INTEGRATION OF ONTOLOGIES WITH PROGRAMS BASED ON RULES

Rita Sofia Moreira Henriques

DISSERTAÇÃO

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Engenharia de Software

2013
INTEGRATION OF ONTOLOGIES WITH PROGRAMS BASED ON RULES

Rita Sofia Moreira Henriques

DISSERTAÇÃO

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Engenharia de Software

Dissertação orientada pela Professora Doutora Maria Isabel Alves Batalha Reis da Gama Nunes e co-orientada pelo Professor Luís Calhorda Cruz-Filipe

2013
Acknowledgments

Completing this thesis and the corresponding master degree would not have been possible without the contribution of all the people that accompanied me in this journey.

First, I would like to thank both my supervisors: Prof. Isabel Nunes for guiding me in this odyssey, giving me useful comments and encouragement along the way; Prof. Luís Cruz-Filipe for teaching me how to program from scratch in the first year of college when part of my world collapsed, for his scientific discipline and rigour, interesting conversations and a great deal of patience.

A special thanks to the other members of the ITSWeb group, Prof. Graça Gaspar, Prof. Patrícia Engrácia and particularly to Daniel Santos, who worked closely with me.

To Thomas Krennwallner for always being available to reply to questions about the dlvhex tool and the DL-plugin.

I would like to thank João Martins for his love and support, for distracting me from my work when I needed, for all the patience, as well as the encouragement to do more and better – without you, I could not imagine to get this far.

To my sister Joana I want to thank all the good times that we have together and the concern for my academic life.

To my mother I want to thank the wonderful 18 years I spent with her, the education and moral values she gave me – I hope you are proud of me and I would have enjoyed that you were here to tell me this.

I would also like to thank my father and the rest of the family for all the support throughout my life.

Last, but not least, I would like to thank all my friends. Everything I have done in my life would have been impossible without their support. I would like to thank Juliana Franco for all the nights we spent doing group works and everything else; Carlos Barata for always being available to talk about some stuff, for fun and for everything; Tiago Aparício for being always ready for fun and for always having a smile on his face. I would also like to thank José Carilho, Fábio Santos, Inês Roque, and to all the people of the Laboratory of Agent Modelling (LabMAg).
Para a minha querida Mãe
Resumo

A combinação de lógicas de descrição e programas baseados em regras tem sido bastante estudada nos últimos anos.

As lógicas de descrição são uma família de linguagens formais que servem para representar conhecimento. Estas são bastante usadas na Web Semântica para exprimir ontologias, como o OWL (Web Ontology Language), ou bases de conhecimento. As lógicas de descrição são utilizadas para representar conceitos e as relações entre eles num dado domínio, e também para raciocinar sobre os mesmos.

Diversos sistemas foram propostos, de onde se destacam os programas da lógica de descrição ou programas-dl. Ao longo destes útimos anos, estes programas têm ganho força na comunidade da Web Semântica. A sintaxe dos programas-dl facilita a interação entre uma ontologia, expressa em lógica de descrição, e um programa baseado em regras, que pode conter átomos-DL. Este tipo de átomos é utilizado para construir as chamadas regras-DL. A interação entre estes dois componentes é conseguida através destas regras que permitem fazer consultas à base de conhecimento, possibilitando ainda a extensão da base de conhecimento com factos do programa em lógica antes da consulta ser feita. Esta extensão da base de conhecimento é apenas local a esta consulta, não tendo, por isso, um efeito global na ontologia. Desta forma é possível enriquecer o programa em lógica original com conhecimento proveniente da ontologia.

A definição original de um programa-dl foi estendida com a capacidade de combinar várias bases de conhecimento em lógica de descrição. Nestes Multi-programas-DL (programas-Mdl), um programa em lógica representa o “condutor” que “coordenha” as bases de conhecimento, que são completamente independentes umas das outras, podendo estar fisicamente separadas ou serem geridas independentemente. Em particular, um programa-dl pode ser visto como um programa-Mdl com apenas uma ontologia. Desta forma, podemos dizer que os programas-Mdl generalizam a definição dos programas-dl. Nesta dissertação podemos ver que uma das principais vantagens dos programas-Mdl é a sua simplicidade. Desta forma, estes programas podem ser extremamente adequados para raciocinar na Web Semântica, onde uma grande parte do esforço é colocada em desenvolver ontologias reutilizáveis.

Nesta dissertação mostramos como uma implementação de programas-dl já existente,
o DL-plugin para a ferramenta dlvhex, pode ser estendida para trabalhar com programas-Mdl, onde se consegue raciocinar com mais que uma ontologia.

Os programas-Mdl podem ser estendidos com novas construções sintáticas, os observadores, que permitem estender conceitos ou relações da base de conhecimento (do ponto de vista do programa sobre aquela base de conhecimento) com todas as instâncias de um predicado do programa em lógica, e reciprocamente.

Esta construção sintática também foi implementada no DL-plugin como anotações no programa em lógica. Estas anotações são processadas por um novo módulo, traduzindo, assim, um programa-Mdl com observadores num programa-Mdl normal.

Nesta dissertação fornecemos uma análise de performance onde podemos concluir que um programa-Mdl com observadores tem praticamente a mesma performance que um programa-Mdl similar. Programas-Mdl com observadores têm a vantagem de serem mais pequenos e simples. Estes programas são mais robustos em relação a futuras mudanças, pois com observadores é garantido que cada extensão à ontologia seja feita de forma adequada; sem observadores, isto teria que ser garantido à mão.

Esta dissertação providencia vários casos de estudo que foram utilizados para ilustrar em detalhe as novas construções. Estes casos de estudo utilizam ontologias reais disponíveis livremente na internet.

**Palavras-chave:** Lógicas de descrição, programas-dl, ontologias, semântica de conjunto-resposta
Abstract

The combination of description logics and rule-based reasoning systems has been widely studied in last years, with the proposal of several different systems that achieve this goal.

Description logic programs (dl-programs) were introduced a few years ago as a mechanism to combine a description logic knowledge base with a logic program that can access and dynamically change its view of the knowledge base.

The original definition of a dl-program was later extended with the capability of combining several description logic knowledge bases. In Multi description logic programs (Mdl-programs) the logic programming represents the “conductor” that “coordinates” the several knowledge bases, which can be physically separated or independently maintained.

In this dissertation we show how a current implementation of dl-programs, the DL-plugin for dlvhex, can be extended to work with Mdl-programs, where one can work with more than one ontology, keeping them completely separated.

Mdl-programs can be extended by new syntactic constructions – observers – allowing to extend concepts or roles from a knowledge base (in the program’s view of that knowledge base) automatically with all instances of a predicate in the logic program or reciprocally. This syntactic construction also was implemented in the DL-plugin as annotations in the logic program.

We provide a performance analysis from which we can conclude that an Mdl-program with observers has practically the same performance as a similar Mdl-program. One Mdl-program with observers is shorter and more legible just because all global observations are clearly marked.

This dissertation also provides some case-studies to illustrate the constructions detailed above, using real-life ontologies freely available on the internet.

Keywords: Description logics, dl-programs, ontologies, answer-set semantics
# Contents

List of Figures xiii

List of Tables xv

1 Introduction 1

1.1 Motivation .......................... 1
1.2 Context .................................. 2
1.3 Goals .................................. 2
1.4 Document structure ................. 2

2 Background 5

2.1 Description Logics .......................... 5
  2.1.1 Syntax and Semantics ......................... 7
  2.1.2 Web Ontology Language (OWL) ............... 9
2.2 Description Logic Programs ............... 11
  2.2.1 Syntax of dl-programs ..................... 11
  2.2.2 Semantics of dl-programs ................. 15
2.3 Tools ................................ 18
  2.3.1 NLP-DL ................................ 18
  2.3.2 dlvhex ............................... 19

3 The DL-plugin for dlvhex 23

3.1 Writing dl-programs in the DL-plugin .... 23
3.2 DL-plugin atoms .......................... 24
3.3 Overview of dlvhex and the DL-plugin .... 29
  3.3.1 DL-plugin use cases ...................... 31
  3.3.2 Interaction between dlvhex and the DL-plugin 32

4 Multi Description Logic programs (Mdl-programs) 35

4.1 Why Mdl-programs ...................... 35
4.2 Syntax and semantics .................... 36
4.3 Adding observers .......................... 39
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>An example network</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>NLP-DL Web prototype</td>
<td>19</td>
</tr>
<tr>
<td>3.1</td>
<td>Brief overview of the DL-plugin with its components, their relationships and information flow</td>
<td>30</td>
</tr>
<tr>
<td>3.2</td>
<td>Use case diagram of the DL-plugin</td>
<td>31</td>
</tr>
<tr>
<td>5.1</td>
<td>Brief overview of the DL-plugin with the new OBSERVER CONVERTER module</td>
<td>47</td>
</tr>
<tr>
<td>5.2</td>
<td>Brief overview of DL-plugin with the new DOMAIN CHECKER module</td>
<td>53</td>
</tr>
<tr>
<td>5.3</td>
<td>Performance analysis: Mdl-programs vs Mdl-programs with observers (with or without DOMAIN)</td>
<td>57</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Description Logic concept constructors. ........................................ 7
2.2 Description Logic roles constructors. ......................................... 8
2.3 Terminological and assertional axioms. ....................................... 8

B.1 OWL DL syntax vs. DL syntax and semantics ............................... 65
B.2 OWL DL axioms and facts .......................................................... 66

C.1 Mdl-program and Mdl-program with observers experiment results (time in seconds) ......................................................... 69
Chapter 1

Introduction

This thesis is on the subject of combining logic programs and description logic knowledge bases. This first chapter presents our motivation in this subject. In the second section, we describe the context in which this work is included. In the third section, we present the main goal of this dissertation. Finally, we provide an overview of the thesis’ structure.

1.1 Motivation

The combination of logic programs and description logic knowledge bases has been widely studied in the last years, with several approaches to achieve this goal proposed for semantic web reasoning, e.g. description logic programs \[10,11\], multi-context systems \[4\] and HEX-programs \[12\], among others.

Ontologies are expressed through decidable fragments of function-free first-order logic with equality, offering a very good ratio expressiveness/complexity of reasoning \[2\]. The addition of some kind of rule capability in order to be able to express more powerful queries together with non-monotonic features (in particular, the negation-as-failure operator not) is achieved by joining ontologies and logic programming, resulting in a very powerful framework for semantic web reasoning. Moreover, in the last few years, there has been a major development of free-access ontologies over the internet. In this way, these knowledge bases can be used for semantic web reasoning and reused in different reasoning contexts. However, the combination of information from different sources brings a set of practical problems, especially in terms of sharing and consistency.

Description logic programs, dl-programs for short, are an already well established framework for coupling description logic knowledge bases with rule-based reasoning. Although they are an interesting framework, dl-programs are not expressive enough for many practical purposes. Can we modify dl-programs to make them more powerful or flexible, and what could be changed or added to make them more versatile?
1.2 Context

This project falls within the scope of the work of the ITSWeb group, composed of two researchers – Prof. Isabel Nunes and Prof. Graça Gaspar – from Departamento de Informática da Faculdade de Ciências da Universidade de Lisboa –, two researchers – Prof. Luís Cruz-Filipe and Prof. Patrícia Engrácia – from another institution, Escola Superior Náutica Infante D. Henrique – and one undergraduate student – Daniel Santos – in Applied Mathematics from Faculdade de Ciências da Universidade de Lisboa.

The research directions within ITSWeb lie in understanding and developing languages and tools for the semantic web. The group has three main working directions: dl-programs and their variants, integration with other programming paradigms, and integrity problems.

There was a need to put into practice all the complex work already developed by the group this far. In this way, my role on ITSWeb was to bring a practical view, implementing some of the new syntactic constructions proposed by the group. By implementing these constructions which were only theoretically illustrated, we can check that they truly work, and we can evaluate and compare with similar constructions. Also, it is important for ITSWeb to have an available tool, where the case-studies can be tested and new constructions can be implemented.

The work took place in the facilities of LabMAg, a multi-disciplinary research unit.

1.3 Goals

The main goals of this work are:

- to explore available tools that combine logic programs with description logic knowledge bases, from a practical perspective;
- to choose a tool to extend with the new syntactic constructions defined by the ITSWeb group;
- to implement these new constructions, to test the implementation with some new case-studies and to analyse the performance; and
- to solve technical issues that may appear during the implementation or the testing phase.

1.4 Document structure

This document is organized as follows:

- Chapter 2 (Background) presents the background. It is divided in three main sections. In the first section we present Description Logics, a formal basis for ontology
languages, that have been used to express ontologies in the Semantic Web, as the Web Ontology Language (OWL). This section also presents the syntax and the semantics of description logics. These are a component of description logic programs (dl-programs), which combine ontologies, based on description logics, with logic programs. Finally we provide a survey of tools that support working with dl-programs and we explain why the DL-plugin for dlvhex was the chosen tool to extend.

• Chapter 3 (The DL-plugin for dlvhex) describes the DL-plugin for the dlvhex tool. We show how to write a dl-program for this plugin, regarding the concrete syntax for a dl-atom. The DL-plugin converts a dl-program to an HEX-program using some external atoms provided by the plugin, described in Section 3.2. Due to the lack of documentation of these tools, it was necessary to perform reverse engineering, analysing both components. This chapter summarizes the information obtained from this process.

• Chapter 4 (Multi Description Logic programs (Mdl-programs)) presents Mdl-programs, a generalization of dl-programs, proposed by the ITSWeb group. Mdl-programs allow working with several description logic knowledge bases, while keeping them separate. In the first section of this chapter, we explain the motivation to define Mdl-programs. This chapter also introduces a useful syntactic construction for Mdl-programs – the observers – allowing one to extend (globally) concepts or roles with instances of a predicate in the program, and reciprocally.

