be any Dynamic Logic Program. From Lemma 4.3.1 we know that:

$\Gamma^R(Y) \subseteq \Gamma^R(X)$

By definition we obtain

$$Generator^R(X) \subseteq Generator(X)$$

then it follows immediately

$$\Gamma^R(Y) \subseteq \Gamma^R(X) \subseteq \Gamma(X)$$

♢

Proof of Theorem 4.5.1 :

First we prove that W is an update model.

Let τ be any rule in any given update P_i and suppose there does not exist a rule η such that $\eta \in P_j,\ i < j$ and $B(\eta)$ is not false in W i.e., by Corollary 4.5.1 $B(\eta)$ is not satisfied by $\Gamma^R(W)$. This implies $\tau \notin Rej(\Gamma^R(W))$, hence W satisfies τ since it is model of a program that contains τ .

Let us suppose that the positive literal A belongs to W. Since

$$W = least(\rho(\mathcal{P}) \setminus Rej(\Gamma^R(W)) \cup Default(\Gamma^R(W)))$$

It has to exist a rule $\tau \in \rho(\mathcal{P}) \setminus Rej(\Gamma^R(W))$ whose head is A and whose body is satisfies by W. We know that $\tau \notin Rej(\Gamma^R(W))$, then , by definition for each rule $\eta \in P_j\ i < j$ such that $\tau \bowtie \eta$, $B(\eta)$ is not true in $\Gamma^R(W)$, hence $B(\eta)$ is false in W by Theorem 4.5.4.

Let us suppose now that the default literal *not* A belongs to W. If *not* $A \in Default(\Gamma^R(W))$ then for each rule $A \leftarrow B.$ in $\bigcup P_i$, B is not true in $\Gamma^R(W)$ and hence false in W. if *not* $A \notin Default(\Gamma^R(W))$ then there exists a rule $\tau :$ *not* $A \leftarrow B.$ *in* P_i, $\tau \notin Rej(\Gamma^R(W))$, then, with by the same reasoning used above, for each rule $\eta \in P_j,\ i < j$ such that $\tau \bowtie \eta$, $B(\eta)$ is false in W.

This concludes the proof.

♢

Proof of Proposition 4.5.1:

By Definition 27, M is a fixpoint of the Γ^R operator. From Theorem 3.5.1 it follows that M is also a fixpoint of the Γ operator.

Then $\Gamma\Gamma^R(M) = \Gamma(M) = M$ i.e. M is a fixpoint of $\Gamma\Gamma^R$. By definition the $WFDy(\mathcal{P})$ is a subset of M.

♢

Proof of Theorem 4.5.2:

In the following we will use the notations $\Gamma, \Gamma^R, \Gamma', \Gamma^{R\prime}$ for, respectively: $\Gamma_{\mathcal{P}}$, $\Gamma^R_{\mathcal{P}}$,$\Gamma_{\mathcal{P}\prime}$, $\Gamma^R_{\mathcal{P}\prime}$. Moreover, let W and W' be the well founded models of, respectively, $\mathcal{P}$ and $\mathcal{P}'$, i.e. the least fixpoints of, respectively, $\Gamma\Gamma^R$ and $\Gamma'\Gamma^{R\prime}$.

We will prove that *i*) W is a fixpoint of $\Gamma'\Gamma^{R\prime}$ and *ii*) W' is a fixpoint of $\Gamma\Gamma^R$. This will conclude the proof since, by minimality of the least fixpoint, from *i*) it follows that $W' \subseteq W$ and by *ii*) it follows $W \subseteq W'$.

We start by some technical results which are useful in the proof of *i*) and *ii*). By definition, for any interpretation X:

$$\begin{aligned}
\Gamma(X) = least(Generator(\mathcal{P},X)) &= least(\rho(\mathcal{P}) \setminus Rej(\mathcal{P},X) \cup Default(\mathcal{P},X)) \\
\Gamma^R(X) = least(Generator^R(\mathcal{P},X)) &= least(\rho(\mathcal{P}) \setminus Rej^R(\mathcal{P},X) \cup Default(\mathcal{P},X)) \\
\Gamma'(X) = least(Generator(\mathcal{P}',X)) &= least(\rho(\mathcal{P}') \setminus Rej(\mathcal{P}',X) \cup Default(\mathcal{P}',X)) \\
\Gamma^{R\prime}(X) = least(Generator^R(\mathcal{P}',X)) &= least(\rho(\mathcal{P}') \setminus Rej^R(\mathcal{P}',X) \cup Default(\mathcal{P}',X))
\end{aligned}$$

Since $\rho(\mathcal{P}')$ is equal to $\rho(\mathcal{P})$ plus a set of tautologies, by Corollary A.1.1 it follows:

$$\begin{aligned}
\Gamma'(X) = least(G(X)) &= least(\rho(\mathcal{P}) \setminus Rej(\mathcal{P}',X) \cup Default(\mathcal{P}',X)) \\
\Gamma^{R\prime}(X) = least(G^R(X)) &= least(\rho(\mathcal{P}) \setminus Rej^R(\mathcal{P}',X) \cup Default(\mathcal{P}',X))
\end{aligned}$$

with $G(X) \subseteq Generator(\mathcal{P},X)$ and $G^R(X) \subseteq Generator^R(\mathcal{P},X)$. This inclusions may be strict since tautologies may reject other rules or default assumption.

We are now ready to prove *i* and *ii*).

i) Let W be the well founded model of $\mathcal{P}$. By what we said above:

$$\Gamma^{R\prime}(W) = least(G^R(W)) \subseteq least(Generator^R(\mathcal{P},W)) = \Gamma^R(W)$$

We have to prove the inverse inclusion. The rules in $G^R(W)$ are a subset of the rules in $Generator^R(\mathcal{P},W)$. Each rule τ with head L in $Generator^R(\mathcal{P},W)$ which is not in $G^R(W)$ is a rule or a default assumption rejected by a tautology $not\,L \leftarrow B$ with $not\,L \in B$. Hence $W \models not\,L$ (otherwise the tautology could not reject τ). By Property 4.5.4 $L = not\,not\,L \notin \Gamma^R(W)$. Hence *the heads of the rules in* $Generator^R(\mathcal{P},W)$ *that are not in* $G^R(W)$ *do not belong to* $least(Generator^R(\mathcal{P},W))$ hence

$$\Gamma^{R\prime}(W) = least(G^R(W)) = least(Generator^R(\mathcal{P},W)) = \Gamma^R(W)$$

By the equality above it follows:

$$\Gamma'\Gamma^{R\prime}(W) = \Gamma'\Gamma^{R}(W) \subseteq \Gamma\Gamma^{R}(W) = W$$

It remains to prove the inverse inclusion. We know that

$$\Gamma'\Gamma^{R}(W) = least(G(\Gamma^{R}(W)))$$

The rules in $G(\Gamma^R(W))$ are a subset of the rules in $Generator^R(\mathcal{P},\Gamma^R(W))$. Each rule τ with head L in $Generator^R(\mathcal{P},\Gamma^R(W))$ which is not in $G(\Gamma^R(W))$ is a rule or a default assumption rejected by a tautology $not\,L \leftarrow B$ with $not\,L \in B$. Hence $\Gamma^R(W) \models not\,L$ (otherwise the tautology could not reject τ). By Property 4.5.4 $L = not\,not\,L \notin W$. Hence *the heads of the rules in* $Generator(\mathcal{P},\Gamma^R(W))$ *that are not in* $G(\Gamma^R(W))$ *do not belong to* $least(Generator(\mathcal{P},\Gamma^R(W)))$ hence

$$\Gamma'(\Gamma^{R}(W)) = least(G(\Gamma^{R}(W))) = least(Generator(\mathcal{P},\Gamma^{R}(W))) = \Gamma(\Gamma^{R}(W)) = W$$

as desired.

ii) Let W' be the well founded model of $\mathcal{P}'$. Then:

$$\Gamma^{R'}(W') = least(G^{R}(W')) \subseteq least(Generator^{R}(\mathcal{P},W')) = \Gamma^{R}(W')$$

If a rule τ with head L belongs to $Generator^R(\mathcal{P},W')$ but not to $G^R(W')$ then τ is either a rule or a default assumption rejected by some tautology $not\,L \leftarrow B$ with $not\,L \in B$. Then $not\,L \in W'$ (otherwise the tautology could not reject other rules). For $not\,L$ to belong to W' it means that $not\,L$ belongs to $Default(\mathcal{P}',\Gamma^{R\prime}(W'))$ or to $Rej(\mathcal{P}',\Gamma^{R\prime}(W'))$. In the first case, L is an atom and there exists no rule $L \leftarrow B2$ with $\Gamma^{R\prime}(W') \models B$. Hence $\Gamma^{R\prime}(W') \not\models B(\tau)$. In the second case, there exists a rule $\eta : not\,L \leftarrow B1$ with $W' \models B1$ in some P_i such that, for each rule $L \leftarrow B2$ in any P_j with $i \leq j$: $\Gamma^{R\prime}(W') \not\models B$. Since $\tau \in Generator^R(\mathcal{P},W')$ then, τ cannot be a default assumption since it would be rejected by η, then $\tau \in P_j$ for some $j > i$ otherwise it would be rejected by η. Then $\Gamma^{R\prime}(W') \not\models B(\tau)$. Since

$$\Gamma^{R\prime}(W') = least(G^{R}(W'))$$

Then $Generator^R(\mathcal{P},W')$ is a definite program obtained by adding to $G^R(W')$ a set of rules U (whose bodies are not satisfied) such for any rule $\tau \in U$, $B(\tau) \not\subseteq least(G^R(W'))$. Hence, by Property A.1.1,

$$\Gamma^{R}(W') = least(Generator^{R}(\mathcal{P},W')) = least(G^{R}(W)) = \Gamma^{R\prime}(W')$$

The inclusion above implies:

$$\Gamma\Gamma^R(W') = \Gamma\Gamma^{R\prime}(W') = least(G(\Gamma^{R\prime}(W\prime)) \qquad \subseteq$$
$$least(Generator(\mathcal{P}, \Gamma^{R\prime}(W')) = \Gamma'\Gamma^R(W') = W'$$