• Chapter 5 (Mdl-programs in the DL-plugin for dlvhex) describes how Mdl-programs were implemented in the DL-plugin for dlvhex. This extension allows the use of multiple ontologies and provides syntactical support for observers. The first two sections present the implementation details. The following section describes some technical issues that arose during the implementation and the solutions adopted. We also have integrated an addition mechanism for the original dl-programs – lifting, which achieves a complete two-way integration between a knowledge base and a program. In the remainder of this chapter we compare the performance of an Mdl-program and an Mdl-program with observers.

• Chapter 6 (Conclusions) presents a summary of the developed work, as well as our contributions.
Chapter 2

Background

This chapter introduces description logics as a formal basis for ontology languages, in particular the Web Ontology Language (OWL), presenting the syntax and the semantics of description logics. Then, description logic programs (dl-programs) are presented as one approach of combining ontologies with logic programs under answer-set semantics. The last section of this chapter provides a survey of tools for dl-programs, describing each one and explaining why the DL-plugin for dlvhex was the chosen tool to extend with Mdl-programs.

2.1 Description Logics

Description logics (DLs) are decidable fragments of first-order logic (FOL), or extension thereof, that have been extensively used to express ontologies in the Semantic Web.

The term “Description Logics” has its roots in the 1970s [2], when two distinct trends appeared in knowledge representation: logic-based formalisms, which use predicate calculus to draw implicit conclusions from explicitly represented knowledge, and non-logic-based representations, which build on cognitive notions. For example, network structures and rule-base representations were built deriving from experiments on recall from human memory and human execution of tasks as mathematical puzzle solving. This approach led to development of frames and semantic networks, where a network structure represents sets of individuals and their relationships.

One important step was the recognition that frames can be given a semantics that is based on first-order logic, by unary predicates representing sets of individuals and binary predicates representing relationships between them [17]. From this, it turns out that semantic networks and frames require only a fragment of first-order predicate logic, instead of relying on full first-order theorem provers. This way, specialized reasoning techniques were suitable to carry out reasoning in these formalisms.

In order to provide some intuition about description logics, Figure 2.1 shows a network representation, where the elements of the network are nodes and links. Nodes denote
Chapter 2. Background

Figure 2.1: An example network.

concepts, i.e., classes of individuals, and links characterize relationships among them. The figure represents knowledge concerning persons, students, professors, etc. Its structure is referred as a terminology. A specific type of link is the “IS-A” relationship, which defines a hierarchy between concepts, like the link between Student or Professor and Person. A characteristic of Description Logics is the ability to represent other kinds of properties, namely binary relationships, which are called “roles”. The figure shows that the concept Student has a property labelled attends, which relates this concept with Course. This role has a “value restriction”, which expresses a limitation on the range of types that can fill that role. For example, the role hasSupervisor with a 1 as value-restriction means that “each MScStudent must have exactly one supervisor”. Relationships of this kind are inherited from concepts to their subconcepts.

A DL models concepts, roles and individuals, and their relationships. The fundamental modelling concept of a DL is the axiom, which is a logical statement relating roles and/or concepts. Description logics use different terminology than the first-order logic. For example, in FOL we have classes, predicates or properties and objects, whereas DLs have concepts, roles and individuals, respectively. The Ontology Web Language (OWL) uses FOL terminology, in spite of being an implementation of a description logic.

There are a variety of different description logics formalisms, each with specific complexity and offering a distinct set of language constructs. $\mathcal{SHOIN}(D)$ and $\mathcal{SHIF}(D)$ are two of them, which are the underpinning of Semantic Web ontology languages and the basis of the dl-programs, that will be presented in Section 2.2.

The naming scheme of specific description logics corresponds to the constructors they allow, in addition to the basic ones like concept union, concept intersection, etc. $\mathcal{SHOIN}(D)$ provides additional operators for role transitivity ($S$), role hierarchy ($H$),
nominals or “one-of” constructor (\(O\)), role inverses (\(I\)), unqualified number restrictions (\(N\)), and datatypes (\(D\)). The \(SHIF(D)\) DL is less expressive, with \(F\) standing for functionality, which is a restricted form of number restriction (\(\leq 1R\)). \(F\) is subsumed by \(N\) of \(SHOIN(D)\). Therefore, \(SHIF(D)\) is a restriction of \(SHOIN(D)\). These description logics are the formal counterparts of languages of the Web Ontology Language OWL, which will be discussed in Section 2.1.2.

2.1.1 Syntax and Semantics

First we introduce some notational conventions: \(A\) and \(B\) represent atomic concepts, \(C\) and \(D\) are concept descriptions. For roles we use \(R\) and \(S\); in number restrictions, non-negative integers are denoted by \(n\) and \(m\), and individuals by \(a\) and \(b\).

Atomic concepts and atomic roles are elementary descriptions. Complex ones can be built from atomic ones inductively with concept constructors (Table 2.1) and role constructors (Table 2.2).

An interpretation \(I = (\Delta^I, \cdot^I)\) consists of a non-empty set \(\Delta^I\) (the domain of the interpretation) and an interpretation function \(\cdot^I\), which assigns to every atomic concept \(A\) a set \(A^I \subseteq \Delta^I\) and to every atomic role \(R\) a binary relation \(R^I \subseteq \Delta^I \times \Delta^I\).

Concept constructors take concept and/or role descriptions and transform them into more complex concept descriptions. Table 2.1 shows the syntax and semantics of common concept constructors.

<table>
<thead>
<tr>
<th>Name</th>
<th>Syntax</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Universal concept</td>
<td>⊤</td>
<td>(\Delta^I)</td>
</tr>
<tr>
<td>Bottom concept</td>
<td>⊥</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>Negation</td>
<td>(\neg C)</td>
<td>(\Delta^I \setminus C^I)</td>
</tr>
<tr>
<td>Intersection</td>
<td>(C \cap D)</td>
<td>(C^I \cap D^I)</td>
</tr>
<tr>
<td>Value restriction</td>
<td>(\forall R.C)</td>
<td>({a \in \Delta^I \mid \forall b.((a, b) \in R^I \rightarrow b \in C^I)})</td>
</tr>
<tr>
<td>Existential quantification</td>
<td>(\exists R.C)</td>
<td>({a \in \Delta^I \mid \exists b.((a, b) \in R^I \land b \in C^I)})</td>
</tr>
<tr>
<td>Limited existential quantification</td>
<td>(\exists R.\top)</td>
<td>({a \in \Delta^I \mid \exists b.(a, b) \in R^I})</td>
</tr>
<tr>
<td>Unqualified number restriction</td>
<td>(\geq n R)</td>
<td>({a \in \Delta^I \mid {b \in \Delta^I \mid (a, b) \in R^I}\geq n})</td>
</tr>
<tr>
<td></td>
<td>(\leq n R)</td>
<td>({a \in \Delta^I \mid {b \in \Delta^I \mid (a, b) \in R^I}\leq n})</td>
</tr>
<tr>
<td></td>
<td>= n R</td>
<td>({a \in \Delta^I \mid {b \in \Delta^I \mid (a, b) \in R^I} = n})</td>
</tr>
<tr>
<td>Qualified number restriction</td>
<td>(\geq n R.C)</td>
<td>({a \in \Delta^I \mid {b \in \Delta^I \mid (a, b) \in R^I \land b \in C^I}\geq n})</td>
</tr>
<tr>
<td></td>
<td>(\leq n R.C)</td>
<td>({a \in \Delta^I \mid {b \in \Delta^I \mid (a, b) \in R^I \land b \in C^I}\leq n})</td>
</tr>
<tr>
<td></td>
<td>= n R.C</td>
<td>({a \in \Delta^I \mid {b \in \Delta^I \mid (a, b) \in R^I \land b \in C^I} = n})</td>
</tr>
</tbody>
</table>

Table 2.1: Description Logic concept constructors.

Role constructors take role and/or concept descriptions and transform them into more
complex role descriptions. Table 2.2 shows the syntax and semantics of common role constructors. The last two constructors are not first-order, so description logics allowing these operations are no longer fragments of first-order logic.

<table>
<thead>
<tr>
<th>Name</th>
<th>Syntax</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Universal role</td>
<td>$U$</td>
<td>$\Delta^I \times \Delta^I$</td>
</tr>
<tr>
<td>Intersection</td>
<td>$R \cap S$</td>
<td>$R^I \cap S^I$</td>
</tr>
<tr>
<td>Union</td>
<td>$R \cup S$</td>
<td>$R^I \cup S^I$</td>
</tr>
<tr>
<td>Complement</td>
<td>$\neg R$</td>
<td>$(\Delta^I \times \Delta^I) \setminus R^I$</td>
</tr>
<tr>
<td>Inverse</td>
<td>$R^{-}$</td>
<td>${ (b,a) \in \Delta^I \times \Delta^I \mid (a,b) \in R^I }$</td>
</tr>
<tr>
<td>Composition</td>
<td>$R \circ S$</td>
<td>${ (a,c) \mid \exists b. (a,b) \in R^I \land (b,c) \in S^I }$</td>
</tr>
<tr>
<td>Transitive closure</td>
<td>$R^+$</td>
<td>$\bigcup_{n \geq 1} (R^I)^n$</td>
</tr>
<tr>
<td>Reflexive-transitive closure</td>
<td>$R^*$</td>
<td>$\bigcup_{n \geq 0} (R^I)^n$</td>
</tr>
</tbody>
</table>

Table 2.2: Description Logic roles constructors.

A description logic knowledge base consists of a set of terminological axioms (often called a TBox) and a set of assertional axioms or assertions (often called an ABox). Table 2.3 shows the syntax and semantics of these axioms. An equality whose left side is an atomic concept (resp. role) is called a concept (resp. role) definition. A finite set of definitions is a terminology if they are unambiguous, that is, no atomic concept occurs more than once on the left side. Axioms of the form $C \sqsubseteq D$ for a complex description $C$ are called general inclusion axioms. A set of axioms of the form $R \sqsubseteq S$ where both $R$ and $S$ are atomic is called a role hierarchy. Such a hierarchy imposes restrictions on the interpretation of roles.

<table>
<thead>
<tr>
<th>Name</th>
<th>Syntax</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concept inclusion</td>
<td>$C \sqsubseteq D$</td>
<td>$C^I \subseteq D^I$</td>
</tr>
<tr>
<td>Role inclusion</td>
<td>$R \sqsubseteq S$</td>
<td>$R^I \subseteq S^I$</td>
</tr>
<tr>
<td>Concept equality</td>
<td>$C \equiv D$</td>
<td>$C^I = D^I$</td>
</tr>
<tr>
<td>Role equality</td>
<td>$R \equiv S$</td>
<td>$R^I = S^I$</td>
</tr>
<tr>
<td>Concept assertion</td>
<td>$C(a)$</td>
<td>$a^I \in C^I$</td>
</tr>
<tr>
<td>Role assertion</td>
<td>$R(a,b)$</td>
<td>$(a^I, b^I) \in R^I$</td>
</tr>
</tbody>
</table>

Table 2.3: Terminological and assertional axioms.

The satisfaction of a description logic axiom $F$ in the interpretation $\mathcal{I} = (\Delta^I, \cdot^I)$, denoted $\mathcal{I} \models F$, is defined with the help of Table 2.3. For example, $\mathcal{I} \models C \sqsubseteq D$ iff $C^I \subseteq D^I$.

An interpretation $\mathcal{I}$ satisfies an axiom $F$, or $\mathcal{I}$ is a model of $F$, iff $\mathcal{I} \models F$. The interpretation $\mathcal{I}$ satisfies a knowledge base $L$, or $\mathcal{I}$ is a model of $L$, denoted $\mathcal{I} \models L$, iff
$I \models F$ for all $F \in L$. We say that $L$ is satisfiable (resp. unsatisfiable), iff $L$ has a model (resp. no model). An axiom $F$ is a logical consequence of $L$, denoted $L \models F$, iff every model of $L$ satisfies $F$. A negated axiom $\neg F$ is a logical consequence of $L$, denoted $L \models \neg F$, iff every model of $L$ does not satisfy $F$.

A small example is shown below, illustrating simple ideas about description logics.

**Example 1.** Consider the following description logic knowledge base $L$ with simple knowledge about parenthood relationships:

$\text{Child} \subseteq (\exists \text{hasFather.} \text{Father}) \cap (\exists \text{hasMother.} \text{Mother})$ \hfill (1)

$\text{hasFather} \subseteq \text{hasParent}$ \hfill (2)

$\top \subseteq \leq 1 \text{hasFather}$ \hfill (3)

$\text{hasMother} \subseteq \text{hasParent}$ \hfill (4)

$\text{hasMother} \equiv \text{isMotherOf}^-$ \hfill (5)

$\top \subseteq \leq 1 \text{isMotherOf}^-$ \hfill (6)

$\text{Child}(\text{Katty})$ \hfill (7)

$\text{hasFather}(\text{Katty}, \text{John})$ \hfill (8)

$\text{isMotherOf}(\text{Susan}, \text{Katty})$ \hfill (9)

The axioms (1–6) form the TBox of $L$; the assertions (7–9) are the ABox of $L$.

The axiom (1) says that all children have some father (of concept $\text{Father}$) and also some mother (of concept $\text{Mother}$). The role hierarchy is achieved by the axioms (2) and (4), where $\text{hasFather}$ and $\text{hasMother}$ are subroles of $\text{hasParent}$. The axiom (3) indicates that $\text{hasFather}$ has at most one value (0 or 1), for each individual. The axiom (5) declares that $\text{hasMother}$ has an inverse role called $\text{isMotherOf}^-$ and the axiom (6) indicates that the inverse role of $\text{isMotherOf}^-$, that is $\text{hasMother}$, has at most one value for each individual. Assertions (7–9) declares that Katty is a child, John is her father and Susan is her mother.

We now can reason about this family knowledge. We can infer that John is a father, i.e., $L \models \text{Father}(\text{John})$ and also that Susan is a mother, i.e., $L \models \text{Mother}(\text{Susan})$. Furthermore, we can infer $\text{hasParent}(\text{Katty}, \text{Susan})$, $\text{hasParent}(\text{Katty}, \text{John})$ and also $\text{hasMother}(\text{Katty}, \text{Susan})$.

### 2.1.2 Web Ontology Language (OWL)

Ontologies play an important role in the Semantic Web. They define the concepts and relationships in a certain domain. The reuse of ontologies is a key technology for the feasibility of the Semantic Web. Also, ontologies are considered a convenient tool for specifying knowledge in interdisciplinary sciences.

The **Web Ontology Language (OWL)** \[3\] was developed by the World Wide Web Consortium (W3C) Web Ontology Working Group. This language emerged from its
predecessors *Simple HTML Ontology Extensions* (SHOE) [13], an extension of HTML with semantic markup to represent ontologies in hypertext documents, and DAML+OIL [24], an RDF-Schema based ontology language and itself was a combination of *Ontology Inference Layer* (OIL) [13] and *DARPA Agent Modelling Language* (DAML) [19]. One design goal for OWL was to maintain as much compatibility to these preceding formalisms as possible. OWL became a W3C Recommendation in February 2004 and is considered the standard language for specifying ontologies on Semantic Web.

OWL consists of three sublanguages with increasing expressivity: OWL Lite, OWL DL and OWL Full. OWL Lite and OWL DL semantics are based on description logics, namely, $SH_{IF}(D)$ and $SH_{OIN}(D)$, respectively. OWL Full does not have some specific restrictions of OWL Lite and OWL DL, so reasoning in OWL Full is undecidable [25].

Syntax and semantics of OWL DL is presented in [33] and summarized in the tables in the Appendix B, which are from [26].

OWL ontologies are in general RDF graphs and can also be represented by RDF triples, but the usual syntactic form to denote OWL ontologies is RDF/XML, which is not designed for human-readability but for machine communication.

**Example 2.** To give an example for an OWL ontology, we translate the DL knowledge base $L$ in Example 1 into OWL abstract syntax, accordingly to the tables from Appendix B.

```
Class( Child partial
   restriction( hasFather someValuesFrom(Father))
   restriction( hasMother someValuesFrom(Mother)) )
ObjectProperty( hasFather super(hasParent) Functional)
ObjectProperty( hasMother super(hasParent)
   inverseOf(isMotherOf))
ObjectProperty( isMotherOf InverseFunctional)
Individual( Katty type(Child) value(hasFather John))
Individual( Susan value(isMotherOf Katty))
```

The concrete syntax of OWL in RDF/XML is much more verbose, for instance, the first axiom in Example 1 has the following scheme:

```
<owl:Class rdf:about="#Child">
 <rdfs:subClassOf>
  <owl:Class>
   <owl:intersectionOf rdf:parseType="Collection">
    <owl:Restriction>
     <owl:onProperty rdf:resource="#hasFather"/>
     <owl:someValuesFrom rdf:resource="#Father"/>
    </owl:Restriction>
   </owl:intersectionOf>
  </owl:Class>
 </rdfs:subClassOf>
</owl:Class>
```
Chapter 2. Background

2.2 Description Logic Programs

This section presents an approach to combining terminological reasoning in the form of ontologies, which are based on Description Logics, with logic programs under answer-set semantics.

A description logic program (dl-program) \([10, 11]\) consists of a description logic knowledge base \(L\) and a generalized normal logic program \(P\), which may contain queries to \(L\). In a query one asks if a given description logic formula or its negation follows from \(L\) or not.

2.2.1 Syntax of dl-programs

A dl-program is a pair \(KB = (L, P)\), where \(L\) is a description logic knowledge base and \(P\) is a generalized logic program, that is a finite set of description logic rules (or dl-rules). Such dl-rules are similar to rules in logic programs with negation as failure, but may also contain queries to \(L\) in their bodies, written as special atoms – dl-atoms.

We first define dl-queries and dl-atoms, which are used to express queries to the description logic knowledge base \(L\). A dl-query \(Q(t)\) is either:

- a concept inclusion axiom \(F\) or its negation \(\neg F\), or
- of the forms \(C(t)\) or \(\neg C(t)\), where \(C\) is a concept and \(t\) is a term, or
- of the forms \(R(t_1, t_2)\) or \(\neg R(t_1, t_2)\), where \(R\) is a role and \(t_1, t_2\) are terms.

A dl-atom is an atom of the form:

\[
DL[S_1op_1p_1, \ldots, S_mop_mp_m; Q](t), \quad m \geq 0
\]

where:

- each \(S_i\) is either a concept or a role;

The full OWL concrete syntax of this example can be found in Appendix B.
Chapter 2. Background

- \( \text{op}_i \in \{\cup, \cap, \}\); 
- \( p_i \) is a unary (resp. binary) predicate symbol if \( S_i \) is a concept (resp. role); 
- \( Q(t) \) is a dl-query.

The sequence \( S_1 \text{op}_1 p_1, \ldots, S_m \text{op}_m p_m \) is the input context of the dl-atom.

The operators \( \cup \) and \( \cap \) are used to extend the program \( P \) view of the description logic knowledge base \( L \) in the (local) context of the current dl-query. Intuitively, \( \text{op}_i = \cup \) (resp. \( \text{op}_i = \cap \)) increases \( S_i \) (resp. \( \neg S_i \)) by the extension of \( p_i \) before evaluating the query, while \( \text{op}_i = \cap \) constrains \( S_i \) to \( p_i \). This does not change \( L \), only affecting \( P \)'s current view of \( L \), and only locally in the scope of the specific dl-atom.

Finally, a dl-rule \( r \) has the form:

\[
a \leftarrow b_1, \ldots, b_n, \textbf{not} \ b_{n+1}, \ldots, \textbf{not} \ b_m
\]

where \( a \) is a logic atom and any literal \( b_1, \ldots, b_m \in B(r) \) may be a dl-atom, as well as a regular one. We define \( H(r) = a \), where \( H(r) \) is the head of rule \( r \), and \( B(r) = B^+(r) \cup B^-(r) \), where \( B(r) \) is the body of rule \( r \), \( B^+(r) = \{b_1, \ldots, b_n\} \) and \( B^-(r) = \{b_{n+1}, \ldots, b_m\} \). If \( B(r) = \emptyset \), then \( r \) is a fact.

The two components of a dl-program – the description logic knowledge base and the logic program – are kept independent, communicating only through dl-atoms. So, although these components function separately, giving modularity properties to dl-programs, there is a bidirectional flow of information via dl-atoms.

The following examples illustrate the main ideas about dl-programs; the first example will be used in the next sections.

**Example 3.** (People Management [23])

Based on certain information about papers, persons and work positions, we want to know who is a good manager and who is overloaded. Consider the following description logic knowledge base \( L_R \), which contains some knowledge about papers and people management:

\[
(\geq 2 \text{paperToReview}) \sqsubseteq \text{Overloaded} \\
\text{Overloaded} \sqsubseteq \forall \text{supervises}^+.\text{Overloaded} \\
\{(a, b)\} \cup \{(b, c)\} \sqsubseteq \text{supervises}
\]

Here, the first axiom defines the concept \text{Overloaded} by putting a cardinality constraint on \text{paperToReview}; in other words, this axiom indicates that someone who has more than two papers to review is overloaded.

The second axiom, where \text{supervises}^+ is the transitive closure of the role \text{supervises}, indicates that an overloaded person causes all her supervised persons to be overloaded as well.
The last axiom, which is equivalent to the assertions \( \text{supervises}(a, b) \) and \( \text{supervises}(b, c) \), defines the supervision hierarchy.

Consider now the dl-program \( KB_R = (L_R, P_R) \), with \( L_R \) as above and \( P_R \) given as follows:

\[
\begin{align*}
goodManager(X) & \leftarrow DL[; \text{supervises}](X, Y). \quad (r_1) \\
\not \ DL[\text{paperToReview} \sqcup \text{paper}; \text{Overloaded}](Y). \quad (r_2) \\
overloaded(X) & \leftarrow \not \ goodManager(X). \quad (r_3) \\
paper(b, p1). \quad (r_4)
paper(b, p2). \quad (r_4)
\end{align*}
\]

Rule \( r_1 \) intuitively says that \( X \) is a good manager, if \( X \), according to \( L_R \), supervises someone who is not overloaded, where the extensional part of the \( \text{paperToReview} \) role in \( L_R \) (which is known to influence \( \text{Overloaded} \)) is augmented by the facts \( r_3 \) and \( r_4 \) of the binary predicate \( \text{paper} \). Rule \( r_2 \) specifies that whoever is not a good manager is overloaded. In this case, \( \text{overloaded} \) from \( P_R \) is not directly related with the concept \( \text{Overloaded} \) form \( L_R \). Rules \( r_3 \) and \( r_4 \) are facts of the binary predicate \( \text{paper} \).

**Example 4. (Reviewer Selection [11])**

Suppose that we want to assign reviewers to papers, based on certain information about the papers and available persons, using a description logic knowledge base \( L_S \) containing knowledge about scientific publications. More concretely, \( L_S \) classifies papers into research areas, stored in a concept \text{Area}, depending on keyword information. The roles \text{keywords} and \text{inArea} associate with each paper its relevant keywords and the areas that it is classified into, respectively; a paper is in an area if it is associated with a keyword of that area. Furthermore, a role \text{expert} relates persons to their areas of expertise; for simplicity, a person is an expert in an area if he or she wrote a paper in that area. The concept \text{Referee} contains all referees. Eventually, a role \text{topicOf} associates with a cluster of similar keywords all its members.

Consider then the following dl-program \( P_S \):

\[
\begin{align*}
paper(p_1). \quad (r_1) \\
paper(p_2). \quad (r_2) \\
kw(p_1, \text{Semantic\_Web}). \quad (r_3) \\
kw(p_2, \text{Bioinformatics}). \quad (r_4) \\
kw(p_2, \text{Answer\_Set\_Programming}). \quad (r_5) \\
kw(P, K_2) & \leftarrow kw(P, K_1), DL[; \text{topicOf}](S, K_1), DL[; \text{topicOf}](S, K_2). \quad (r_6) \\
paperArea(P, A) & \leftarrow DL[\text{keywords} \sqcup kw; \text{inArea}](P, A). \quad (r_7) \\
candidate(X, P) & \leftarrow paperArea(P, A), DL[; \text{Referee}](X), DL[; \text{expert}](X, A). \quad (r_8) \\
assign(X, P) & \leftarrow candidate(X, P), \not \ nonAssign(X, P). \quad (r_9)
\end{align*}
\]
nonAssign(Y, P) ← candidate(X, P), assign(X, P), X ≠ Y.  \( (r_{10}) \)
\[ a(P) ← assign(X, P). \] \( (r_{11}) \)
\[ error(P) ← paper(P), not\ a(P). \] \( (r_{12}) \)

Intuitively, facts \( r_1 \)–\( r_5 \) specify the keyword information (by the predicate \( kw \)) of two papers, \( p_1 \) and \( p_2 \), which should be assigned to reviewers. The rule \( r_6 \) allows for retrieving keyword information from \( L_S \). The predicate \( kw \) is augmented via dl-atoms by those keywords in \( L_S \) that share the same topic.

Rule \( r_7 \) intuitively says that paper \( P \) is in area \( A \) if \( P \) is in \( A \) according to \( L_S \), where the extensional part of the role keyword in \( L_S \) (which is known to influence inArea) is augmented by the facts of the binary predicate \( kw \) from the program. In other words, this rule queries the augmented \( L_S \) to retrieve the areas each paper is classified into.

Rule \( r_8 \) defines reviewers candidates for a given paper. A reviewer \( X \) is candidate to review the paper \( P \), if the paper is in area \( A \) and \( X \) is known in \( L_S \) to be a Referee and an expert in the area \( A \).

Rules \( r_9 \) and \( r_{10} \) pick one of the candidate reviewers for a paper (multiple reviewers can be selected similarly). Finally, \( r_{11} \) and \( r_{12} \) check if each paper is assigned; if not, an error is flagged. In view of rules \( r_6 \)–\( r_8 \), information flows in both directions between the description logic knowledge base \( L_S \) and the knowledge represented by the dl-program.

To illustrate the use of \( \cap \), imagine one wants to define a unary predicate \( possibleReferees \) in \( P_S \), and to add “Referee \( \cap \) possibleReferees” in the first dl-atom of \( r_8 \). The effect of this modification would be to add to \( L_S \) negative assertions \( \neg\)Referee(r) for all the \( r \) such that possibleReferees does not hold, thus constraining the possible referees to the domain of possibleReferees.

The dl-rule below, which defines a new predicate expert, shows in particular how dl-rules can be used to encode certain qualified number restrictions (represented by the letter \( Q \)), which are not available in \( SHOTN(D) \). Here, the term “qualified” means that we do not express restrictions on the overall number of values of a property, but only on the number of values of a certain type. The dl-rule defines an expert as an author of at least three papers of the same area:

\[
\text{expert}(X, A) ← DL[; isAuthorOf](X, P_1), DL[; isArea](P_1, A), \]
\[
DL[; isAuthorOf](X, P_2), DL[; isArea](P_2, A), \]
\[
DL[; isAuthorOf](X, P_3), DL[; isArea](P_3, A), \]
\[
P_1 ≠ P_2, P_2 ≠ P_3, P_3 ≠ P_1.
\]

Even though this formula represents a qualified number restriction, it defines a role, so we could not define it, even if we added \( Q \) to \( SHOTN(D) \).
2.2.2 Semantics of dl-programs

This section formalizes the semantics of dl-programs. Let \( KB = (L, P) \) be a dl-program. The Herbrand base of \( P \), denoted by \( HB_P \), is the set of all ground atoms consisting of predicate symbols and terms that occur in \( P \).