If a rule τ with head L belongs to $Generator(\mathcal{P}, \Gamma^{R\prime}(W'))$ but not to $G^R(\Gamma^{R\prime}(W')$ then τ is either a rule or a default assumption rejected by some tautology $not\, L \leftarrow B$ with $not\, L \in B$. Then $not\, L \in \Gamma^{R\prime}(W')$ (otherwise the tautology could not reject other rules). For $not\, L$ to belong to $\Gamma^{R\prime}(W')$ it means that $not\, L$ belongs to $Default(\mathcal{P}', W'))$ or to $Rej(\mathcal{P}', W')$. In the first case, L is an atom and there exists no rule $L \leftarrow B2$ with $W' \models B$. Hence $W' \not\models B(\tau)$. In the second case, there exists a rule $\eta : not\, L \leftarrow B1$ with $\Gamma^{R\prime}(W') \models B1$ in some P_i such that, for each rule $L \leftarrow B2$ in any P_j with $i < j$: $W' \not\models B$. Since $\tau \in Generator(\mathcal{P}, \Gamma^{R\prime}(W'))$ then, τ cannot be a default assumption since it would be rejected by η, then $\tau \in P_j$ for some $j \geq i$ otherwise it would be rejected by η. Then $W' \not\models B(\tau)$. Since

$$W' = least(G(\Gamma^{R\prime}(W')))$$

Then $Generator(\mathcal{P}, \Gamma^{R\prime}(W'))$ is a definite program obtained by adding to $G(\Gamma^{R\prime}(W'))$ a set of rules U (whose bodies are not satisfied) such for any rule $\tau \in U, B(\tau) \not\subseteq least(G(\Gamma^{R\prime}(W'))$. Hence, by Property A.1.1,

$$\Gamma(\Gamma^{R\prime}(W')) = least(Generator(\mathcal{P}, \Gamma^{R\prime}(W'))) = least(G(\Gamma^{R\prime}(W'))) = \Gamma'\Gamma^{R\prime}((W')) = W'$$

as desired.

♢

Proof of Proposition 4.5.4:

Let us suppose $L = A$. By definition $A \in F$ iff there exists a rule

$$\tau :\ A \leftarrow B_1,\ \tau \in P_i,\ F \models B_1$$

such that there does not exist

$$\eta :\ not\ A \leftarrow B_2,\ \eta \in P_j,\ i < j,\ \Gamma^R(F) \models B_2$$

By definition $not\ A \notin \Gamma^R(F)$ iff there exists a rule

$$\tau : A \leftarrow B_1,\ \tau \in P_i,\ F \models B_1$$

such that there does not exist a rule

$$\eta :\ A \leftarrow B_2,\ \eta \in P_j,\ i < j,\ \ \Gamma^R(F) \models B_2$$

The two condition are the same then $A \in F$ iff *not* $A \notin \Gamma^R(F)$

Let us suppose $L = not\ A$. By definition *not* $A \in F$ iff for each rule

$$\tau:\ A \leftarrow B_1,\ \tau \in P_i,\ \ \Gamma^R(F) \models B_1$$

there exists a rule

$$\eta:\ not\ A \leftarrow B_2,\ \eta \in P_j,\ i \leq j,\ F \models B_2$$

By definition $A \notin \Gamma^R(F)$ iff for each rule

$$tau : A \leftarrow B_1,\ \tau \in P_i,\ \Gamma^R(F)\ \models B_1$$

there exists a rule

$$\eta:\ not\ A \leftarrow B_2,\ \eta \in P_j,\ i \leq j,\ F \models B_2$$

The two condition are the same then *not* $A \in F$ iff $A \notin \Gamma^R(F)$ ♢

Before to proceed with the proof of Theorem 4.5.3 we need the following technical result. For any ordinal α, in the following we adopt the notation W_α for $\Gamma\Gamma^R \uparrow^\alpha$.

Lemma B.0.4 *Let $\mathcal{P}$ be any Dynamic Logic Program and α a limit ordinal. Then the following equivalence holds:*

$$\Gamma^R(W_\alpha) \ = \ \bigcap_{\beta<\alpha} \Gamma^R(W_\beta) \tag{B.1}$$

proof

Given two ordinals, $\beta 1$ and $\beta 2$ be two ordinals with with $\beta 1 \leq \beta 2$. Then, by inclusion 2.1 it follows:

$$W_{\beta 1} \subseteq W_{\beta 2} \tag{B.2}$$

and from the anti monotonicity of Γ^R it follows:

$$\Gamma^R(W_{\beta 1}) \supseteq \Gamma^R(W_{\beta 2})$$

Hence to prove equivalence B.1 it is sufficient to prove that a literal L belongs to $\Gamma^R(W_\alpha)$ iff L *definitively*[1] belongs to $\Gamma^R(W_\beta)$ for some β .

By definition: $W_\alpha = \bigcup_{\beta<\alpha} W_\beta$, hence by inclusion B.2 it follows that *i*) L belongs to W_α iff it definitively belongs to W_β for some β.

This means W_α satisfies the body of a rule iff the succession of W_β definitively satisfies it.

[1]Here and in the following definitively means that the statement is true for the considered ordinal and for any ordinal grater than it

Moreover we prove that *ii*) any finite subset of $\rho(\mathcal{P}) \setminus Rej^R(W_\alpha) \cup Default(W_\alpha)$ definitively belongs to $\rho(\mathcal{P}) \setminus Rej^R(W_\beta) \cup Default(W_\beta)$ for some β.

Let us suppose L is the negative literal *not A* and *not A* belongs to $Default(W_\alpha)$. This means, by definition, iff there does not exist a rule $A \leftarrow B$ such that W_α satisfies B which is equivalent to say that W_β definitively satisfies B which means iff L definitively belongs to $Default(W_\beta)$.

Let $\tau \in P_i$ be any rule that belongs to $\rho(\mathcal{P}) \setminus Rej^R(W_\alpha)$. By definition, this is equivalent to say that there does not exist a rule $\eta \in P_j$ with $i \leq j$ such that $\tau \bowtie \eta$ and W_α satisfies $B(\eta)$. This is equivalent to say τ definitively belongs to $P \setminus Rej^R(W_\beta)$ for $\beta \leq \alpha$. Hence *ii*) is proved.

By *i*) and by Property A.1.5 follows that

$$L \in \Gamma^R(W_\alpha) = least(\rho(\mathcal{P}) \setminus Rej^R(W_\alpha) \cup Default(W_\alpha))$$

if and only if L belongs to the least Herbrand Model of some P' that is a finite subprogram of

$$\rho(\mathcal{P}) \setminus Rej^R(W_\gamma) \cup Default(W_\gamma)$$

for some γ: $\gamma < \alpha$.

By *ii*) P' is a finite subprogram of

$$\rho(\mathcal{P}) \setminus Rej^R(W_\gamma) \cup Default(W_\gamma)$$

for some γ: $\gamma < \alpha$ iff P' definitively belongs to

$$\rho(\mathcal{P}) \setminus Rej^R(W_\beta) \cup Default(W_\beta)$$

for all $\beta \geq \gamma$ i.e. L definitively belongs to $\Gamma^R(W_\beta)$ since L belongs to $least(P')$.

$\diamondsuit$

Proof of Theorem 4.5.3:

We have to prove that the well founded model W of a DyLP is consistent If and only if

$$\forall \tau, \eta \in P_i,\ (\tau \bowtie \eta, W \models body(\tau), W \models body(\eta)) \Rightarrow$$

$$\exists \gamma \in P_j\ i < j\ such\ that\ \gamma \bowtie \tau\ or\ \gamma \bowtie \eta\ and\ \Gamma^R(W) \models body(\gamma)$$

From Proposition 4.5.6 we know that W is consistent iff $W \subseteq \Gamma^R(W)$.

Let us suppose W is consistent.

We prove *by contradiction* the implication above. Let us suppose there exist two rules:

- τ: $L \leftarrow B_1$ $\tau \in P_i$ $W \models B_1$

- η : $not\ L \leftarrow B_2\ \eta \in P_i\ W \models B_2$

such that there does not exist a rule

$$\gamma \bowtie \tau\ or\ \gamma \bowtie \eta\ and\ \Gamma^R(W) \models B(\gamma)$$

This implies that both τ and η are in $\rho(\mathcal{P}) \setminus Rej(\Gamma^R(W))$ then W satisfies both rules and, since it satisfies also the body of both, L, $not\ L\ \in W$

Let us suppose now that W satisfies the condition above, we have to prove that W is consistent i.e. that $W \subseteq \Gamma^R(W)$

We prove by transfinite induction that $W_\alpha \subseteq \Gamma^R(W_\alpha)$. This implies, sine $W = W_\alpha$ for some ordinal α that $W \subseteq \Gamma^R(W)$ and concludes the proof.

Basic step $W_0 = \emptyset$ then the condition trivially holds proved

Transfinite step Let $W_\alpha = \bigcup_{\beta<\alpha} W_\beta$. By inductive hypothesis:

$$W_\beta \subseteq \Gamma^R(W_\beta) \subseteq \bigcap_{\varepsilon \leq \beta} \Gamma^R(W_\varepsilon)$$

The last inclusion comes from the fact that $W_\varepsilon \subseteq W_\beta$ for $\varepsilon \leq \beta$ and hence, by anti monotonicity of Γ^R, $\Gamma^R(W_\beta) \subseteq \Gamma^R(W_\varepsilon)$ for $\varepsilon \leq \beta$.

By definition of W_α this implies also

$$W_\alpha \subseteq \bigcap_{\beta \leq \alpha} \Gamma^R(W_\beta)$$

By Lemma B.0.4 $\bigcap_{\beta\leq\alpha} \Gamma^R(W_\beta) = \Gamma^R(W_\alpha)$ and thus

$$W \subseteq \Gamma^R(W)$$

as desired.

Inductive step Let $W_\beta \subseteq W$ the β^{th} iteration of $\Gamma\Gamma^S$ starting from the empty set and let us assume, by inductive hypothesis that $W_\beta \subseteq \Gamma^R(W_\beta)$. Let $W_\alpha = \Gamma\Gamma^R(W_\beta)$, we have to prove $W_\alpha \subseteq \Gamma^R(W_\alpha)$

As an intermediate step we prove that $W_\alpha \subseteq \Gamma^R(W_\beta)$.

By definition:

$$\begin{aligned} W_\alpha &= least(\rho(\mathcal{P}) \setminus Rej(\Gamma^R(W_\beta)) \cup Default(\Gamma^R(W_\beta))) = least(Q) \\ \Gamma^R(W_\beta) &= least(\rho(\mathcal{P}) \setminus Rej^R(W_\beta) \cup Default(W_\beta)) = least(Q_T) \end{aligned}$$

where Q and Q_T are definite logic programs. By the anti monotonicity of $Default$ and by the inductive hypothesis it follows:

$$Default(\Gamma^R(W_\beta)) \subseteq Default(W_\beta)$$

Hence, the set $Rej^R(W_\beta) \setminus Rej(\Gamma^R(W_\beta))$ is the set of rules that belong to Q but not to Q_T. We will prove that this set is formed by rules whose body is not satisfied by $W_\alpha = least(Q)$ or by rules whose heads belong to $\Gamma^R(W_\beta) = least(Q_T)$. This, by Property A.1.1 implies that $least(Q) = least(Q'_T)$ for some subset Q'_T of Q_T and hence

$$W_\alpha = least(Q) \subseteq least(Q_T) = \Gamma^R(W_\beta)$$

By the monotonicity of Rej^R and by the inductive hypothesis it follows:

$$Rej^R(W_\beta) \subseteq Rej^R(\Gamma^R(W_\beta))$$

and hence $Rej^R(W_\beta) \setminus Rej(\Gamma^R(W_\beta)) \subseteq Rej^R(W_\beta) \setminus Rej(W_\beta)$.

We want to prove that the head of any rule in $Rej^R(W_\beta) \setminus Rej(\Gamma^R(W_\beta))$ whose body is satisfied by W_β belongs to $\Gamma^R(W_\beta)$.

By the last inclusion, for any rule $\tau \in P_i$ that belongs to $Rej^R(W_\beta) \setminus Rej(\Gamma^R(W_\beta))$, there exists a conflicting rule η in the same update whose body is satisfied by W_β (and hence by W). If the body of τ is satisfied by W_α (and hence by W) by hypothesis there exists a rule γ in a later update conflicting with either τ or η whose body is satisfied by $\Gamma^R(W)$. Hence, since $\Gamma^R(W) \subseteq \Gamma^R(W_\beta)$, the body of γ is also satisfied by $\Gamma^R(W_\beta)$. Since τ does not belong to $Rej(\Gamma^R(W_\beta))$, τ it is not rejected by any rule, and hence it is not rejected by γ i.e. τ and γ are not conflicting rules and hence they have the same head ($hd(\tau) = hd(\gamma)$). Moreover γ is not in $Rej^R(W_\beta)$, otherwise, since $W_\beta \subseteq \Gamma^R(W_\beta)$ τ would be rejected by the same rule and hence τ would belong to $Rej(\Gamma^R(W_\beta))$. Hence γ is a non rejected rule with the same head of τ whose body is true in $\Gamma^R(W_\beta)$ and hence its head (which is also the head of τ) belongs to $\Gamma^R(W_\beta)$.

Hence, by Property A.1.1 we conclude that:

$$W_\alpha \subseteq \Gamma^R(W_\beta)$$

By definition:

$$\begin{aligned} W_\alpha &= least(\rho(\mathcal{P}) \setminus Rej(\Gamma^R(W_\beta)) \cup Default(\Gamma^R(W_\beta))) = least(Q) \\ \Gamma^R(W_\alpha) &= least(\rho(\mathcal{P}) \setminus Rej^R(W_\alpha) \cup Default(W_\alpha)) = least(Q_R) \end{aligned}$$

where Q and Q_R are definite logic programs.

By antimonotonicity of $Default$ and the monotonicity of Rej^R we derive the following inclusion:

$$Default(\Gamma^R(W_\beta)) \subseteq Default(W_\alpha)$$

Hence, the set $Rej^R(W_\alpha) \setminus Rej(\Gamma^R(W_\beta))$ is the set of rules that belong to Q but not to Q_R. We will prove that this set is formed by rules whose body is not satisfied by $W_\alpha = least(Q)$ or by rules whose heads belong to $\Gamma^R(W_\alpha) = least(Q_R)$. This, by Property A.1.1 implies that $least(Q) = least(Q'_R)$ for some subset Q'_R of Q_R and hence

$$W_\alpha = least(Q) \subseteq least(Q_R) = \Gamma^R(W_\alpha)$$

By the monotonicity of Rej^R we derive the following inclusion:

$$Rej(W_\alpha) \subseteq Rej(\Gamma^R(W_\beta))$$

and hence $Rej^R(W_\alpha) \setminus Rej(W_\alpha) \subseteq Rej^R(W_\alpha) \setminus Rej(W_\alpha)$.

We want to prove that the head of any rule in $Rej^R(W_\alpha) \setminus Rej(\Gamma^R(W_\beta))$ whose body is satisfied by W_α belongs to $\Gamma^R(W_\alpha)$.

By the last inclusion, for any rule $\tau \in P_i$ that belongs to $Rej^R(W_\alpha) \setminus Rej(\Gamma^R(W_\beta))$, there exists a conflicting rule η in the same update whose body is satisfied by W_α (and hence by W). If the body of τ is satisfied by W_α (and hence by W) by hypothesis there exists a rule γ in a later update conflicting with either τ or η whose body is satisfied by $\Gamma^R(W)$. Hence, since $\Gamma^R(W) \subseteq \Gamma^R(W_\alpha)$, the body of γ is also satisfied by $\Gamma^R(W_\alpha)$. Since τ does not belong to $Rej(\Gamma^R(W_\alpha))$, τ it is not rejected by any rule, and hence it is not rejected by γ i.e. τ and γ are not conflicting rules and hence they have the same head ($hd(\tau) = hd(\gamma)$). Moreover γ is not in $Rej^R(W_\alpha)$, otherwise, since $W_\alpha \subseteq \Gamma^R(W_\beta)$, τ would be rejected by the same rule and hence τ would belong to $Rej(\Gamma^R(W_\beta))$. Hence γ is a non rejected rule with the same head of τ whose body is true in $\Gamma^R(W_\alpha)$ and hence its head (which is also the head of τ) belongs to $\Gamma^R(W_\alpha)$.

Hence, by Property A.1.1 we conclude that:

$$W_\alpha \subseteq \Gamma^R(W_\alpha)$$

$\diamondsuit$

Proof of Theorem 4.6.2:

Let P be any generalized logic program in the language $\mathcal{L}$, P^{not}, $R1$, $R2$ as from Definition 17. We denote with Γ^{not} the operator $\Gamma_{P^{not}}$ and with Γ^{Snot} the operator $\Gamma^S_{P^{not}}$.

We will prove that W is a fixpoint of $\Gamma\Gamma^R$ iff there exist a fixpoint $W1$ of $\Gamma^{not}\Gamma^{Snot}$ such that $W|\mathcal{H} = W1|\mathcal{H}$.

Let W be any fixpoint of $\Gamma\Gamma^R$ Let $W1$ be the superset of W defined in the following way:

$$\begin{aligned} W1 &= W \cup F1 \cup F2 \cup F3 \\ F1 &= \{A_P : A_P \leftarrow B. \in R1 \wedge W \models B\} \\ F2 &= \{A_n : A_n \leftarrow B. \in R2 \wedge W \models B\} \\ F1 &= \{\neg A : A_n \in F2\} \end{aligned}$$

We have to prove that $W1$ is a fixpoint of $\Gamma^{not}\Gamma^{Snot}$.

Let $K1, K2$ be the following interpretations:

$$\begin{aligned} K1 &= \Gamma^{Snot}(W1) = least(W1 - P^{Snot}) \\ K2 &= least(\rho(\mathcal{P}) \setminus Rej^R(W) \cup NH(W) \cup R1 \cup R2 \cup R3_N) = least(P_G) \end{aligned}$$

where P_G is a generalized logic program and

$$R3_N = \{A \leftarrow A_p,\ not\ \neg A.\ \ \neg A \leftarrow A_n,\ not\ \neg A.\ \forall A \in \mathcal{H}\}$$

It is clear, by Definition 43, that $K2|\mathcal{L} = \Gamma_P^{GR}(W)$

First of all we show that $K1 = K2 \setminus NH(W)$.

We show, by successive steps of partial evaluation (see Theorem A.2.1), that $least(P_G) = least(W1 - P^{Snot})$ if we restrict the language to the literals of P^{Snot}

Step 1 Let $E1$ be the program obtained from P^{Snot} by replacing every clause

$$A \leftarrow A_P,\ not\ \neg A.$$

in P^{Snot} with the clauses

$$A \leftarrow B_i,\ not\ \neg A.$$

for each rule

$$A_P \leftarrow B_i.$$

in P^{Snot}.

Step 2 Let $E2 = W1 - E1$ be the $W1$-reduce of E_1. By the principle of partial evaluation $E2$ and $W1 - P^{Snot}$ have the same least Herbrand model.

Step 3 Let $G1$ be the program obtained by P_G by eliminating from the body of each clause the default negative literals that are in $NH(W)$

By the principle of partial evaluation, $G1$ and P_G have the same least Herbrand model.

Then, let G_2 be the the program obtained from G_1 by pruning all the rules with negative head and all the rules

Let us consider now the programs $G2$ and $E2$. The process of partial evaluation we have performed over the default conclusion is equivalent to compute the $W1 - reduct$. Hence we have that the clauses of the form

$$A_p \leftarrow B.$$

$$A_n \leftarrow B.$$

$$\neg A \leftarrow A_n.$$

are in $E2$ iff they are in $G2$.

Finally the rule

$$A \leftarrow B.$$

is in $E2$ iff

$$A \leftarrow B', \; not \; \neg A.$$

is in $E1$ such that B is the set of positive literals in B', all the negative literals in B' are in $NH(W)$, $\neg A$ is in not in $W1$ i.e. it does not exits a rule

$$not \; A \leftarrow B.$$

in P such that $W \models B$. This mean

$$A \leftarrow B.$$

is in $E2$ iff it is in $G2$ hence the two programs coincide, then they have the same least Herbrand model.

Let us consider now the following interpretations:

$$\begin{aligned} M1 = \Gamma^{not}(K1) &= least(P^{not\ K1}) = least(\Gamma_P^{GR}(W) - P^{not}) \\ M2 &= least(P \cup \Gamma_P^{GR}(NH(W)) \cup R1 \cup R2 \cup R3) \end{aligned}$$

Where It is clear that $M2|\mathcal{L} = W$. This implies also $M2 = W1$

Using the same techniques of partial evaluation used above we prove $M1 = M2$, hence $M2 = W1$. We repeat steps 1-3. For what regards step 4, consider the set of rules

$$not \; A \leftarrow B.$$

of $G1$. Let us suppose $M2$ satisfies the body of a rule of this kind, then $\neg A \in W1$, this means $A \notin \Gamma_P^{GR}(W)$, i.e. $not\ A \in NH(\Gamma_P^{GR}(W))$. Then we can eliminate this rules since they are redundant, then step 4 remain unchanged. Also step 5 is unchanged.