An interpretation \( I \) relative to \( P \) is a consistent subset of \( HB_P \). \( I \) is a model of \( b \in HB_P \) under \( L \), or \( I \) satisfies \( b \) under \( L \), denoted \( I \models_L b \), if \( b \in I \). \( I \) is a model of a ground dl-atom \( a = DL[S_1 op_1 p_1, \ldots, S_m op_m p_m; Q](c) \) under \( L \), or \( I \) satisfies \( a \) under \( L \), denoted \( I \models_L a \), if \( L \cup \bigcup_{i=1}^{m} A_i(I) \models Q(c) \), where

- \( A_i(I) = \{ S_i(e) \mid p_i(e) \in I \} \), for \( op_i = \cup \);
- \( A_i(I) = \{ \neg S_i(e) \mid p_i(e) \in I \} \), for \( op_i = \cap \).

An interpretation \( I \) is a model of a ground dl-rule \( r \) iff \( I \models_L b \) for all \( b \in B^+(r) \) and \( I \not\models_L b \) for all \( b \in B^-(r) \) implies \( I \models_L H(r) \), where \( H(r) \), \( B^+(r) \) and \( B^-(r) \) are the head, the positive and the negative part of the body of rule \( r \), respectively. \( I \) is a model of a dl-program \( KB = (L, P) \), or \( I \) satisfies \( KB \), denoted \( I \models KB \), if \( I \models_L r \) for all \( r \in ground(P) \). So, \( KB \) is satisfiable if it has a model, otherwise \( KB \) is unsatisfiable.

Example 5. (People Management, continued) Given the dl-program \( KB_R \) of Example\[5\] its Herbrand base contains all ground atoms built from applying \( goodManager \), \( overloaded \) and \( paper \) not only to the constants of \( P_R \) \(-\) \( b, p1 \) and \( p2 \) \(-\) but also to all individuals of \( L \) \(-\) \( a, b \) and \( c \), thus, its Herbrand base will be

\[
HB_{KB_R} = \{ goodManager(t), overloaded(t), paper(t_1, t_2) \mid t, t_1, t_2 \in \{ a, b, c, p1, p2 \} \}
\]

This may seem a little odd, since e.g. \( paper(a, c) \) or \( overloaded(p2) \) does not fit well with the intended interpretation of the predicates \( paper \) and \( overloaded \); but this is a well-known side-effect of the absence of types in logic programming.

Examples of interpretations for \( KB_R \) are:

\[
I_1 = \emptyset
\]

\[
I_2 = \{ paper(b, p1), paper(b, p2), overloaded(a) \}
\]

\[
I_3 = \{ paper(a, c), goodManager(p2), overloaded(b) \}
\]

\[
I_4 = \{ paper(b, p1), paper(b, p2), overloaded(a), overloaded(b), overloaded(c), overloaded(p1), overloaded(p2) \}
\]

It is easy to verify that only \( I_4 \) is a model of \( KB_R \). \( I_1 \) and \( I_3 \) are not models of \( KB_R \) since they do not satisfy rules\[3\] and \[4\]. The interpretation \( I_2 \) does not satisfy rule\[2\] with \( X = b \).

\[1\]In the sense that \( I \) does not contain a literal and its negation.
A dl-program is positive if the rules in \( P \) do not contain negations. Positive dl-programs have the same usual properties of positive logic programs, as they have a unique least model \( M_{KB} \) that can be constructed by computing the least fixed-point of the Herbrand transformation \( T_{KB} \), defined as the usual Herbrand transformation for logic programs but using \( L \) to evaluate dl-atoms.

A ground dl-atom \( a \) is monotonic relative to \( KB = (L, P) \) provided that \( I \models_L a \) implies \( I' \models_L a \), for \( I \subseteq I' \subseteq HB_P \). A dl-program \( KB = (L, P) \) is positive if

(i) \( P \) is “not”-free, i.e., for all \( r \in P, B^-(r) = \emptyset \), and

(ii) every ground dl-atom occurring in \( ground(P) \) is monotonic relative to \( KB \).

Notice that while dl-atoms containing only \( \sqcup \) and \( \sqcap \) are always monotonic, a dl-atom containing \( \sqcap \) may fail to be monotonic, since an increasing set of \( p_i(e) \) in \( P \) results in a reduction of \( \neg S_i(e) \) in \( L \). The dl-program \( KB_R \) in Example 3 is not a positive dl-program because of rules \( r_1 \) and \( r_2 \).

For any dl-program \( KB = (L, P) \), one denotes by \( DL_P \) the set of all ground dl-atoms that occur in \( ground(P) \). It is assumed that \( KB \) has an associated set \( DL^+_P \subseteq DL_P \) of ground dl-atoms which are known to be monotonic, and one denotes by \( DL^-_P = DL_P \setminus DL^+_P \) the set of all other dl-atoms.

**Answer-Set Semantics**

The answer-set semantics of dl-programs is defined in analogy to that of logic programs. There are two possible generalizations: strong and weak answer-set semantics.

**Strong answer-set semantics**

Given a dl-program \( KB = (L, P) \), we can obtain a positive dl-program by replacing \( P \) with its strong dl-transformation \( sP_L^I \) of \( P \) relative to \( L \) and a interpretation \( I \subseteq HB_P \). This is obtained by grounding every rule in \( P \) and then

(i) deleting every dl-rule \( r \) such that either \( I \not\models_L a \) for some \( a \in B^+(r) \cap DL^+_P \), or \( I \models_L b \) for some \( b \in B^-(r) \), and

(ii) deleting from each remaining dl-rule \( r \) all literals in \( B^-(r) \cup (B^+(r) \cap DL^-_P) \), in other words, deleting the negative body from each remaining dl-rule.

This is a generalization of the Gelfond-Lifschitz reduct. Note that \( KB^I = (L, sP_L^I) \) is a positive dl-program, which has a unique least model \( M_{KB^I} \) if it is satisfiable. \( I \subseteq HB_P \) is called a strong answer set of \( KB \) iff it is the least model \( M_{KB^I} \) of \( KB^I \).

**Example 6.** (People Management, continued) Consider again the dl-program \( KB_R \) from Example 3 and the interpretations \( I_1, I_2, I_3 \) and \( I_4 \) defined in Example 5. We can verify that \( I_4 \) is a strong answer set for \( KB_R \).
First, we compute $s_{P_{L,R}^I}$. Consider all ground instances of $r_1$; there are only two cases when $I_4 \models DL[\text{supervises}](X, Y)$: when $X$ is $a$ and $Y$ is $b$, and when $X$ is $b$ and $Y$ is $c$, so all the other ground instances of $r_1$ will be removed from $s_{P_{L,R}^I}$. Moreover, $I_4 \models DL[\text{paperToReview} ⊔ \text{paper}; \text{Overloaded}](b)$ by the first axiom from $L_R$, and $I_4 \models DL[\text{paperToReview} ⊔ \text{paper}; \text{Overloaded}](c)$ by the second and third axioms. So, $s_{P_{L,R}^I}$ does not contain ground instances of $r_1$. Regarding $r_2$, since $I_4$ does not contain $\text{goodManager}(t)$ for any $t$, all its ground instances will be included in $s_{P_{L,R}^I}$ with their body removed (ii). Lastly, $r_3$ and $r_4$ are grounded rules with no negative literals in their bodies, so they are copied to $s_{P_{L,R}^I}$ unchanged. Therefore, $s_{P_{L,R}^I}$ is the following program:

$$
\begin{align*}
\text{overloaded}(a) & \quad \text{overloaded}(b) & \quad \text{overloaded}(c) \\
\text{overloaded}(p1) & \quad \text{overloaded}(p2) & \quad \text{paper}(b, p1) & \quad \text{paper}(b, p2)
\end{align*}
$$

As $s_{P_{L,R}^I}$ contains only facts, it is easy to compute the least model of $KB_{R}^{I_4}$ and check that it coincides with $I_4$. So, $I_4$ is an answer set for $KB_{R}$.

For comparison purposes, consider the interpretation $I_2$. It is easy to verify that $s_{P_{L,R}^I}$ coincides with the $s_{P_{L,R}^I}$, since in this case the construction of the reduct just depends on the instances included in the interpretation. So, the least model of $KB_{R}^{I_2}$ is $I_4$, and thus $I_2$ is not an answer set for $KB_{R}$. The interpretation $I_4$ is actually the only strong answer set for this dl-program.

**Weak answer-set semantics** The weak dl-transformation of $P$ relative to $L$ and to an interpretation $I \subseteq HB_P$, denoted $wP^I_L$, is the ordinary positive program obtained from $\text{ground}(P)$ by

(i) deleting all dl-rules $r$ where either $I \not\models_L a$ for some dl-atom $a \in B^+(r)$, or $I \models_L b$ for some $b \in B^-(r)$, and

(ii) deleting from every remaining dl-rule $r$ all the dl-atoms in $B^+(r)$ and all the literals in $B^-(r)$.

Weak answer sets associate a larger set of models than strong answer sets. Notice that strong answer sets of $KB$ are weak answer sets of $KB$, but not vice versa in general.

For any positive dl-program $P$, both the strong reduct as well as the weak reduct coincide with the usual Gelfond-Lifschitz reduct (used in logic programs semantics), and thus the weak and the strong answer sets coincide with the standard answer sets of $P$.

We do not illustrate the weak answer-set semantics with an example, because it will not be used in this dissertation.

**Well-founded Semantics** This semantics will not be used in this dissertation, but will be presented for completeness purposes. The well-founded semantics represents another
widely used semantics for ordinary non-monotonic logic programs, which generalizes well-founded semantics for logic programs.

There are several equivalent ways to define this semantics. We define the well-founded semantics of a dl-program $KB = (L, P)$ by means of the operator $\gamma_{KB}$, such that $\gamma_{KB}(I)$ is the least model of the positive dl-program $KB^I$ defined above.

This operator is anti-monotonic (if $I \subseteq J$, then $\gamma_{KB}(I) \supseteq \gamma_{KB}(J)$), so $\gamma_{KB}$ is monotonic and therefore it has a least and a greatest fixpoint, denoted $\text{lfp}(\gamma_{KB})$ and $\text{gfp}(\gamma_{KB})$, respectively. An atom $a \in HB_P$ is well-founded if $a \in \text{lfp}(\gamma_{KB})$ and unfounded if $a \notin \text{gfp}(\gamma_{KB})$. The well founded semantics of $KB$ is the set containing all well-founded atoms and the negations of all unfounded atoms. Intuitively, well-founded atoms are true in every model of $P$, whereas unfounded atoms are always false.

2.3 Tools

This section provides a survey of tools that calculate the semantics of the dl-programs, describing each one of these available reasoners. In the remainder of this section we explain why the second tool was the chosen reasoner for dl-programs to be extended with Mdl-programs.

2.3.1 NLP-DL

NLP-DL (Nonmonotonic Logic Programming with Description Logics) is a reasoner for dl-programs that can operate under different semantics [14]. This tool, which is available through a Web interface, has been developed by coupling the two state-of-the-art solvers DLV [29], for nonmonotonic logic programs, and RacerPro [16], for description logics. Due to this integration, NLP-DL is a powerful platform for expressive knowledge representation and reasoning, such as ontologies, rules under negation as failure, strong negation besides negation as failure, and constraints.

The system’s architecture integrates the external DLV and RacerPro engines and is composed of several modules implemented in the PHP scripting language, such as a caching module, a well-founded semantics module, an answer-set semantics module, a preprocessing module, and a postprocessing module.

It can be used either to compute the model of a given dl-program or to perform reasoning, both according to a previously selected semantics, which can be chosen between the strong answer-set semantics and the well-founded semantics. The reasoning mode requires the specification of one or more query atoms as input from the user; in the case of answer-set semantics, another choice between brave and cautious reasoning is available\(^2\).

Furthermore, the result can be filtered by specific predicate names.

\(^2\)In cautious reasoning, the result only includes the atoms that are in all answer sets; in brave reasoning, the result includes all atoms that appear in any answer set.
This tool accepts dl-programs as input, given by an ontology in OWL-DL (as processed by RacerPro) and a set of dl-rules in the syntax described in section 2.2.1 where $\leftarrow$, $\sqcup$, $\sqcap$, and $\cap$ are written as “:-”, “+\text{=}”, “-\text{=}”, and “?=\text{}”, respectively. In this tool, a dl-atom without input context does not have “;” as shown in Figure 2.2.

On choosing the “Evaluate” option, the computation procedure is started, interactively calling DLV and RacerPro. If the model generation task is selected, the answer set(s) or the well-founded model found by the program are shown upon termination. If the query evaluation task is selected, the corresponding query answer is given. If query answering under the answer-set semantics is chosen, one can additionally decide between brave and cautious reasoning.

Figure 2.2 shows the task selection part of the Web prototype and the answer set result for the wine example available on this Web page. Two progress bars are also shown to the user, displaying the time spent by each of the external applications: DLV and RacerPro. Below these bars, a status message informs about the currently executed subtask and additionally indicates that the system is in a running state.

### 2.3.2 dlvhex

Another available tool is dlvhex [13], an application for computing the models of so-called HEX-programs [12], that is, higher-order logic programs with external atoms, which are an extension of answer-set programs integrating external computation sources.
HEX-programs are a more general logic programming framework that is still based on answer-set semantics and can be considered as successor to the dl-programs. This new formalism generalizes dl-programs in two levels: the dl-atoms for querying external description logic knowledge bases have been abstracted to accommodate a universal interface for arbitrary sources of external computation, and meta-reasoning has been introduced through higher-order atoms. Intuitively, a higher-order atom allows one to quantify values over predicate names and to exchange predicate symbols with constant symbols freely.