We have then proved that if W is a fixpoint of $\Gamma_P\Gamma_P^R$ then there exist a fixpoint $W1$ of $\Gamma^T\Gamma^{TS}$ such that $W = W1|\mathcal{L}$.

Let now $W1$ be any fixpoint of $\Gamma^T\Gamma^{TS}$. Let us consider W defined in the following way.

$$W = W1|\mathcal{H} \cup \Gamma^{Snot}(W1)|\mathcal{H}^-$$

We will prove that W is a fixpoint of $\Gamma_P\Gamma_P^R$.

Let us consider

$$K1 = least(\rho(\mathcal{P}) \setminus Rej^R(W) \cup NH(W) \cup R1 \cup R2 \cup R3_N)$$

Clearly $K1|\mathcal{L} = \Gamma_P^{GR}(W)$. Let us consider now

$$K2 = \Gamma^{Snot}(W1) = least(P_N^{not\ W1}) = least(P_N^{not\ W|\mathcal{H}})$$

We observe preliminary that $\neg A$ is in $W1$ iff there exists in P a rule

$$not\ A \leftarrow B.$$

such that the positive literals in body are in $W1$, and Hence in W, and no one of the negative literals in B is in $\Gamma^{Snot}(W1)|\mathcal{H}$. This means that $\neg A$ is in $W1$ iff all the rules

$$A \leftarrow B.$$

of P are in $Rej^R(W)$.

After this considerations we can prove $K1|\mathcal{H}_{P^{not}} = K2$ by the same techniques of partial evaluation used above.

Then we obtain $\Gamma_P^{GR}(W)|\mathcal{H} = \Gamma^{Snot}(W1)|\mathcal{H}$.

Let us consider

$$M2 = least(P \cup NH(\Gamma_P^{GR}(W)) \cup R1 \cup R2 \cup R3)$$

Clearly $M2|\mathcal{L} = \Gamma_P^G(\Gamma_P^{GR}(W))$ We have also

$$W1 = least(P^{not\ K2}) =$$

Again we follow the partial evaluation steps 1-3. For what regards step 4, consider the set of rules

$$not\ A \leftarrow B.$$

of $G1$.

It is not clear at this point if these rules are redundant, hence consider $G2$ as constructed from steps 4-5. By the same considerations done above we can prove that $G2$ and $E2$ have the same least Herbrand model. We have to prove that $least(G2)$ and $least(G2)$ coincide if we restrict to positive conclusions. Clearly

$$least(G2) \subseteq least(G1)$$

If there exist a positive literal that belongs to $least(G1) \setminus least(G2)$ it has to exists a rule

$$not\ A \leftarrow B.$$

of $G1$ such that $least(G)$ satisfies B and $not\ A \notin NH(\Gamma_P^{GR}(W))$. If $least(G2) = W1$ satisfies B, then $\neg A$ is in $W1$. This mean $A \notin \Gamma_P^{GR}(W)$, hence $not\ A \in NH(\Gamma_P^{GR}(W))$, against hypothesis. This prove that $least(G2)$ and $least(G2)$ coincide if we restrict to positive conclusions. The rest of the proof follows as above.

Then we have

$$W|\mathcal{H} = W1|\mathcal{H} = M2|\mathcal{H} = \Gamma_P^G(\Gamma_P^{GR}(W))|\mathcal{H}$$

Further we have that the default literals of $\Gamma_P^G(\Gamma_P^{GR}(W))$ are exactly the literals of

$$NH(\Gamma_P^{GR}(W)) = (\Gamma^{Snot}(W1)|NH(\mathcal{H}))$$

By definition these are the negative literals of W, hence W and $\Gamma_P^G(\Gamma_P^G(W))$ coincide. We have then proved that

$$\Gamma_P^G(\Gamma_P^{GR}(W))|\mathcal{H} = W|\mathcal{H}$$

We have then proved that W is a fixpoint of $\Gamma_P\Gamma_P^R$ iff there exist a fixpoint $W1$ of $\Gamma^T\Gamma^{TS}$ such that $W|\mathcal{H} = W1|\mathcal{H}$.

Let now W, $W1$ be such fixpoints. By definition we know that $W1 \cup NH(\Gamma^{Snot}(W1)))$ is a (paraconsistent) partial stable model of P^{not} and hence $(W1 \cup NH(\Gamma^{Snot}(W1)))|\mathcal{L}$ is a (paraconsistent) partial stable model of P following Definition 42. From the discussion above it follows also that $\Gamma_P^{GR}(W)|\mathcal{H} = \Gamma^{Snot}(W1)|\mathcal{H}$, then by Theorem B.0.1 and Proposition 4.5.5 we obtain

$$NH(\Gamma^{Snot}(W1)|\mathcal{L}) = \{not\ A\ \in\ W\}$$

and finally

$$W = (W1 \cup NH(\Gamma^{Snot}(W1)))|\mathcal{L}$$

This means that the fixpoints of $\Gamma_P\Gamma_P^R$ and the paraconsistent partial stable models of P coincide, and hence the two definitions of well founded semantics also coincide.

$\diamondsuit$

Proof of Theorem 4.6.1:

To prove Theorem 4.6.1 we prove a more general result; an interpretation F is a the fixpoints of $\Gamma\Gamma^R$ iff it is also a fixpoint of $\Gamma^G\Gamma^{GS}$. Since the well founded model is the intersection of all the fixpoints, as a corollary, we obtain that the well founded models of the two semantics coincide.

Theorem B.0.1 *Let P be the Dynamic Logic Program of one single update and Then*

$$F = \Gamma\Gamma^R(F) \;\Leftrightarrow\; F = \Gamma^G(\Gamma^{GR}(F))$$

Moreover $\Gamma^R(F) = \Gamma^{GR}(F)$

For simplicity we omitted P from the notation of the operators.
proof

Let F be a fixpoint of $\Gamma\Gamma^R(F)$. Then F is computed as the least Herbrand model of $P \cup Default(\Gamma^R(F))$. By definition:

$$NH(F) = \{not\, A : \; A \notin F\}$$

Since

$$A \in F = least(P \cup Default(\Gamma^R(F))) \;\Leftrightarrow\; \exists A \leftarrow B : F \models B$$

Then:

$$NH(F) = \{not\; A : \; \nexists A \leftarrow B \; \in P,\; F \models B\} = Default(F)$$

This implies, by definition of the two operators, that $\Gamma^R(F) = \Gamma^{GR}(F)$. Then

$$F = least(P \cup Default(\Gamma^R(F)) = least(P \cup Default(\Gamma^{GR}(F))$$

By definition

$$Default(\Gamma^{GR}(F)) = \{not\; A : \; \nexists \tau : \; A \leftarrow B\; \tau \in P,\; \Gamma^{GR}(F) \models B\}$$

and, since $\Gamma^{GR}(F) = least(P \setminus Rej^R(F) \cup Default(F))$:

$$NH(\Gamma^{GR}(F)) = \{not\; A : \; \nexists \tau : \; A \leftarrow B,\; \tau \in P \setminus Rej^R(F),\; \wedge\, \Gamma^{GR}(F) \models B\}$$

$$Rej^R(F) = \{\tau \in P : \;\; \exists\, \eta \;\in\; P : \; \tau \bowtie \eta \wedge F \models B(\eta) \;\}$$

This implies that $Default(\Gamma^{GR}(F)) \subseteq NH(\Gamma^{GR}(F))$ and if *not* A belongs to $NH(\Gamma^{GR}(F))$ but it does not belong to $Default(\Gamma^{GR}(F)$, then a rule with head A

was rejected i.e. there exists

$$\eta : \ not\ A \leftarrow B \in P,\ F \models B$$

Since F is a model of P, we obtain $not\ A \in F$. Hence, by Property A.1.1 we have

$$F = least(P \cup Default(\Gamma^{GR}(F))) = least(P \cup NH(\Gamma^{GR}(F))) = \Gamma^{G}(\Gamma^{GR}(F))$$

as desired.

Let F be a fixpoint of $\Gamma^{G}(\Gamma^{GR}(F))$. By definition:

$$NH(F) = \{not A : \ A \notin F\}$$

Since

$$A \in F = least(P \cup NH(\Gamma^{GR}(F))) \ \Leftrightarrow \ \exists A \leftarrow B : F \models B$$

Then:

$$NH(F) = \{not\ A : \ \nexists A \leftarrow B \ \in P,\ F \models B\} = Default(F)$$

Hence $\Gamma^{GR}(F) = \Gamma^{R}(F)$.

Then

$$F = \Gamma^{G}(\Gamma^{R}(F)) = least(P \cup NH(\Gamma^{R}(F)))$$

By definition

$$Default(\Gamma^{R}(F)) = \{not\ A : \ \nexists \tau : \ A \leftarrow B \ \wedge \ \tau \in P \ \wedge \ \Gamma^{R}(F) \models B\}$$

and

$$NH(\Gamma^{R}(F)) = \{not\ A : \ \nexists \tau : \ A \leftarrow B \ \wedge \ \tau \in P \setminus Rej^{R}(F) \ \wedge \ \Gamma^{R}(F) \models B\}$$

This implies that $Default(\Gamma^{R}(F)) \subseteq NH(\Gamma^{R}(F))$ and if $not\ A$ belongs to $NH(\Gamma^{R}(F))$ but it does not belong to $Default(\Gamma^{R}(F)$, then there exists

$$\eta : \ not\ A \leftarrow B \in P,\ F \models B$$

Since F is a model of P, we obtain $not\ A \in F$. Then, by Property A.1.1 we have

$$F = least(P \cup Default(\Gamma^{R}(F)) = \Gamma\Gamma^{R}(F)$$

$\diamondsuit$

As corollary we obtain Theorem 4.6.1

♢

Proof of Theorem 4.7.1:

We prove a stronger result from which the theorem derives as a consequence.

Theorem B.0.2 *Let $\mathcal{P}$be any Dynamic logic program, $\mathcal{P}^T$ its transformational equivalent.Then F be a fixpoint of $\Gamma_{\mathcal{P}^k}\Gamma^S_{\mathcal{P}^k}$ iff there exists a fixpoint F^T of $\Gamma^2_{\mathcal{P}^{Tk}}$ such that*

$$F = \{A \mid A \in F^T\} \cup \{not\ A \mid A^- \in F^T\}$$

We know that

$$WFDy(\mathcal{P}^k) = \bigcap F\ \forall F :\ F = \Gamma_{\mathcal{P}^k}\Gamma^S_{\mathcal{P}^k}(F)$$

From Theorem B.0.2 for each such F there is a corresponding F^T that is a fixpoint of $\Gamma^2_{\mathcal{P}^{Tk}}$ and since

$$W^{Tk} = \bigcap F^T$$

we obtain as a consequence Theorem 4.7.1.

proof

First of all let us consider a partially evaluated (see Theorem A.2.1) version of $\mathcal{P}^T$. We replace the rules of the form

$$rej(\overline{L}^R, j) \leftarrow rej(\overline{L}^R, i) \qquad rej(\overline{L}, j) \leftarrow rej(\overline{L}, i)$$

For each rule τ in P_j of the form

$$\tau :\ notL \leftarrow B$$

with the the set of rules

$$rejAt(L)_i \leftarrow \overline{B}$$

for each i such that $i \leq j$, there exists a rule η in P_j such that $\tau \bowtie \eta$and

$$rejAt(L)^S_i \leftarrow \overline{B}^S$$

for each i such that $i < j$, there exists a rule η in P_j such that $\tau \bowtie \eta$.

After this partial evaluation, we divide each P^T_i in two programs P^E_i and P^S_i in the following way: Each rule whose head is a literal of the form L^S is in P^S_i, all the other rules are in P^E_i. In this way we also obtain the two sequences of programs $\mathcal{P}^{Ek}$ and $\mathcal{P}^{Sk}$. We use Γ^T, Γ^E and Γ^O, instead of the $\Gamma_{\mathcal{P}^{Tk}}$, $\Gamma_{\mathcal{P}^{Ek}}$, $\Gamma_{\mathcal{P}^{Sk}}$ respectively.

We observe further that in the body of each rule in $\mathcal{P}^{Ek}$ there are only positive literals of the form $At(L)$ and at most one negative literal of the form $notrejAt(L)^S_i$. Conversely, in the body of each rule in $\mathcal{P}^{Sk}$ there are only positive literals of the form $At(L)^S$

and at most one negative literal of the form $notrejAt(L)_i$, where L is a literal of $\mathcal{L}$.

Let X be any subset of $\mathcal{L}^T$. We divide X in two parts X_1 and X_2 with

$$X_1 = X|\mathcal{L}^E \ , \ X_2 = X|\mathcal{L}^S$$

Define what $\mathcal{L}^E$ is

Then, from what previously said it results

$$\begin{aligned} \Gamma^{T2}(X) &= \\ \Gamma^T(\Gamma^E(X_2) \cup \Gamma^O(X_1)) &= \\ \Gamma^E(\Gamma^O(X_1)) \cup \Gamma^O(\Gamma^E(X_2)) &= \\ \Gamma^E(\Gamma^O(X_1))|\mathcal{L}^E \cup \Gamma^O(\Gamma^E(X_2))|\mathcal{L}^S & \end{aligned}$$

From now on we will use Γ and Γ^S instead of $\Gamma_{\mathcal{P}^k}$ and $\Gamma^S_{\mathcal{P}^k}$ respectively.

Let W^{Tk} be the well founded model of $\mathcal{P}^{Tk}$ and

$$W^S = W|\mathcal{L}^S$$

define L^S before

Let F be a fixpoint of $\Gamma\Gamma^S$. We define F^E in the following way

$$\begin{aligned} F^E &= \{A \mid A \in F\} \cup \{A^-|not\ A \in F\} \\ &\cup \{rejA_0^- \mid not\ A \notin Default(F)^k\} \\ &\cup \{rejAt(L)_i \mid \exists\, \tau \in P_i,\ \tau \in Rej^R(F)^k,\ hd(\tau) = L\} \end{aligned}$$

Then we consider $F^T = F^E \cup W^S$. By the previous argumentations it follows

$$\Gamma^{T\ 2}(F^T) = \Gamma^E\Gamma^O(F^E) \cup \Gamma^O\Gamma^E(W^S)$$

Since $F^T = F^E \cup W^S$, and $\Gamma^O\Gamma^E(W^S) = W^S$, we have simply to prove

$$\Gamma^E\Gamma^O(F^E) = F^E$$

By definition

$$\Gamma^O(F^E) = least(\bigcup_0^k E^E - P_i^O)$$

We have that

$$not\ A \in Default(F)^k \Leftrightarrow rejA_0^- \in F^E \Leftrightarrow A^{-S}. \in E - P_0^O$$

and

$$\tau \in P_i \cap Rej^R(F)^k \Leftrightarrow rejAt(hd(\tau))^S \in F^E \Leftrightarrow At(\tau)^S \in E - P_i^O$$

From this it follows

$$\forall L | L \in \mathcal{L},\ L \in \Gamma^R(F) \Leftrightarrow At(L) \in \Gamma^O(F^E)$$

and this implies

$$not\ A \in Default(\Gamma^R(F))^k \Leftrightarrow rejA^{-S} \in \Gamma^O(F^E) \tag{B.3}$$

$$\tau \in Rej^R(\Gamma^R(F))^k \cap P_i \Leftrightarrow rejAt(hd(\tau))^S \in \Gamma^O(F^E) \tag{B.4}$$

Let us consider now

$$\Gamma^E\Gamma^O(F^E) = least(\ \Gamma^O(F^E) - \mathcal{P}^{Ek}\ )$$

From B.3 and B.4 it follows

$$A^-. \in (\Gamma^O(F^E)) - P_0^E \Leftrightarrow not\ A \in Default(\Gamma^R(F))^k$$

$$At(\tau) \in (\Gamma^O(F^E)) - \mathcal{P}^{Ek} \Leftrightarrow \tau \in \bigcup_1^k P_i \setminus Rej^R(\Gamma^R(F))^k$$

Then it follows

$$At(L) \in \Gamma^E\Gamma^O(F^E) \Leftrightarrow L \in \Gamma\Gamma^R(F)$$

Moreover, by definition of Γ^E we derive also

$$rejA_0^- \in \Gamma^E\Gamma^O(F^E) \Leftrightarrow not\ A \in Default(F)^k \Leftrightarrow rejA_0^- \in F^E$$

$$rejAt(hd(\tau))_i \in \Gamma^E\Gamma^O(F^E) \Leftrightarrow \tau \in P_i,\ \tau \in Rej^R(F)^k \Leftrightarrow rejAt(L)_i \in F^E$$

And this finally implies $F^E = \Gamma^E\Gamma^O(F^E)$ and hence $F^T = \Gamma^{T\ 2}(F)$

Conversely, let F^T be a fixpoint of $\Gamma^{T\ 2}$. Let F be the following set of literals.

$$F = \{A \mid A \in F^T\} \cup \{not\ A \mid A^- \in F^T\}$$

We have to prove $F = \Gamma\Gamma^R(F)$.

Let consider the set $F^E = F^T|_{\mathcal{L}^E}$. Clearly $F \subseteq F^E$. From the previous discussion we now that $F^E = \Gamma^E\Gamma^O(F^E)$. By definition it follows

$$rejA_0^{-S} \in \Gamma^O(F^E) \Leftrightarrow not\ A \in Default(\Gamma^R((F))^k \tag{B.5}$$

$$rejAt(hd(\tau))_i^S \in \Gamma^O(F^E) \Leftrightarrow \tau \in P_i,\ \tau \in Rej^R(\Gamma^R(F)) \tag{B.6}$$

Then from $F^E = \Gamma^E\Gamma^O(F^E)$ it follows

$$\Gamma\Gamma^R(F) = least(\bigcup_1^n P_i \setminus Rej(\Gamma^R(F))) = F$$

This concludes the proof.

♢

As corollary we obtain Theorem 4.7.1.

Proof of Theorem 4.7.2: It follows immediately from Definition 44. The number of *defaults assumptions* is at most twice the size of the set of atoms in $\mathcal{L}$ i.e. l. The number of *rewritten rules* is exactly twice the number of all the rules in any update P_i i.e. $2m$. Each rule $L \leftarrow B$ in any P_i generates at most four rejection rules, one of the form $rej(\overline{not\, L}, j) \leftarrow \overline{B}$, another of the form $\overline{L}^R \leftarrow \overline{body}^R$, $not\ rej(\overline{L}, i)$, another one of the form $rej(\overline{L}, j) \leftarrow rej(\overline{L}, i)$, and another one of the form $rej(\overline{L}^R, j) \leftarrow rej(\overline{L}^R, i)$. Hence there are at most $4m$ rejection rules. There are no other rules in $\mathcal{P}^{Tn}$. Hence there are at most $6m + l$ rules in $\mathcal{P}^{Tn}$ ♢

♢

Appendix C

Proof of theorems from Chapters 5 and 6

Proof of Theorem 5.3.1: We separately prove the double implication of theorem 5.3.1 in the two directions:

Let us assume that s' is a resulting state from s given I, D and the set of actions K.

By Definition 51, this is equivalent to state that there exists an evolving stable model M_1, s^* of I, D given the input programs $s \cup K, \emptyset$ such that $s' \equiv_{\mathcal{F}} s^*$. An interpretation M_1 is an evolving stable of I, D given the input programs $s \cup K$ i.e. iff M_1 is a refined stable models of $I, D \cup s \cup K$ i.e.

$$M_1 = least\left((I \cup D \cup s \cup K) \setminus Rej^s(M_1, I, D \cup K \cup s) \cup Default(M_1)\right)$$

All the atoms of the form $event(\tau)$, where τ is the effect of a dynamic rule, are false by default assumption. Hence the rules of the form (5.3) and (5.5) play no role. Also, all the literals of the form $prev(Q)$ where Q is a fluent literal are false by default, hence the rules of the form $Q \leftarrow prev(Q)$ in I plays no role. Since s is consistent w.r.t. all the static rules, there is no conflict between the static rules in D, hence such rules does not reject any literal in s nor they infer any fluent literal that does not belong to s. Hence we can simplify the expression above in the following way:

$$M_1 = least\left((D^* \cup s \cup K) \cup Default(M_1)\right)$$

where D^* is the set of rules consisting of any rule of the form

$$assert(event(\tau)) \leftarrow Cond.$$

for every dynamic rule **effect(τ)** $\leftarrow Cond.$ in D and any rule of the form

$$assert(prev(Q)) \leftarrow Q. \qquad assert(not\ prev(Q)) \leftarrow not\ Q.$$

for every Q such that **inertial(Q)** belongs to D. Hereafter, for sake of simplicity

we will omit the negative literals of the form *not A* when A is an auxiliary literal or an action literal, i.e. $A \notin \mathcal{F}$. Moreover by $Prev(s)$ we indicate the set of literals which are either of the form $prev(F)$, where F is a fluent literal that is declared as inertial in D and is true in s, or of the form $not\ prev(F)$ where F is a fluent literal that is declared as inertial in D and is false in s.