This tool has been implemented in the C++ language as a command-line application that only works for now in operating systems based on UNIX. The dlvhex tool conservatively extends DLV \[29\], so that dlvhex behaves equally to DLV for any ordinary answer-set program. This was a principal design goal of dlvhex: integrating and reusing existing reasoning applications instead of writing them from scratch. Other design goals followed a modular approach concerning the integration of external atom evaluation and exploiting the object-oriented principles for extensibility and maintainability.

A feature of dlvhex is the integration of external sources of computation, therefore this tool provides a way to build plugins that guarantee extensibility and flexibility, trying to keep the interface lean and making external atoms easy to implement by the user. A plugin is represented by a shared library that is linked to dlvhex at runtime and provides one or more external atoms and their evaluation functions. Some plugins are available at the website of dlvhex, such as the DL-plugin.

**DL-plugin for dlvhex**

The Description Logic plugin (DL-plugin) for dlvhex simulates the behaviour of dl-programs within the more general framework of HEX-programs. It was developed to model dl-programs in terms of HEX-programs.

The DL-plugin supports various external atoms for querying concepts and roles to a description logic knowledge base and extending the latter, before submitting a query, by means of atoms input parameters, in accord to the syntax of dl-programs.

Additionally, the DL-plugin provides a converter that processes the syntax of dl-atoms as presented in Section \[2.2.1\] and transforms it into external atoms of HEX-programs, allowing the use of the dlvhex directly as a reasoner for dl-programs.

This plugin is implemented in the C++ language as well as dlvhex and uses RacerPro \[16\] as a DL reasoning engine, being able to process OWL DL ontologies as description logic knowledge bases in the language \(SHOIN(D)\).

These two tools are from the same developers. The dlvhex framework and the DL-plugin are newer than the NLP-DL web prototype. Moreover, the developers guarantee full support, because the dlvhex and its plugins are a work in progress.
So, **DL-plugin** for **dlvhex** is the chosen tool to extending with the generalization of dl-programs – Mdl-programs – proposed by the ITSWeb group and described in Chapter [4]. In the following chapter, an overview of the **DL-plugin** is shown, as well as how this plugin transforms dl-programs into HEX-programs.
Chapter 3

The DL-plugin for dlvhex

In this chapter we describe how the dl-programs are written in the DL-plugin, indicating the concrete syntax for the dl-atoms, as well as illustrating this syntax with an example. These dl-programs are converted to HEX-programs using some external atoms provided by this tool. Section 3.2 provides a detailed explanation of all external atoms available in the DL-plugin with some examples. There is a lack of documentation about these tools. With the purpose of understand them it was needed to do reverse engineering, by analysing the structure and the source code of both tools. The last section of this chapter gives a global view of the dlvhex and its plugin, describing some components, use cases provided by the DL-plugin, and showing the interaction between the dlvhex and its plugins.

3.1 Writing dl-programs in the DL-plugin

A dl-program, as defined in Section 2.2, can be evaluated using dlvhex and the DL-plugin. To process dl-programs with this plugin, the concrete syntax for dl-atoms is

\[ \text{DL}[S_{1}\text{op}_1p_1, \ldots, S_m\text{op}_mp_m; Q](X_1, \ldots, X_n) \]

where \( Q(X_1, \ldots, X_n) \) is a dl-query and \( \text{op}_i \) is "+=" and "-=" for \( \text{op}_i = \sqcup \) and \( \sqcap \), respectively. In this way DL-plugin accepts dl-rules in the syntax described in Section 2.2.1 where \( \leftarrow \) is written as ":=" and dl-atoms as above. A dl-atom without input context does not have ";" as illustrated in the following example.

Example 7. (Example \[\textbf{3}\] continued). To illustrate the concrete syntax of dl-programs in the DL-plugin the \( P_R \) from Example \[\textbf{3}\] with the differences described above, is written as:

\[
\begin{align*}
goodManager(X) & \leftarrow \text{DL}[\text{supervises}](X, Y), \\
& \quad \text{not DL}[\text{papersToReview} \leftarrow \text{paper}]; \text{Overloaded}](Y). \\
overloaded(X) & \leftarrow \text{not goodManager}(X).
\end{align*}
\]

\[\text{1}\]The third operator \( \sqcap \) was removed by the authors in the implementation of the DL-plugin, as it can be defined as an abbreviation of the other two.
Chapter 3. The DL-plugin for dlvhex

paper(b,p1).
paper(b,p2).

This program is in the file overloaded.dlp and the file overloaded.owl contains the ontology in OWL-DL, which is described in Example 3.

To run this dl-program in the DL-plugin for dlvhex, the command is:

$ dlvhex --ontology=overloaded.owl overloaded.dlp

where --ontology=URL is the option to encode the ontology in the program and URL can be a file or a URL to an OWL Ontology.

The output shows the computed answer set:

{overloaded("c"),overloaded("b"),overloaded("a"),
 overladen("p1"),overloaded("p2"),paper(b,p2),paper(b,p1)}

Interestingly, this dl-program does not run on the original version of dlvhex because this tool performs a safety check over the input and this process fails. In this case, the second rule is not dl-safe. In Section 5.3.3 we clarify this issue and explain how it was solved.

This concrete syntax is transformed into HEX-syntax, using some external atoms provided by the DL-plugin. The HEX CONVERTER module allows one to use dl-atoms in the plugin and ensures the correct transformation to HEX-syntax before evaluation by dlvhex.

### 3.2 DL-plugin atoms

DL-plugin atoms are external atoms provided by the DL-plugin that allow one to extend a description logic knowledge base by means of input parameters and then submit a query to this extension.

One can see external atoms as a foreign function interface for accessing services provided by other programming languages or reasoning facilities. In this case, these external atoms allow the communication between the logic program $P$ and the description logic knowledge base $L$.

A detailed explanation of all external atoms available in the DL-plugin with examples is the subject of the remainder of this section.

**Concept Queries** A query to a concept is carried out by the &dlC external atom in the following way:

&dlC[KB,a,b,c,d,Q](X)

with the following input parameters:

- **KB**: constant string denoting the URI or the file path of the OWL-DL ontology, e.g., http://www.example.org/food.owl
Chapter 3. The DL-plugin for dlvhex

a name of a binary predicate used to extend a concept. For instance, specifying a to be the predicate addPerson together with an interpretation that includes the facts addPerson("Person", "Alice") and addPerson("Person", "Bob") will extend the DL-concept Person by the individuals Alice and Bob.

b name of a binary predicate used to extend the complement of a concept. This works as above, but affecting the negation of the concept.

c name of a ternary predicate used to extend a role. For example, using a predicate foo while having an interpretation including the fact foo("knows", "Bob", "Alice") will add the pair ("Bob", "Alice") to the role knows from the ontology.

d name of a ternary predicate used to extend the complement of a role. This works as above, but affecting the negation of the role.

Q constant string that denotes the concept to be queried;

and with the following output:

X individuals of concept Q, possibly including those added by the input parameters, in the scope of this dl-query. If the external atom has a non-ground output, i.e., X is a variable, then it retrieves all known members of concept Q; otherwise, if X is an individual, then it holds iff X is an instance of concept Q.

For a simple query, only the first and the last input arguments are required.

wine(X) :- &dlC["wine.rdf", a, b, c, d, "Wine"](X).

Provided that a, b, c and d do not occur elsewhere in the dl-program, this rule would do nothing else but add all members of the concept Wine of the ontology wine.rdf into the predicate wine.

The term Wine is expressed in RDF and has thus an XML namespace attached to it. Since, in this case, the concept Wine uses the default namespace name http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#, we simply refer to the concepts of all wines as Wine. If Wine were in the scope of a different namespace than the default namespace, this would not be possible. So, if we have an ontology where the concept of all wines is defined as:

<owl:Class rdf:ID="vin:Wine"/>

where the term Wine is bound to the XML namespace vin, to refer to the concept vin:Wine, which is the short for the fully expanded RDF/XML URI

http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#Wine,
we may use two different methods for doing this: (1) we can use the fully expanded concept name in the query part of the external atoms, or (2) we can add a namespace declaration to the logic program. This is accomplished by adding:

```prolog
#namespace(vin, "http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#")
wine(X) :- &dlC["wine.rdf",a,b,c,d,"vin:Wine"](X).
```

**Example 8.** Imagine that we have an ontology with some knowledge about an university and the simple query:

```prolog
student(X) :- &dlC["univ.owl",a,b,c,d,"Student"](X).
```

If we want to extend the concept Freshman by “Bob Smith” before querying Student, we can do:

```prolog
student(X) :- &dlC["univ.owl",a,b,c,d,"Student"](X).
a(Freshman, "Bob Smith").
```

At the second position of its input list, the external atom expects a binary predicate, whose first argument denotes the concept to be extended and the second the actual individuals to be added to the concept. So, this can become very versatile:

```prolog
student(X) :- &dlC["univ.owl",a,b,c,d,"Student"](X).
a(Freshman,X) :- attends(X,Y), firstyearcourse(Y).
```

Adding roles works analogously:

```prolog
student(X) :- &dlC["univ.owl",a,b,c,d,"Student"](X).
c(enrolled,X,Y) :- person(X), studies(X,Y).
```

These extensions of concepts or roles are local and only extend the knowledge base for the purpose of that specific query.

**Role Queries** A query for pairs of a role (object property in OWL) is facilitated by the `&dlR` external atom in the following way:

```prolog
&dlR[KB,a,b,c,d,Q](X,Y)
```

This second atom follows the same input mechanism, but queries a role, hence here the result is binary.

**Example 9.** This example retrieves all the students enrolled in Computer Science, since the external atom retrieves all pairs `(X,ComputerScience)` from the role `enrolled`.

```prolog
studentComputerScience(X) :-
    &dlC["univ.owl",a,b,c,d,enrolled](X,"ComputerScience").
```
**Datatype Role Queries**  A query for pairs of a datatype role (datatype property in OWL) is facilitated by the &dlDR atom in the following way:

\[ &\text{dlDR}[KB,a,b,c,d,Q](X,Y) \]

Datatype roles are properties with literal values as fillers. In OWL, they are distinguished from object roles and therefore also need a separate query. From the point of view of description logics, datatype roles are simply roles.

**Consistency Check**  The atom &dlConsistent tests the given description logic knowledge base for consistency under the specified extensions:

\[ &\text{dlConsistent}[KB,a,b,c,d]. \]

If KB is consistent after possible augmentation of the A-Box according to the input list, the atom evaluates to true, otherwise it evaluates to false.

**Example 10.** The program

\[ a(\text{"Freshman"}, \text{"Bob Smith"}). \]
\[ :- &\text{dlConsistent}[,\text{"univ.owl"},a,a,c,d]. \]

has no answer set, since we augment the ABox of univ.owl by the axioms Freshman ("Bob Smith") and ¬Freshman("Bob Smith").

For all DL-plugin external atoms, when no input to the description logic knowledge base is given, a, b, c and d serve as dummy predicates.

**Conjunctive Queries**  The atom &dlCQ is more general and flexible than the atomic ones presented above. This atom allows any conjunction of concepts (resp. roles) in the query:

\[ &\text{dlCQ}[KB,a,b,c,d,q](\vec{X}) \]

Here, \( \vec{X} \) represents a tuple of arbitrary arity, reflecting the free variables in the conjunction q. Conjunctive queries provide a versatile interface to DL-reasoner. Multiple queries can be joined into a single one, reducing the number of interactions between dlvhex and the DL-reasoner.

However, the conjunction of two queries might not necessarily yield the same result as their join outside the description logic KB.
Example 11. This example from [34] illustrates the use of the external atom \&dlCQ.

Let $KB_N = (L_N, P_N)$ be a dl-program with $L_N$:

\begin{align*}
Zebra \sqsubseteq Animal \\
Lion \sqsubseteq Animal \\
Lion \sqsubseteq \exists eats.Zebra
\end{align*}

and $P_N$:

\begin{align*}
\text{add("Lion", "Bob").} \\
carnivore(X) :- \&dlCQ["L", add, b, c, d, "eats(X,Y),Animal(Y)"](X).
\end{align*}

Since Bob is added to the concept Lion, he must also occur in the relation eats together with a Zebra. That is, even if we don’t know any specific Zebra, Bob will certainly eat one and thus, because each Zebra is also an Animal, be returned for $X$ in the conjunctive query. Let us replace the second rule by the following one:

\begin{align*}
carnivore(X) :- &dlR["L", add, b, c, d, "eats(X,Y)"](X,Y), \\
&dlC["L", add, b, c, d, "Animal(Y)"](Y).
\end{align*}

Here, one uses role and concept queries and joins their result only in the logic program. Given that no explicit tuple occurs in eats, the result of the the first query is empty and so is the extension of carnivore.

As we mentioned at the end of the previous section, dl-programs are transformed into HEX-syntax, through the external atoms shown previously. This transformation is achieved by the HEX CONVERTER module, translating the dl-atoms in the corresponding external atoms.

Example 12. (Example 3 continued). We now illustrate the transformation that occurs in the DL-plugin and that is completely invisible to the user, translating the dl-program $KB_R$ from the Example[7] to the corresponding HEX-program with external atoms.

\begin{align*}
goodManager(X) :- \\
&dlR["overloaded.owl",a,b,c,d,"supervises"](X,Y), \\
\text{not } &dlC["overloaded.owl",a,b,plusR,d,"Overloaded"](Y). \\
plusR("papersToReview",X,Y) :- paper(X,Y). \\
overloaded(X) :- \text{not goodManager(X).} \\
paper(b,p1). \\
paper(b,p2).
\end{align*}

All queries in this program use the ontology overloaded.owl, which contains the description logic knowledge base $L_R$. The first rule queries the role supervises without any input parameters and queries the concept Overloaded with an input parameter that extends the role papersToReview with the predicate paper.
The generality of an external atom in the HEX-syntax results in the inconvenient of having to specify the extension of the description logic knowledge base by these four predicates. In contrast, dl-atoms, having the only purpose of interacting with an ontology, are more intuitive by allowing for a list of mappings of arbitrary length. The HEX CONVERTER module allows to use dl-atoms in a program and ensures the correct transformation to HEX-syntax before the evaluation of dlvhex.

### 3.3 Overview of dlvhex and the DL-plugin

The dlvhex plugin architecture is divided in three parts: (1) initialization, (2) program rewriting and (3) external atoms. In the first part, dlvhex initializes all the plugins; in the second part, dlvhex calls the converters of each plugin, in order to transform the special syntax or syntactic sugar of a program specially designed for a particular plugin into the HEX-syntax. In the third part, dlvhex asks queries to external atoms, defined in the plugin, during the program evaluation.

The plugin API of dlvhex defines some interfaces as PluginInterface, PluginAtom, PluginConverter and PluginOptimizer, to be implemented by the plugins. In the DL-plugin case, it implements these interfaces in the components: RacerInterface, RacerExtAtom, HEXConverter and DLOptimizer.

A brief overview of the structure of the DL-plugin is depicted in Figure 3.1. The initialization phase is processed by dlvhex, which receives the dl-program as input, and by RacerInterface, the plugin interface of DL-plugin, which instantiates and maintains the network connection with the RacerPro reasoner, instantiates the DLCache and tells dlvhex which external atoms are available in the DL-plugin.

In the program rewriting phase, the input program is translated to HEX-syntax by the HEXConverter in the HEX CONVERTER module.

In the last phase, dlvhex delegates to the RacerExtAtom component of the DL-plugin the processing of the external atoms. This component uses the QueryDirector component, which delegates the query creation to RacerBuilder and the answer parser to RacerParser. Only these two components communicate with RacerPro. So, if one wants to adapt the DL-plugin to work with different description logics reasoners, one only needs to change these two components. The TCPIOStream handles the network connection with RacerPro. The DLCache accelerates the query processing, by storing answers from previously processed queries.

The numbers in the Figure 3.1 represent the information flow in this system. The sequential steps are:

1. input: dl-program;
2. instantiate all the modules of DL-plugin;
Figure 3.1: Brief overview of the DL-plugin with its components, their relationships and information flow

3. convert dl-program to HEX-syntax;

4. query handle by DL-plugin:
   (a) query DL-plugin (check cache),
   (b) transform query to RacerPro query,
   (c) retrieve answer;

5. retrieve answer to dlvhex;

6. output: answer sets of the input program.
3.3.1 **DL-plugin use cases**

The use case diagram for the DL-plugin is shown in Figure 3.2, providing an overview of DL-plugin’s usage scenarios and showing that the DL-plugin is the linking bridge between `dlvhex` and RacerPro. The primary actor is the `dlvhex`, which is a proactive actor. The supporting actor is RacerPro as a DL reasoner, which waits for requests from the DL-plugin and answers queries.

The DL-plugin offers four main use cases for `dlvhex`: (1) “Set Options”, (2) “Convert program”, (3) “Optimize Program” and (4) “Query External Atom”. In use case (1), `dlvhex` may set some options in the DL-plugin, which are provided as a list of command line arguments. In use case (2), DL-plugin receives the input program and converts it as explained earlier. In use case (3), DL-plugin may optimize an HEX-program by syntactic transformations. The use case (4) shows the requirements for the external atoms provided by the DL-plugin and the services provided by it.

![Figure 3.2: Use case diagram of the DL-plugin](image-url)
3.3.2 Interaction between dlvhex and the DL-plugin

The dlvhex tool controls all the process, as it is the main process that calls the services provided by its plugins. The interaction between these tools is shown below:

1. dlvhex creates a manager for the Answer Set Program Solver Software (by default, DLV).

2. dlvhex processes all the arguments that are given by the command line; and each input (which can be: a file, a URI or a string) is added to the program.

   2.1. If no arguments are given to the program, this prints the usage help and exits.

3. dlvhex sets up the PluginContainer, which searches plugins for dlvhex in the file system, imports all the plugins to the context, and sets the options for each plugin.

4. All input sources are read to buffers.

5. For each Converter of each plugin:

   5.1. The buffer with the input is converted and the result is saved in the same buffer.

   **In DL-plugin**

   5.1.1. The content of the buffer is written to a string, and this string is passed to a Lexer, which processes the input and converts it to tokens, that are meaningful symbols defined by the grammar of regular expressions in the DLLexer (lexical analysis).

   5.1.2. These tokens are parsed by the DLParser, that is, the parser checks if the tokens form an allowable expression, which is done with reference to the DLGrammar, which defines rules to transform the dl-program to an HEX-program (syntactic analysis).

      5.1.2.1. If the input is not well formed a PluginError is output to the user.

   5.1.3. The result, which is the HEX-program converted from the dl-program as well as the extra rules, is saved in the same buffer.

6. The output of the Converter, which is the correspond HEX-program, is parsed to an Abstract Syntax Tree (AST) as a Program object: this is done with reference to the HexGrammar.
Chapter 3. The DL-plugin for dlvhex

7. Namespaces are inserted, that is, constant names are expanded. A constant name is in the form “prefix”：“localName” or just “localName”, as shown in Section 3.2.

8. For each Rewriter of each plugin:

8.1. The rewriting process, which can rewrite a custom syntax for the external atoms of the plugin, is applied to the Program object, the IDB. Also the set of initial facts, the EDB is passed to the rewriter and can be considered/ altered.

**In DL-plugin**

DL-plugin does not have a Rewriter.

9. Create a Node Graph of the given program.

10. For each plugin that has an Optimizer, perform an optimize operation based on the Node Graph and the EDB.

**In DL-plugin**

10.1. DL-plugin does an equivalence-preserving transformation in rules, where the rule bodies and rules involving dl-atoms can be rewritten. When it is possible, rules can be replaced by other rules, preserving the semantics of the program. For instance, dl-queries of one rule can be transform in a conjunctive query, increasing the performance and decreasing the number of queries to the ontology.

11. Create a Dependency Graph.

12. Perform a Safety Check based on the IDB.

13. Perform a Strong Safety Check based on the Dependency Graph.

14. To evaluate the program, first check the EDB for consistency and then compute the program.

14.1. When dlvhex finds an ExternalAtom, it is passed to DL-plugin.

---

2IDB stands for intensional database, intensional predicate symbols are defined only by rules
3EDB stands for extensional database, extensional predicate symbols are only defined by facts
In DL-plugin

14.1.1. The ExternalAtom is processed, creating a RacerQuery.

14.1.2. The DLCache is checked.

14.1.2.1. If the this query is in the DLCache, an Answer is retrieved to dlvhex.

14.1.3. The communication with RacerPro is opened: first the ontology is increased with the atom extra information and then the dl-query is send to RacerPro.

14.1.4. The retrieved Answer is saved in the DLCache and it is return to dlvhex.

15. Perform a Post Process analysis.

16. Remove namespaces, that is, contract constant names, if specified.

17. To build the output, dlvhex calls all the OutputBuilders of each plugin, and those that know how to build the correspondent output process the output.

In DL-plugin

DL-plugin does not have an OutputBuilder for dl-programs, but dlvhex has a default TextOutputBuilder.

17.1. The output is shown to the user in the command line.
Chapter 4

Multi Description Logic programs (Mdl-programs)

Mdl-programs, proposed by the ITSWeb group\(^7\), generalize the definition of dl-programs, presented in section \(^2.2\), to accommodate several description logic knowledge bases. In particular, dl-programs can be seen as Mdl-programs with only one knowledge base. As we will show, the main advantage of Mdl-programs is their simplicity: these programs are quite adequate for reasoning within the semantic web, where a lot of effort is being put into developing reusable ontologies.

Some mechanisms for combining different reasoning mechanisms are available, such as HEX-programs\(^{12}\) or multi-context systems\(^{4}\). In the first section of this chapter we explain why ITSWeb has defined Mdl-programs instead of working with these frameworks, and what are the advantages of using Mdl-programs instead of merging all the desired ontologies and using the result in a standard dl-program. The second section describes the syntax and the semantics of Mdl-programs, illustrating with an example that uses two ontologies freely available on-line. In the third section, we introduce a useful syntactic construction for Mdl-programs, allowing one to extend (globally) concepts or roles from a knowledge base in an automatic way, with all instances of a predicate in the rule-based program, and reciprocally. These new operators designed to make programming tasks more structured and modular, are also illustrated with an example.

4.1 Why Mdl-programs

Mdl-programs allow working with different ontologies at same time, while keeping them separate. One can think that merging all the relevant knowledge bases into a single is simpler, but this process has its own specific problems, namely relating to consistency – logical and structural. Besides, ontologies are typically very wide-spectrum and in practice one does not work with them entirely. So, keeping them separate reduces the number of compatibility issues that one has to solve. For example, if we have two concepts in two different knowledge bases that should be identified but are logically inconsistent,
this will always be an issue. But if this problem is irrelevant to the objective (when the concepts involved are not used), then not merging the ontologies will not raise that problem; Mdl-programs bypass this issue.

The advantages to keep ontologies separate coincide with the reasons to defending modularity of large-scale systems. It is much more convenient to have independent knowledge bases, which can be physically separated or independently managed, than a gigantic single one. Also, this separation allows us to make the most of the positive aspects of each ontology – relevant when one is very efficient at performing reasoning tasks or when other has richer concepts and relationships.

Other frameworks similar to Mdl-programs include multi-context systems [4] and HEX-programs [12]. MCSs consist of several knowledge bases, with no restriction on the underlying languages, each declaring additional rules that allow communication with the others. Heterogeneous contexts and non-monotonic reasoning are supported in MCSs. HEX-programs were also proposed as a heterogeneous programming language for the Semantic Web, and they were designed for interoperating with heterogeneous sources via external atoms.

Mdl-programs fully support non-monotonicity and limit heterogeneity to two different frameworks: description logics for the knowledge bases part and logic programming for the rule part. This last part represents the “conductor” that “coordinates” the other parts. Mdl-programs are therefore a simpler framework than the other two more powerful alternatives discussed above. Moreover, description logics are at the core of the Semantic Web, with a huge effort being currently invested in the interchange between OWL and a diversity of rule languages, e.g. via the definition of RIF (Rule Interchange Format) [27].

Furthermore, Mdl-programs do not aim at being general frameworks for combining sources of information of different nature. By being much more general, the other frameworks generate their own specific problems, which Mdl-programs avoid, providing a nice framework to combine description logics with rules.

### 4.2 Syntax and semantics

The essential difference between dl-programs and Mdl-programs is the presence of a set of description logic knowledge bases, instead of a single one. So, an Mdl-program (for Multi Description Logic Program) is a pair \( \langle \{ L_1, \ldots, L_n \}, P \rangle \), where:

- each \( L_i \) is a description logic knowledge base;
- \( P \) is a set of \( dl\)-rules; i.e. rules of the form

\[
a \leftarrow b_1, \ldots, b_k, not b_{k+1}, \ldots, not b_p
\]
where $a$ is a logic program atom and each $b_j$, for $1 \leq i \leq p$, is either a logic program atom or a dl-atom relative to $\{L_1, \ldots, L_n\}$.

A dl-atom relative to $\{L_1, \ldots, L_n\}$ has the form:

$$DL_i[S_1 op_1 p_1, \ldots, S_m op_m p_m; Q](t),$$

often abbreviated to $DL_i[\chi; Q](t)$, where:

- $1 \leq i \leq n$
- $S_k, p_k$ and $Q(t)$ are as before
- $op_k \in \{\sqcup, \sqcap\}$

Like in dl-programs, the operators $\sqcup$ and $\sqcap$ are used to extend a description logic knowledge base $L_i$ locally (that is, in the program’s view of $L_i$), with $S_k \sqcup p_k$ increasing $S_k$ by the extension of $p_k$ and with $S_k \sqcap p_k$ increasing $\neg S_k$ by the extension of $p_k$. The difference between dl-programs and Mdl-programs is that, in the latter, dl-atoms add the information to the corresponding $L_i$ and then ask only this description logic knowledge base for the set of terms satisfying the dl-query $Q(t)$.

The semantics of Mdl-programs is a straightforward generalization of the semantics for dl-programs, shown on section 2.2.2. As in dl-programs, the Herbrand base of $P$, denoted by $HB_P$, is the set of all ground atoms built from predicate symbols from $P$ and constants in $P$ or any $L_i$. An interpretation $I \subseteq HB_P$ satisfies a ground atom $a$, $I \models a$, if:

- $a \in HB_P$ and $a \in I$;
- $a$ is $DL_i[S_1 op_1 p_1, \ldots, S_m op_m p_m; Q](c)$ and $L_i \cup \bigcup_{k=1}^m A_k(I) \models Q(c)$, where
  $$A_k(I) = \begin{cases} 
\{S_k(e) \mid p_i(e) \in I\} & \text{if } op_k = \sqcup \\
\{\neg S_k(e) \mid p_i(e) \in I\} & \text{if } op_k = \sqcap
\end{cases}$$

From this, one can define answer-set semantics and well-founded semantics for Mdl-programs as for dl-programs. All the results shown in [10] and in [11] hold for Mdl-programs. Also, Mdl-programs are a generalization of dl-programs in the following way. Let $\langle L, P \rangle$ be a dl-program. Then:

- a set $I \subseteq HB_P$ is a strong answer set for $\langle L, P \rangle$ iff $I$ is a strong answer set for the Mdl-program $\langle \{L\}, P \rangle$;
- The well-founded semantics of $\langle L, P \rangle$ and the well-founded semantics of $\langle \{L\}, P \rangle$ coincide.

---

1The third operator defined in dl-programs can be expressed as an abbreviation using these two operators
The following example illustrates the main ideas about Mdl-programs.