Finally, by $ED(s,K)$ we mean the set of literals $event(\tau)$ such that

$$assert(event(\tau)) \leftarrow Cond.$$

belongs to D and $s \cup K \models Cond$. Hence, the unique refined stable model M_1 is:

$$M_1 = s \cup K \cup assert(Prev(s)) \cup assert(ED(s,K))$$

The trace associated with any evolving interpretation M_1, s^* is then the sequence $\mathcal{J}:\ I, D, Prev(s) \cup ED(s,K)$. Hence, M_1, s^* is an evolving stable model of I, D give the sequence of input programs $K, \emptyset$ iff s^* is a refined stable model of $\mathcal{J}$.

Let s^* be any interpretation over the language of I, D, and$s' = s*|\mathcal{F}$. To prove the theorem, we simply have to prove that s^* is a refined stable model of $\mathcal{J}$ iff s' satisfies the equality 5.6. By definition s^* is a refined stable model of $\mathcal{J}$ iff

$$s^* = least\left((I \cup D \cup Prev(s) \cup D(s,K)) \setminus Rej^R(s^*) \cup Default(s^*)\right)$$

We start by assuming s^* is a refined stable model of $\mathcal{J}$. We first simplify the expression above in order to obtain an expression similar to 5.6.

Let $s' = s^*_{\mathcal{F}}$. Since s' only contains fluent literals, the dynamic rules and the inertial declarations in D plays no role in determining s'. Hence, the only rules we are interested in are the static rules in $\mathcal{R}$. Moreover, since s^* is two valued, there is no mutual rejection between the rules in $\mathcal{R}$; otherwise there would be a fluent literal Q such that all the rules with head Q or *not* Q would be rejected and *not* Q would not be in the set $Default(s^*)$ as well. Hence neither Q nor *not* Q would be in s^* in contrast with the two valuedness of s^*. Finally, by partially evaluating the facts in $ED(s,K)$, and the rules of the form

$$F \leftarrow Body,\ event(F \leftarrow Body). \quad (5.3)$$

we can erase the facts of the form $event(\tau)$ from the rules (5.3) whenever $event(\tau) \in ED(s,K)$ and ignore the rules (5.3) whenever $event(\tau) \notin ED(s,K)$ Hence we obtain the following equivalence for s'.

$$s' = least\left(I \setminus Rej^R(s^*) \cup Prev(s) \cup \mathcal{R} \cup D(s,K) \cup Default(s^*)\right)$$

We can split the set of default assumptions in two subsets, the one concerning

the inertial fluent literals and the one concerning the fluent literals that are not inertial. We obtain then:

$$s' = least\left(I \setminus Rej^R(s^*) \cup Prev(s) \cup Default(s^*)|_I \cup \mathcal{R} \cup D(s,K) \cup Default(s^*)|_{(\mathcal{F}\setminus I)}\right)$$

where $Default(s^*)$ stands for $Default(s^*, I, R \cup D(s,K))|_{(\mathcal{F}\setminus I)}$. The expression

$$Default(s^*, I, R \cup D(s,K))|_{(\mathcal{F}\setminus I)}$$

is equivalent to

$$Default(s', \mathcal{R} \cup D(s,K))|_{(\mathcal{F}\setminus I)}$$

Moreover, the expression $Default(s^*)|_I$ is equivalent to

$$Default(s', s \cup \mathcal{R} \cup D(s,K))|_{(I)}$$

Let $Inherit(s)$ be the following set of rules

$$Inherit(s^*) = \{Q \in \mathcal{F} : Q \leftarrow prev(Q) \in I \setminus Rej^R(()s^*) \wedge prev(Q) \in Prev(s)\}$$

What remains to do in order to prove that s' satisfies the equality 5.6 is then to prove that:

$$Inherit(s^*) \cup Default(s^*)|_I) \equiv (s \cap s' \cap I)$$

We consider separately the negative and the positive fluent literals. Let Q be a fluent literal that belongs to $(s \cap s' \cap I)$. We want to prove this is equivalent to saying that $Q \leftarrow prev(Q)$ belongs to $I \setminus Rej^R(s^*)$ and that $Prev(Q) \in Prev(s)$ i.e $Q \in Inherit(s^*)$.

The literal Q belongs to $(s \cap s' \cap I)$ iff $Q \in I$, $not\, Q \notin s$ and $not\, Q \notin s'$. This implies that there exists no rule in $\mathcal{R} \cup D(s,K)$ whose head is $not\, Q$ and whose body is true. Hence the rule $Q \leftarrow prev(Q)$ belongs to $I \setminus Rej^R(s^*)$ and, by $Q \in s$ we derive, by definition of $Prev(s)$, that $Prev(Q) \in Prev(s)$.

On the other hand, if $Q \leftarrow prev(Q)$ belongs to $I \setminus Rej^R(s^*)$, this implies that there exists no rule in $\mathcal{R} \cup D(s,K)$ whose head is $not\, Q$ and whose body is true, and if $Prev(Q) \in Prev(s)$, this implies $not\, Q \notin Default(s^*)$ and hence $not\, Q$ is not derived by any rules nor by default assumption. Hence $not\, Q \notin s'$ and so $Q \in s$. On the other hand, $prev(Q) \in Prev(s)$ implies, by definition, $Q \in s$ and $Q \in I$. Thus, we have proved that

$$Q \in (s \cap s' \cap I) \Leftrightarrow Q \leftarrow prev(Q) \in I \setminus Rej^R(s^*) \ \wedge \ prev(Q) \in Prev(s)$$

Let us now consider the negative fluent literals. We want to prove that, for any

inertial fluent, the following double implication holds.

$$not\,Q \in (s \cap s') \Leftrightarrow not\,Q \in Default(s', s \cup \mathcal{R} \cup D(s,K))|\mathcal{F}$$

We know $not\,Q \in s'$ iff $Q \notin s'$, which implies, since s' is a model of $\mathcal{R} \cup D(s,K)$ that there exists no rule in $\mathcal{R} \cup D(s,K)$ whose head is Q and whose body is satisfied by s'. This, together with the fact that $Q \notin s$, by definition of $Default$ implies $not\,Q \in Default(s', s \cup \mathcal{R} \cup D(s,K))$ as desired.

We have hence proved

$$Inherit(s^*) \cup Default(s^*)|_I) \equiv (s \cap s' \cap I)$$

and so that s' satisfies the equality 5.6.

Let us assume now that s' satisfies the equality 5.6, i.e.

$$s' = least\left((s \cap s' \cap I) \cup Default(s', \mathcal{R} \cup D(s,K))|_{(\mathcal{F} \setminus I)} \cup D(s,k) \cup \mathcal{R}\right)$$

Let NED be the set of literals of the form $\neg event(\tau)$ such that $event(\tau) \in ED(s,K)$ and there is no dynamic rule of the form **effect(τ)** $\leftarrow Cond$ such that s' satisfies $Cond$. Let s' be the following evolving interpretation (again we omit the negative literals which are not fluent literals).

$$s^* = s' \cup Prev(s) \cup ED(s,K) \cup NED \cup assert(ED(s',K)) \cup assert(Prev(s))'$$

We have to prove that s^* is a refined stable model of $\mathcal{J}$. We start by proving that

$$Inherit(s^*) \cup Default(s^*)|_I) \equiv (s \cap s' \cap I)$$

Let Q be a fluent literal in $(s \cap s' \cap I)$. This is equivalent to $Prev(Q) \in Prev(s)$, $not\,Q \notin s'$. Since s' is a model of $\mathcal{R} \cup D(s,K)$, we conclude there exists no rule in $\mathcal{R} \cup D(s,K)$ with head $not\,Q$ and true body in s'. Hence $Q \leftarrow prev(Q) \in I \setminus Rej^R(s^*)$ and thus $Q \in Inherit(s^*)$. On the other hand, if $Q \in Inherit(s^*)$, i.e. $Q \leftarrow prev(Q) \in I \setminus Rej^R(s^*)$ and $prev(Q) \in Prev(s)$, then $Q \in s$, i.e. $not\,Q \notin s$, $Q \in I$, and there exists no rule with head Q whose body is true in s'. So $not\,Q \notin s'$, i.e. $Q \in s'$ and finally $Q \in (s \cap s' \cap I)$.

Let us now consider the negative fluent literals. We want to prove that, for any inertial fluent, the following equivalence holds.

$$not\,Q \in (s \cap s') \Leftrightarrow not\,Q \in Default(s', s \cup \mathcal{R} \cup D(s,K))|\mathcal{F}$$

The proof proceeds in the same way as above, and we conclude

$$Inherit(s^*) \cup Default(s^*)|_I) \equiv (s \cap s' \cap I)$$

We obtain then the following equality:

$$s' = least\left(Inherit(s^*) \cup Default(s^*)|_I) \cup Default(s', \mathcal{R} \cup D(s,K))|_{(\mathcal{F} \setminus I)} \cup D(s,k) \cup \mathcal{R}\right)$$

Which is equivalent to

$$s' = least\,(\,Inherit(s^*) \cup Default(s^*) \cup D(s,k) \cup \mathcal{R})\,|_{\mathcal{F}}$$

Since s' is consistent w.r.t. $D(s,K)$ and $\mathcal{R}$, these sets of rules do not contain any pair of rules with conflicting heads and whose bodies are both true in s'. Hence, replacing $Inherit(s^*)$ with $Prev(s) \cup I \setminus Rej^R(s^*)$ we obtain

$$s' = least\,\big(\,(I \cup D(s,K) \cup mR) \setminus Rej^R(s^*) \cup Default(s^*)\big)\,|_{\mathcal{F}}$$

and from this, considering the definition of s^*

$$s^* = least\left((I \cup D \cup Prev(s) \cup D(s,K)) \setminus Rej^R(s^*) \cup Default(s^*)\right)$$

This is equivalent to saying that M_1, s^* is an evolving stable model of I, D given the sequence of input programs $K, \emptyset$, i.e. s' is a resulting state from s given I, D and the set of actions K.