**Example 13.** Consider a simple program \( KB = \langle \{L_1, L_2\}, P \rangle \) that uses two ontologies freely available on-line: a travel ontology `travel.owl` \([28]\) – \( L_1 \), which defines a series of travel-related concepts, including that of (tourist) `Destination`; and a wine ontology `wine.rdf` \([32]\) – \( L_2 \), which compiles a substantial amount of information about wines, including the locations of several important wineries around the world; in particular, this ontology contains a concept `Region` identifying some major wine regions throughout the world.

Using these two ontologies, \( P \) is the following logic program:

\[
\begin{align*}
wineDest(X) & \leftarrow DL_2[; Region](X) & (r_1) \\
wineDest(Tasmania) & \leftarrow & (r_2) \\
wineDest(Sydney) & \leftarrow & (r_3) \\
overnight(X) & \leftarrow DL_1[; hasAccommodation](X,Y) & (r_4) \\
oneDayTrip(X) & \leftarrow DL_1[Destination \uplus \text{wineDest}; Destination](X), \text{not overnight}(X) & (r_5)
\end{align*}
\]

This simple program extends the definition of the predicate `wineDest` with a query to \( L_2 \) in rule \( r_1 \) importing the individuals of the concept `Region` into `wineDest`. Through the facts \( r_2 \) and \( r_3 \) we add these new wine destinations to predicate `wineDest`. Informally, the goal is to have a subconcept of `Destination`, but without actually changing \( L_1 \).

Rule \( r_5 \) identifies the destinations that are only suitable for one-day trips. The possible destinations are selected not only from the information originally in \( L_1 \), but by (i) extending the concept `Destination` of \( L_1 \) with all instances of the predicate `wineDest` in \( P \) (including those from \( L_2 \) via rule \( r_1 \)), and then (ii) querying this extended view of \( L_1 \) for all instances of `Destination`. The result is then filtered using the auxiliary predicate `overnight` defined in rule \( r_4 \) as the set of destinations for which some accommodation is known. This rule uses the role `hasAccommodation` of \( L_1 \), where `hasAccommodation(t_1, t_2)` holds wherever \( t_1 \) is a `Destination` and \( t_2 \) is an accommodation facility located in \( t_1 \). The reason to use rule \( r_4 \) is the usual in logic programming: the operational semantics of negation-as-failure requires that all variables in a negated atom appear in non-negated atoms in the body of the same rule.

We could also extend `Destination` in rule \( r_4 \), but it is not necessary, according to the structure of \( L_1 \), the role `hasAccommodation` is defined as the set of its instances (without any axioms), so changing other concepts or roles has no effect on its semantics.

An important aspect of this example is that `Sydney` is already an individual of \( L_1 \). One of the characteristics of dl-programs is that the atoms of \( P \) may use constants of the knowledge base as terms. Rule \( r_2 \) adds a new constant to the domain of \( KB \) (indirectly via rule \( r_5 \)), but rule \( r_3 \) adds information about an individual already in \( L_1 \). In this case,
Sydney is already an instance of Destination in $L_1$, so the input context in rule $r_5$ adds only the new instances of the predicate wineDest (including those from $L_2$ via rule $r_1$ and excluding Sydney) to this concept.

Note how the extend query in rule $r_5$ is relevant: if Destination were not updated with the information from wineDest, we would not be able to infer e.g. oneDayTrip(Tasmania) nor oneDayTrip(SouthAustraliaRegion), a Region from $L_2$.

The answer-set semantics for this program is simple to compute. The Herbrand base contains all ground atoms from applying the predicates wineDest, overnight and oneDayTrip not only to the constants of $P$ – Tasmania and Sydney – but also to all individuals of both $L_1$ – which includes (among others) Canberra and FourSeasons (which is not an instance of Destination) – and $L_2$ – which includes, for instance, AustralianRegion. Therefore, $HB_{KB}$ contains, among others, the following atoms:

\[
\begin{align*}
\text{wineDest(AustralianRegion)} & \quad \text{overnight(Tasmania)} & \quad \text{oneDayTrip(Canberra)} \\
\text{wineDest(FourSeasons)} & \quad \text{overnight(Tasmania)} & \quad \text{oneDayTrip(Sydney)}
\end{align*}
\]

This program has only one model, which is its only strong answer set. The examination of $L_1$ shows that this ontology only contains one instance of hasAccommodation – a hotel in Sydney. Therefore, the interpretation $I$ contains

\[
\begin{align*}
\text{overnight(Sydney)} & \quad \text{wineDest(Tasmania)} \\
\text{wineDest(Sydney)} & \quad \text{oneDayTrip(Tasmania)}
\end{align*}
\]

and all information obtained from $L_2$ by the rule $r_1$ (i.e. wineDest($t$) for every $t$ for which Region($t$) holds in $L_2$), as well as information inferred via $L_1$ by rule $r_5$ (i.e. oneDayTrip($t$) for the same set of $t$, as well as for every $t \neq$ Sydney for which Destination($t$) holds in $L_1$).

### 4.3 Adding observers

On top of Mdl-programs, a useful syntactic construction was defined in [7], allowing concepts or roles from a description logic knowledge base $L_i$ to be automatically extended (in $P$’s view of $L_i$) with all instances of a predicate in $P$, and reciprocally.

An **Mdl-program with observers** is a pair $\langle KB, O \rangle$ where:

- $KB = \langle \{L_1, \ldots, L_n\}, P \rangle$ is an Mdl-program;
- $O = \langle \{\Lambda_1, \ldots, \Lambda_n\}, \{\Psi_1, \ldots, \Psi_n\} \rangle$, the observer sets, for $1 \leq i \leq n$, where:
  - $\Lambda_i$ is a finite set of pairs $\langle S, p \rangle$;
  - $\Psi_i$ is a finite set of pairs $\langle p, S \rangle$;
where $S$ is a (negation of a) concept from $L_i$ and $p$ is a unary predicate from $P$, or $S$ is a (negation of a) role from $L_i$ and $p$ is a binary predicate from $P$.

Intuitively, $\Lambda_i$ contains the concepts and roles in $L_i$ that $P$ needs to observe, in the sense that $P$ should be able to detect whenever new facts about them are derived, whereas $\Psi_i$ contains the predicates in $P$ that $L_i$ wants to observe. Note that a specific symbol (be it a predicate, concept or role) may occur in different $\Lambda_i$s or $\Psi_i$s; this yields a way to communicate between different knowledge bases, as shown in the example below.

An Mdl-program with observers $\langle KB, O \rangle$ can be transformed in a (standard) Mdl-program $\langle \{L_1, \ldots, L_n\}, P^O \rangle$, where $P^O$ is obtained from $P$ by:

- adding rule $p(X) \leftarrow DL_i[S](X)$ for each $\langle S, p \rangle \in \Lambda_i$, if $S$ is a concept (or its binary counterpart, if $S$ is a role); and
- in each dl-atom $DL_i[\chi; Q](t)$ (including those added in the previous step), adding $S \cup p$ to $\chi$ for each $\langle p, S \rangle \in \Psi_i$ and $S \cup \neg p$ to $\chi$ for each $\langle p, \neg S \rangle \in \Psi_i$.

Writing Mdl-programs with observers has several advantages. First, the program is shorter and more legible – all global observations, or even identifications (when $p$ and $S$ belong to both $\Lambda$ and $\Psi$), are clearly marked. More importantly, the program is more robust with respect to future changes. In particular, consider future changes to $P$; by writing the observers separately, it is guaranteed that every Mdl-atom’s input context is always adequately extended; without this mechanism, this would have to be ensured by hand, and it is well-known that this kind of internal consistency is very easy to lose while developing more complex programs.

The following example illustrates the main ideas about Mdl-programs with observers. We will use the same scenario of Example 13 from section 4.2.

Example 14. The ontologies $L_1$ and $L_2$ are as before. The program $P$ defines a unary predicate $\text{wineDest}$, of which two instances are known ($\text{Tasmania}$ and $\text{Sydney}$), and which should also inherit all instances of the concept $\text{Region}$ from $L_2$. So, we can make $\text{wineDest}$ an observer of $\text{Region}$, adding the pair $\langle \text{Region}, \text{wineDest} \rangle$ to the observer set $\Lambda_2$.

Moreover, the goal is to extend the concept $\text{Destination}$ from $L_1$ with all the instances of $\text{wineDest}$ in $P$. This can be achieved by adding the pair $\langle \text{wineDest}, \text{Destination} \rangle$ to $\Psi_1$, registering $\text{Destination}$ as an observer of $\text{wineDest}$.

We thus obtain the following Mdl-program with observers. The knowledge bases $L_1$ and $L_2$ are unchanged, but $P$ is:

\[
\begin{align*}
\text{wineDest}(&\text{Tasmania}) \leftarrow \quad (r_2) \\
\text{wineDest}(&\text{Sydney}) \leftarrow \quad (r_3) \\
\text{overnight}(&X) \leftarrow DL_1[; \text{hasAccommodation}](X,Y) \quad (r_4) \\
\text{oneDayTrip}(&X) \leftarrow DL_1[; \text{Destination}](X), \textbf{not} \ \text{overnight}(X) \quad (r_5)
\end{align*}
\]
together with the following set of observers:

\[ O = \langle \{\emptyset, \{\text{Region}, \text{wineDest}\}\}, \{\{\text{wineDest}, \text{Destination}\}, \emptyset\} \rangle \]

Transforming this into an Mdl-program, as explained above, we regain the rule \( r_1 \) from Example 13 and the Mdl-atom in rule \( r'_5 \) is also changed back into the old rule \( r_5 \). Furthermore, rule \( r_4 \) becomes

\[
\text{overnight}(X) \leftarrow DL_1[D\text{estination} \sqcup \text{wineDest}; \text{hasAccommodation}](X, Y) \quad (r'_4)
\]

which, as discussed in Example 13, does not affect its semantics.

If we consider the rule \( r_4 \) in its original context, this rule works because there are no axioms characterizing \text{hasAccommodation} in \( L_1 \). However, it is possible that future versions of \( L_1 \) may change this; and it is reasonable to assume that changing the instances of \text{Destination} may affect this role. Using observers, this positive affect is obtained automatically.

However, if one wants to augment a knowledge base only locally in a dl-rule, this can not be accomplished with an observer, since the latter has a global effect in the program.
Chapter 5

Mdl-programs in the DL-plugin for dlvhex

The prototype implementation of Mdl-programs was built upon the DL-plugin for dlvhex, extending this tool with two new aspects: allowing the use of multiple ontologies and providing syntactical support for observers. This chapter presents these aspects in detail as well as some relevant technical issues that emerged during the implementation. In the two first sections we explain how Mdl-programs were implemented in the DL-plugin and how we added observers to this tool. The third section presents some technical issues that arose during the implementation and the solutions adopted. In the fourth section we show how to integrate an additional mechanism for the original dl-programs called lifting, which achieves a complete two-way integration between a knowledge base and a program. We also show how this construction can be achieved in Mdl-programs using observers. In the last section we compare the performance of an Mdl-program and an Mdl-program with observers.

5.1 Processing multiple ontologies

The ITSWeb group proposed Mdl-programs, a way to combine several description logic knowledge bases, described in the previous chapter.

To have Mdl-programs available within the DL-plugin, two changes had to be made:

1. the command-line options of this tool had to be adapted to allow the user to provide the location of all desired ontologies;

2. the processing of dl-atoms had to be adapted to reflect the extended syntax of Mdl-programs and to access the correct ontology.

Instead of one ontology, the parameter \(--ont\)ology= now receives one or more ontologies, in the form \(--ont\)ology=URI[,URI]'. For example, a command to run an Mdl-program in the DL-plugin could look like:
When this option is processed, instead of saving a single ontology in a shared pointer of type Ontology, this set of ontologies is saved in a vector of shared pointers of type std::vector<Ontology>, which can be accessed everywhere in the plugin. The order in which the ontologies are written defines their sequence numbers, used later in dl-atoms. This is in line with the syntax of Mdl-programs.

To use more than one ontology in the DL-plugin – recall that dl-programs in the DL-plugin work on a single knowledge base – it was necessary to adapt the concrete syntax for dl-atoms in the DL-plugin to receive also the Mdl-syntax (Section 4.2). In this way, dl-atoms now have an additional first argument indicating the sequence number of the ontology being used (the i in $L_i$, as in the usual syntax of Mdl-programs). A typical Mdl-atom for an Mdl-program in the DL-plugin has the form:

$$DL[i;S_1op_1p_1,...,S_mop_mp_m;Q](t)$$

where $1 \leq i \leq n$ is the sequence number of the ontology to which this dl-atom refers, according to the list of ontologies in the parameter -ontology=.

To accept this syntax, the DLLexer from the HEX CONVERTER module was modified to recognize a sequence number i and to convert it to a token. A new rule was added to the DLGrammar to recognize a sequence number token and the rule that recognizes a dl-atom was modified to include the non-terminal symbol that represents this new rule.

To process dl-atoms relating to Mdl-programs, the HEX CONVERTER module was modified to receive also the vector of ontologies.

The DLParser, which uses the DLGrammar, now has a function to handle the sequence number in the Mdl-atom, which does the mapping between the sequence number of the ontology and the vector of ontologies, and replaces the sequence number by the URI of the corresponding ontology. This transformation was easier than expected due to the fact that the output of the HEX CONVERTER module is an HEX-program, in which external atoms have as input parameter a constant string denoting the URI of the corresponding ontology to query.