$\diamondsuit$

Proof of Theorem 5.5.1: By Definition 52 the sequence $s_1, \ldots, s_n, s'$ is a sequence of possible resulting states iff there exists a sequence of evolving interpretations $M_0, M_1, \ldots M_n, s^*$ such that $M_0|_{\mathcal{F}} \equiv s$, $M_i|_{\mathcal{F}} \equiv s_i$ and $s^*|_{\mathcal{F}} \equiv s'$. Let $I_0, D_0, T_1 \ldots, T_n$ be the trace of $M_0, M_1, \ldots M_n, s^*$. By Definition 46 this amount to say that s^* is a refined stable model of the program

$$I_0, D_0, T_1 \ldots, T_n \cup E_n \tag{C.1}$$

We prove that the various T_i have one of these 3 forms:

1. $T_i = Aux_i$ where Aux_i is a set of literals of the form $Prev(Q)$ or $not\, Prev(Q)$ where Q is an inertial literal, or literals of the form $event(\tau)$ or $not\, event(\tau)$ and τ is the effect of some dynamic rules.
2. $T_i = Aux_i \cup$ **initialize($\mathcal{F}_\beta$)**.
3. $T_i = Aux_i \cup D_k$ for some k

Indeed, the set Aux_i is a set of auxiliary literals that are asserted. The auxiliary literals of the form $Prev(Q)$ and $not\ Prev(Q)$ are those that are asserted to guarantee that the truth value of the inertial fluent Q is preserved from one state to the next unless some static or dynamic rule changes it, while those of the form $event(\tau)$ or $not\ event(\tau)$ are asserted to specify that some **effect** τ has been triggered from a dynamic rule whose head is **effect**(τ).

If the input program E_i corresponding to M_i does not contain other rules except actions, the literals of the form Aux_{i+1} are the only rules appearing in T_{i+1} (case 1). If this is not the case then, either E_i contains the set of assertions $assert($**initialize(**$\mathcal{F}_\beta$**)**$)$ (case 2) or it contains the set of assertions $assert(D_k)$ (case 3). In case 2, the trace T_{i+1} also contains (above Aux_{i+1}) all the rules of **initialize(**$\mathcal{F}_\beta$**)** (and nothing else), in the case 3 it also contains all the rules of D_k (and nothing else).

By definition, there exists no rule with head $not\ assert(prev(Q))$ or $not\ assert(prev(not\ Q))$ in the D_k or I_k. Moreover, since every fluent only appears in an update $D_k s$ after being initialized, there exists no rule with head Q or $not\ Q$ in any T_j with $j < i$ where T_i is the T_i containing **initialize(**$\{Q\}$**)**. Hence, by Property 3.7.4, it is possible to move every $\mathcal{F}_\beta$ to the first program in the sequence without changing the semantics of the program C.1. Since $\bigcup \mathcal{F}_\beta = I$ we can simplify program C.1 to the following program

$$(I_0 \cup I), D_0, T_1' \ldots, T_n' \cup E_n \tag{C.2}$$

where T_i' is T_i pruned by any initialization rule.

We prove now that it is possible to ignore the various sets Aux_i thus obtaining a simplified program, i.e. the program C.2 is equivalent to the following DyLP:

$$(I_0 \cup I), D_0, D_1, \ldots, D_k \cup Aux_n \tag{C.3}$$

Since s_n is a two valued interpretation, then, for any inertial fluent Q, either $Q \in s_n$ or $not\ Q \in s_n$ and hence,by the rules in I, either $prev(Q) \in Aux_n$ or $not\ prev(Q) \in Aux_n$. Hence, by Corollary 3.7.1, it is possible to eliminate the facts of the form $prev(Q)$ and $not\ prev(Q)$ from any Aux_i with $i < n$ without changing the semantics of the program.

Let

$$\textbf{effect(}\tau\textbf{)} \leftarrow Cond.$$

be any dynamic rule in the updated EAP. By Section 5.3 the expression above is a macro for the set of rules

$$\begin{aligned} F &\leftarrow Body,\ event(\tau). \\ assert(event(\tau)) &\leftarrow Cond. \\ assert(not\ event(\tau) &\leftarrow event(\tau), not\ assert(event(\tau)). \end{aligned}$$

If either $event(\tau)$ or $not\ event(\tau)$ belongs to Aux_n, by Corollary 3.7.1 it is possible to remove both $event(\tau)$ and $not\ event(\tau)$ from any Aux_i with $i < n$ without changing the semantics of the program.

It remains to examine the case when neither $event(\tau)$ nor $not\ event(\tau)$ belong to Aux_n. Let Aux_j be the set of auxiliary literals containing either $event(\tau)$ or $not\ event(\tau)$ with the greatest index j. By the hypothesis above, $j < n$. We prove that $not\ event(\tau) \in\in Aux_j$.

If, by contradiction, $event(\tau) \in Aux_j$, then $event(\tau)$ belongs to M_j, hence, either $assert(event(\tau)) \in M_j$ or, by the rule

$$assert(not\ event(\tau) \leftarrow event(\tau), not\ assert(event(\tau))$$

$assert(not\ event(\tau)) \in M_j$ i.e. either

$$event(\tau) \in Aux_{j+1} \subseteq T_{j+1}$$

or

$$not\ event(\tau) \in Aux_{j+1} \subseteq T_{j+1}$$

thus contradicting the hypothesis of maximality of j.

Hence it can only be that $not\ event(\tau) \in Aux_j$. By Corollary 3.7.1 it is it is possible to remove both $event(\tau)$ and $not\ event(\tau)$ from any Aux_i with $i < j$ without changing the semantics of the program. Then, by Proposition 3.7.3, it is it is possible to remove $not\ event(\tau)$ from Aux_j without changing the semantics of the program.

Hence any set of literals of the form Aux_j can be removed without changing the semantics of the program.

Finally, by Proposition 3.7.5 all the empty $T_i s$ can be removed without changing the semantics of the program. Hence we end up concluding that we can simplify the trace of $M_0, M_1, \ldots M_n, s^*$ to the *DyLP*.

$$(I_0 \cup I), D_0, D_1, \ldots, D_k \cup Aux_n$$

as desired.

The set Aux_n consists of set $Prev(s_n)$, the set $ED(s_n, K)$ and the set $R(s_n)$ of literals of the form $not\ event(\tau)$ for each dynamic rule whose preconditions where true in s_{n-1} and false in s_n. The negative literals in $R(s_n)$ simply rejects facts of the form $event(\tau)$ from Aux_{n-1}. Since we have already simplified the trace by erasing all the $Aux_i s$ with $i < n$, by Proposition 3.7.3 we can remove the set $R(s_n)$ without changing the semantics of the program.

Thus, $s_1, \ldots s'$ is a sequence of possible resulting states iff s^* such that $s^*|_{\mathcal{F}} \equiv s'$ is a refined stable model of $I_0 \cup I, D_0, D_1, \ldots, D_k, ED(s_n, K) \cup Prev(s_n)$. This, by Definition 55, is equivalent to say that s' is a resulting state from s given $I_0 \cup I, D_0, D_1, \ldots, D_k$ and the set of actions K_{n+1}, as desired. $\diamondsuit$

Proof of Proposition 6.3.1: For any event e, by Definition 59:

$$\mathcal{S} = (\mathcal{P}, E_P, E_i.E_F) \vdash_e \Leftrightarrow \mathcal{P}' \vdash e$$

where $\mathcal{P}' = \mathcal{P}^R \uplus Ev(\mathcal{S}, i)$ and $\vdash$ is any of the inference relations $\vdash_{Se}$, $\vdash_\cap$ or $\vdash_{WF}$ defined as follows:

$$\begin{aligned} \mathcal{P}' \vdash_{Se} e &\Leftrightarrow e \in Se(\mathcal{RS}(\mathcal{P}')) \\ \mathcal{P}' \vdash_\cap e &\Leftrightarrow e \in M \,\forall\, M \in RS(\mathcal{P}') \\ \mathcal{P}' \vdash_{WF} e &\Leftrightarrow e \in WFDy(\mathcal{P}') \end{aligned}$$

where $\mathcal{RS}(\mathcal{P}')$ is the set of all the refined stable model of P' and $Se/1$ is a function that selects one of the refined model of $\mathcal{P}'$ (see Section 6.3.1).

These conditions, by definition of the refined and well founded semantics, are equivalent to, respectively:

$$\begin{aligned} e \in Se(\mathcal{RS}(\mathcal{P}')) &\wedge M = least\left(\rho\left(\mathcal{P}'\right) \setminus Rej^R\left(\mathcal{P}', M\right) \cup Def\left(\mathcal{P}', M\right)\right) \\ e \in M \,\forall\, M \in RS(\mathcal{P}') &\wedge M = least\left(\rho\left(\mathcal{P}'\right) \setminus Rej^R\left(\mathcal{P}', M\right) \cup Def\left(\mathcal{P}', M\right)\right) \\ e \in WFDy(\mathcal{P}') &\wedge W = least\left(\rho\left(\mathcal{P}'\right) \setminus Rej\left(\Gamma^S(W)\right) \cup Def\left(\Gamma^S(W)\right)\right) \end{aligned}$$

and, since the only rules whose heads are events belongs to $Ev(\mathcal{S}, i)$ and none of this rule is rejected [1], for any positive event e_p these conditions are equivalent to, respectively:

$$M = Se(\mathcal{RS}(\mathcal{P}')) \wedge \exists\, e_p \leftarrow B \in Ev(\mathcal{S}, i) : B \subseteq M \tag{C.4}$$

$$\forall\, M \in RS(\mathcal{P}') : \exists\, e_p \leftarrow B \in Ev(\mathcal{S}, i) : B \subseteq M \tag{C.5}$$

$$W = WFDy(\mathcal{P}') \wedge \exists\, e_p \leftarrow B \in Ev(\mathcal{S}, i) : B \subseteq W \tag{C.6}$$

and, for any negative event *not* e_p, since it may only belong to the set of default literals, these conditions are equivalent to, respectively:

$$M = Se(\mathcal{RS}(\mathcal{P}')) \wedge \nexists\, e_p \leftarrow B \in Ev(\mathcal{S}, i) : B \subseteq M \tag{C.7}$$

$$\forall\, M \in RS(\mathcal{P}') : \nexists\, e_p \leftarrow B \in Ev(\mathcal{S}, i) : B \subseteq M \tag{C.8}$$

$$W = WFDy(\mathcal{P}') \wedge \nexists\, e_p \leftarrow B \in Ev(\mathcal{S}, i) : B \subseteq \Gamma^S(W) \tag{C.9}$$

We are now ready to prove the statements $1-6$.

1. Let e_b be any basic event. By definition of $Ev(\mathcal{S}, i)$, the only rule whose head is e_b may be the fact e_b whose body is always satisfied and belongs to $Ev(\mathcal{S}, i)$ iff

[1] since there exists no rule whose head is the negation of an atom

$e_b \in E_i$. Hence the thesis follows by conditions C.4, C.5, C.6.

2. Given any complex event $e = e_1 \triangle e_2$, by conditions C.4, C.5, C.6, we have $\mathcal{S} \vdash_e e$ iff for $M = Se(\mathcal{RS}(\mathcal{P}'))$ (resp. for any refined model M of $\mathcal{P}'$, or for the well founded model W of $\mathcal{P}'$) there exists a rule $e \leftarrow B$ in $Ev(\mathcal{S}, i)$ such that $B \subseteq M$ (resp. $B \subseteq W$).

   The only rule in $Ev(\mathcal{S}, i)$ whose head is $e_1 \triangle e_2$ is:

   $$e_1 \triangle e_2 \leftarrow e_1, e_2.$$

   Then $\mathcal{S} \vdash_e e_1 \triangle e_2$ iff $e_1, e_2 \in M$ (resp. $e_1, e_2 \in W$), i.e. (by Definition 59) $\mathcal{S} \vdash_e e_1 \wedge \mathcal{S} \vdash_e e_2$.

3. Given any complex event $e = e_1 \bigtriangledown e_2$, by conditions C.4, C.5, C.6, we have $\mathcal{S} \vdash_e e$ iff for $M = Se(\mathcal{RS}(\mathcal{P}'))$ (resp. for any refined model M of $\mathcal{P}'$, or for the well founded model W of $\mathcal{P}'$) there exists a rule $e \leftarrow B$ in $Ev(\mathcal{S}, i)$ such that $B \subseteq M$ (resp. $B \subseteq W$).

   The only rules whose head is $e_1 \bigtriangledown e_2$ are:

   $$e_1 \bigtriangledown e_2 \leftarrow e_1. \qquad e_1 \bigtriangledown e_2 \leftarrow e_2.$$

   Hence $\mathcal{S} \vdash_e e_1 \bigtriangledown e_2$ iff either $e_1 \in M$ (resp. $e_1 \in W$) or $e_2 \in M$ (resp. $e_2 \in W$), i.e. (by Definition 59) either $\mathcal{S} \vdash_e e_1$ or $\mathcal{S} \vdash_e e_2$.

4. Given the event name e_{nam}, by conditions C.4, C.5, C.6, we have $\mathcal{S} \vdash_e e_{nam}$ iff for $M = Se(\mathcal{RS}(\mathcal{P}'))$ (resp. for any refined model M of $\mathcal{P}'$, or for the well founded model W of $\mathcal{P}'$) there exists a rule $e_{nam} \leftarrow B$ in $Ev(\mathcal{S}, i)$ such that $B \subseteq M$ (resp. $B \subseteq W$). The only rules whose head is e_{nam} are those of the form:

   $$e_{nam} \leftarrow e. \qquad \forall\, e_{nam} \textbf{ is } e \quad \in \mathcal{S}$$

   Hence, $\mathcal{S} \vdash_e e_{nam}$ iff, for some e such that

   $$e_{nam} \textbf{ is } e \quad \in \mathcal{S}$$

   $e \in M$ (resp. $e \in W$), i.e. (by Definition 59) $\mathcal{S} \vdash_e e$.

5. Given the event name $e = A(e_1, e_2, e_3)$, by conditions C.4, C.5, C.6, we have $\mathcal{S} \vdash_e e$ iff for $M = Se(\mathcal{RS}(\mathcal{P}'))$ (resp. for any refined model M of $\mathcal{P}'$, or for the well founded model W of $\mathcal{P}'$) there exists a rule $A(e_1, e_2, e_3) \leftarrow B$ in $Ev(\mathcal{S}, i)$ such that $B \subseteq M$ (resp. $B \subseteq W$).

The only rule whose head is e may be one the form:

$$e = A(e_1, e_2, e_3) \leftarrow e_2.$$

Moreover, by the definition of $Ev(\mathcal{S}, i)$ and $AR(\mathcal{S}, i)$, the rule above belongs to $Ev(\mathcal{S}, i)$ iff

$$\exists\, m < i \; s.t. \;\; \mathcal{S}^m \;\vdash_e e_1 \wedge \; \forall j \; s.t. \; m < j \leq i : \;\; \mathcal{S}^j \;\;\nvdash_e e_3$$

Hence, $\mathcal{S} \;\vdash_e A(e_1, e_2, e_3)$ iff $e_2 \in M$ (resp. $e_2 \in W$), i.e. (by Definition 59) $\mathcal{S} \;\vdash_e e_2$ and the condition above is satisfied.

6. Let $not\, e_p$ be any negative event. If M is a refined model of $\mathcal{P}'$, since M is two valued, then $not\, e_p \in M$ iff $e_p \notin M$.

   If the chosen inference relation is $\vdash_{Se}$, $\mathcal{S} \vdash_{Se\ e} not\, e_p$ iff $not\, e_p \in Se(\mathcal{RS}(\mathcal{P}'))$ and this condition is satisfied iff $e_p \notin M$, i.e. (by Definition 59) $\mathcal{S} \;\nvdash_e e_p$ as desired.

   If the chosen inference relation is $\vdash_{\cap}$ and $\mathcal{S} \vdash_{\cap\ e} not\, e_p$, then, for any refined stable model M of $\mathcal{P}'$, $not\, e_p \in M$, i.e. $e_p \notin M$ and hence $\mathcal{S} \nvdash_{\cap\ e} e_p$ immediately follows.

   If, $\mathcal{S} \nvdash_{\cap\ e} e_p$ we proceed by induction on the structure of e_p to prove that

   $$\mathcal{S} \vdash_{\cap\ e} not\, e_p.$$

   If e_p is a basic event, we already proved in (1) that $\mathcal{S} \nvdash_{\cap\ e} e_p$ implies $e_p \notin E_i$. Hence there exists no rule in $Ev(\mathcal{S}, i)$ whose head is e_b and hence $not\, e_p$ belongs to $Def(\mathcal{P}', M)$ for any refined model M, and hence:

   $$not\, e_p \in M = least\left(\rho\left(\mathcal{P}'\right) \setminus Rej^R\left(\mathcal{P}', M\right) \cup Def\left(\mathcal{P}', M\right)\right)$$

   Which is equivalent to $\mathcal{S} \vdash_{\cap\ e} not\, e_p$.

   If e_p is a complex event and $\mathcal{S} \nvdash_{\cap\ e} e_p$ then, by condition C.5, there exists at least one refined model M of P' such that $e_p \notin M$, i.e. for any rule $e \leftarrow B \in Ev(\mathcal{S}, i)$ there exists some literal B_i with $B_i \in B$ such that $B_i \notin M$, i.e. $\mathcal{S} \nvdash_{\cap\ e} B_i$. By definition of $Ev(\mathcal{S}, i)$, the various B_i are simpler events than e_p hence, by inductive hypothesis, $\mathcal{S} \vdash_{\cap\ e} not\, B_i$, i.e. $not\, B_i$ belongs to any refined model M of P'. Hence the body of the rule B is not the subset of any refined model M. This holds for any rule of the form $e \leftarrow B$. By condition C.8 this implies $\mathcal{S} \vdash_{\cap\ e} not\, e_p$ as desired.

   It remains examine the inference relation $\vdash_{WF}$. We remember that, for any event e, $\mathcal{S} \vdash_{WF\ e} e$ iff $e \in W$ where W is the well founded model of $\mathcal{P}'$. We proceed again by induction on the structure of e_p.

   If e_p is a basic event, then by (1), $\mathcal{S} \nvdash_{WF\ e} e_p$ iff $e_p \notin E_i$ which, in turn, is true iff $e_p \in P'$. Since there is no other rule in P' whose head is e_p, then $e_p \in P'$ iff $not\, e_p \in Def\left(\Gamma^{\mathcal{S}}(W)\right)$ which, since there is no rule in P' whose head is $not\, e_p$, is

equivalent to

$$not\, e_p \in W = least\left(\rho\left(\mathcal{P}'\right) \setminus Rej\left(\Gamma^S(W)\right) \cup Def\left(\Gamma^S(W)\right)\right)$$

i.e. $\mathcal{S} \vdash_{WF\ e} not\, e_p$. So we conclude $\mathcal{S} \nvdash_{WF\ e} e_p$ iff $\mathcal{S} \vdash_{WF\ e} not\, e_p$.

If e_p is a complex event, by condition, C.9 $\mathcal{S} \vdash_{WF\ e} not\, e$ is equivalent to say:

$$\forall\ e_p \leftarrow B\ \in Ev(\mathcal{S}, i) :\ \exists B_i \in B\ s.t.\ \ B_i \notin \Gamma^S(W)$$

By Proposition 4.5.4, this is equivalent to:

$$\forall\ e_p \leftarrow B\ \in Ev(\mathcal{S}, i) : \exists B_i \in B\ s.t.\ \ not\, B_i \in W$$

By inductive hypothesis, this is equivalent to:

$$\forall\ e_p \leftarrow B\ \in Ev(\mathcal{S}, i) : \exists B_i \in B\ s.t.\ \ B_i \notin W$$

or, equivalently:

$$\nexists\ e_p \leftarrow B\ \in Ev(\mathcal{S}, i) :\ B \subseteq W$$

which is the negation of condition C.9 i.e. it is a sufficient and necessary condition for $\mathcal{S} \nvdash_{WF\ e} e_p$ as desired.

$\diamondsuit$