We now illustrate the constructions detailed above with the help of the Example 13 from Section 4.2. The Mdl-program KB is implemented as the following program (programMdl.dlp).

```prolog
wineDest(X) :- DL[2;Region](X).
wineDest("Tasmania").
wineDest("Sydney").
overnight(X) :- DL[1;hasAccommodation](X,Y).
oneDayTrip(X) :- DL[1;Destination+=wineDest;Destination](X),
    not overnight(X).
```
The invocation of `dlvhex` has the following form.

```bash
$ dlvhex --ontology=travel.owl,wine.rdf programMdl.dlp
```

After `dlvhex` sends its inputs to the DL-plugin, the HEX CONVERTER module processes the program and rewrites all Mdl-atoms in the form of HEX-atoms, replacing the ontology identifiers 1 and 2 with the corresponding URIs. The corresponding HEX-program is:

```hex
wineDest(X) :- &dlC["file:/home/wine.rdf",dl_pc_0,dl_mc_0,
wineDest("Sydney").
wineDest("Tasmania").
dl_pc_1("http://www.owl-ontologies.com/travel.owl#Destination",X) :- wineDest(X).
oneDayTrip(X) :- &dlC["file:/home/travel.owl",dl_pc_1,
    dl_mc_0,dl_pr_0,dl_mr_0,"http://www.owl-ontologies.com/travel.owl#Destination"](X), not overnight(X).
onOvernight(X) :- &dlR["file:/home/travel.owl",dl_pc_0,
    dl_mc_0,dl_pr_0,dl_mr_0,"http://www.owl-ontologies.com/travel.owl#hasAccommodation"](X,Y).
```

As one can see, the HEX CONVERTER module also adds the namespace of the corresponding ontology to the constant names denoting concepts and roles.

The answer set shown to the user, which is the answer set computed by the DL-plugin, contains the facts

```hex
wineDest("Sydney")
wineDest("Tasmania")
onOvernight("Sydney")
oneDayTrip("Tasmania")
```

Together with all information inferred by the program (namely, `wineDest(t)` for every `t` for which `Region(t)` holds in `wine.rdf`, and `oneDayTrip(t)` for every `t` for which `Region(t)` holds in `wine.rdf`). This corresponds to the (only) answer set for this Mdl-program, as discussed in Section 4.2.

### 5.2 Adding observers

The support for observers required more profound changes to the DL-plugin, as they are syntactically different from the features already available, and also for efficiency reasons.
To implement observers, in accordance with the definition in Section 4.3, it was necessary to decide how these should be represented.

- To represent a pair in $\Lambda_i$, $(S, p)$, where $S$ is a concept, a role or a negation of either from $L_i$, and $p$ is a predicate from $P$ that needs to be updated every time the extent of $S$ (of the same arity as $p$) in $L_i$ is changed, we add to the program the annotation:

$$i : p \leftarrow S.$$ 

where $i$ is the index corresponding to $L_i$. The motivation for this notation is the idea that the information flows from $S$ to $p$.

- To represent a pair in $\Psi_i$, $(p, S)$, where $p$ is a predicate from $P$ and $S$ is a concept, a role or a negation of either from $L_i$ that needs to be updated every time the extent of $p$ (of the same arity as $S$) in $P$ is changed, we add to the program the annotation:

$$i : p \rightarrow S.$$ 

where $i$ is the index corresponding to $L_i$. The motivation for this notation is the idea that the information flows from $p$ to $S$.

- Furthermore, it is useful to conjugate several pairs from $\Lambda_i$ and from $\Psi_i$ of $L_i$ in the same line. For example,

$$i : p_1 \leftarrow S_1, p_2 \rightarrow S_2, p_3 \rightarrow S_3, p_4 \leftarrow S_4.$$ 

will add $(S_1, p_1)$ and $(S_4, p_4)$ to $\Lambda_i$, as well as $(p_2, S_2)$ and $(p_3, S_3)$ to $\Psi_i$. Furthermore, there may be different annotations for the same $i$ in different places of the Mdl-program.

The motivation behind this apparent lack of structure is to maintain consistence with the tradition in logic programming, where clauses with the same head are typically grouped together. Since $p \leftarrow S$ effectively corresponds to such a clause, it makes sense to group it with other clauses whose head contains $p$.

For the DL-plugin to be able to process those observers, it was necessary to implement a new module called OBSERVER CONVERTER, which translates an Mdl-program with observers to a (standard) Mdl-program as defined in Section 4.3. This new module precedes the HEX CONVERTER module, and the output of OBSERVER CONVERTER is the input of HEX CONVERTER, as is shown in Figure 5.1.

This new module receives the input program with observers, which is passed to the ObserverLexer, which processes the input and converts it to tokens. These tokens are parsed by the ObserverParser, that is, the parser checks if the tokens form an
allowable expression. This is done with reference to the ObserverGrammar, which defines rules to process the observers.

When the parser finds a pair \( \langle S, p \rangle \) from \( \Lambda_i \), in the form \( p \leftarrow S \), it first checks whether \( S \) is a concept or a role from \( L_i \) and adds one of the following rules in the same place in the program \( P \):

\[
\begin{align*}
p(X) & \leftarrow \text{DL}[i;S](X). \quad \text{if } S \text{ is a concept} \\
p(X,Y) & \leftarrow \text{DL}[i;S](X,Y). \quad \text{if } S \text{ is a role}
\end{align*}
\]

To process the other observers in the form \( p \rightarrow S \), it is necessary to add to each
Mdl-atom $DL[i; \chi; S](X)$ (including those added by the pairs from $\Lambda_i$):

- $S \cup p$ to $\chi$ for each $\langle p, S \rangle \in \Psi_i$
- $S \cup -p$ to $\chi$ for each $\langle p, -S \rangle \in \Psi_i$

For efficiency reasons, a different approach was followed in the implementation. To add these new input parameters to every Mdl-atom, even those added by $\Lambda_i$, one needs to process them in a second pass, because they have a global effect on the program.

The first step of this process occurs in the Observer CONVERTER module. When the ObserverParser finds $p \rightarrow S$, it first checks whether $S$ is a concept, a role, or a negation of either, from $L_i$. According to this, an Atom object is created and it is put in an AtomSet (a special type used during the conversion to HEX-atoms) that contains all the observers of this type for each $L_i$.

The second step of this process occurs in the HEX CONVERTER (that has received the AtomSet vector) and does not directly extend the input context of all Mdl-atoms, but rather postpones this task to the translation done by HEX CONVERTER. This module rewrites the input context of each dl-atom as an AtomSet; the AtomSet relative just to $L_i$ collected by Observer CONVERTER (which is empty if there are no observers) is then appended to the AtomSet of the query relative to $L_i$. In this way, there is no need for one extra pass of the whole program. Note that the original behaviour of the DL-plugin is kept unchanged, that is, if an input program does not have observers, this new module, the Observer CONVERTER, checks whether observers exist, does not change this program and passes it unchanged to the next module, the HEX CONVERTER.

We now illustrate the details of this mechanism with the help of Example 14 from Section 4.3. The Mdl-program with observers $\langle KB, O \rangle$ is implemented as the following program (programObs.dlp).

```
wineDest("Tasmania").
wineDest("Sydney").
2: wineDest <- Region.
1: wineDest -> Destination.
overnight(X) :- DL[1;hasAccommodation](X,Y).
oneDayTrip(X) :- DL[1;Destination](X), not overnight(X).
```

Note that the placement of the declaration of wineDest as observing the concept Region is included after the facts about this predicate. This is in keep with the tradition in logic programming.

The invocation of dlvhex has the following form.

```
$ dlvhex --ontology=travel.owl,wine.rdf programObs.dlp
```

\[1\] where $\chi$ is the abbreviation of $S_1 \circ p_1 \circ p_1, ..., S_m \circ p_m \circ p_m$
After \texttt{dlvhex} sends its input to the \texttt{DL-plugin}, the \texttt{OBSE}
\texttt{ERVER CONVERTER} module processes the program, replacing it
with

\begin{verbatim}
wineDest("Tasmania").
wineDest("Sydney").
wineDest(X) :- DL[2;Region](X).

overnight(X) :- DL[1;hasAccommodation](X,Y).
oneDayTrip(X) :- DL[1;Destination](X), not overnight(X).
\end{verbatim}

and transforms 1: \texttt{wineDest} $\rightarrow$ \texttt{Destination}. into an Atom object that is stored
in an AtomSet relative to $L_1$, in this case. Next, the \texttt{HEX CONVERTER}
rewrites all Mdl-atoms in the form of HEX-atoms, replacing the ontologies identifiers with the corre-
sponding URIs, and adding to the queries to \texttt{travel.owl} the extension to the concept
\texttt{Destination} from the observer stored in the Atom object.

The resultant HEX-program is the following:

\begin{verbatim}
wineDest(X) :- &dlC["file:/home/wine.rdf",dl_pc_0,dl_mc_0,
   dl_pr_0,dl_mr_0,"http://www.w3.org/TR/2003/PR-owl-guide-
   20031209/wine#Region"](X).
wineDest("Sydney").
wineDest("Tasmania").
dl_pc_1("http://www.owl-ontologies.com/travel.owl#
   Destination",X) :- wineDest(X).
oneDayTrip(X) :- &dlC["file:/home/travel.owl",dl_pc_1,
   dl_mc_0,dl_pr_0,dl_mr_0,"http://www.owl-ontologies.com/
   travel.owl#Destination"](X), not overnight(X).
ownight(X) :- &dlR["file:/home/travel.owl",dl_pc_1,
   dl_mc_0,dl_pr_0,dl_mr_0,"http://www.owl-ontologies.com/
   travel.owl#hasAccommodation"](X,Y).
\end{verbatim}

As one can see, the predicate \texttt{dl_pc_1} represents the extension of the concept
\texttt{Destination}, which is added to the queries to \texttt{travel.owl}, as we have shown in
Section 4.3. The answer set shown to the user is the same of the example from Section 5.1.

\section{Specific technical issues}

Some technical issues arose during the implementation of Mdl-programs. This section
presents these aspects and the solutions that were implemented to solve them.
5.3.1 Namespaces

As was shown in Section 3.2, namespaces are an important feature of OWL, and they help with identifying a concept, a role or an individual univocally. A small, but very important detail concerns how individuals are shared between the different components of a dl-program. Any individual passing from the logic program to the ontology (through a dl-atom) automatically receives the namespace corresponding to that knowledge base. This is a feature that ensures RacerPro produces correct answers.

One principle of Mdl-programs is that individuals should be shared between the different components. When one has several knowledge bases in an Mdl-program, it is important to identify individuals with the same name in different ontologies, which have different namespaces associated to them. So, a small change was made to the DL-plugin to remove namespaces automatically from the result of dl-queries. This feature can be turned on by the user in the command line when invoking dlvhex, using the parameter --nonamespace. For example, a command using this feature could look like:

```
$ dlvhex --ontology=travel.owl,wine.owl tourism.dlp --nonamespace
```

Namespace removal should be activated by the user when one wants to reason in Mdl-programs in order to obtain true sharing of information between the different ontologies.

5.3.2 Output Builder

The output of dlvhex, the answer set in the case of the DL-plugin, was not shown to the user in a human readable way. An excerpt of the output from the Example 13 from Section 4.2 is shown below:

```
```

In order to have a human readable output, we modified the construction of the output. The dlvhex tool has an interface, the OutputBuilder, to be implemented by other classes in other plugins, that builds the output to be shown in the console. This tool also
provides a PrintVisitor, which implements the well known Visitor pattern. This visitor has functions for each object structure that belongs to the output (e.g. AtomSet here represents an answer set). The first change was adding a break-line between the facts in the output in the function which prints an AtomSet of this visitor.

The second change, which affects only the DL-plugin, was to give the user the chance to choose if he wants to see the namespaces in the answer set or not. By default, DL-plugin prints the namespaces in the output. Now, this can be overridden by using the parameter --out in the command line when invoking dlvhex.

To implement this feature, a new class, the DLOutputBuilder, was created in the DL-plugin, which implements the OutputBuilder interface from dlvhex. This new class removes the namespaces from each fact in the output and also calls the PrintVisitor to print the result.

With these modifications, the excerpt of the answer set from the example above that is shown to the user is now:

\{ oneDayTrip(CurrawongBeach), 
 oneDayTrip(BondiBeach), 
 oneDayTrip(Cairns), 
 oneDayTrip(Canberra), 
 oneDayTrip(CapeYork), 
 oneDayTrip(Woomera), 
 oneDayTrip(Warrumbungles), 
 oneDayTrip(BlueMountains), 
 ... \}

The DLOutputBuilder also removes all the auxiliary domain facts (presented in the next section) from the output.

### 5.3.3 Domain Checker

An additional module, the Domain Checker, was implemented due to a technical necessity. When the semantics of an HEX-program is computed, dlvhex performs a strong-safety check on its input, identifying circular dependencies between predicates. It then requires that, whenever a predicate \( p \) involved in a circularity appears at the head of a rule, the variables in the arguments of \( p \) must appear in the body of the same rule as arguments of a predicate not involved in that circularity.

This condition is not met, in general, by HEX-programs generated from Mdl-programs with observers. A typical example arises when \( p \) and \( S \) mutually observe each other. If we have a dl-program with observers where the knowledge base is travel.owl and an empty logic program with observers:
Chapter 5. Mdl-programs in the DL-plugin for dlvhex

then this generates the extended logic program

destination(X) :- DL[1;Destination += destination;
    Destination](X).

and running it in the dlvhex returns the error message “rule not expansion-safe”.

To circumvent this issue, an extra module was built, the DOMAIN CHECKER, that generates a special predicate domain corresponding to the universal concept T. The DOMAIN CHECKER module has a DomainLexer and a DomainParser that collect the list of all known individuals referred to in the program and add domain(X) to every rule to satisfy the strong-safety check. The DOMAIN CHECKER also obtains all the individuals from the ontology (from each ontology, if it is an Mdl-program). Next, all individuals are added to the program as domain facts.

The information about domain is removed from the output, so this auxiliary predicate is not visible to the user.

With these changes, the example above now generates the intermediate program:

destination(X) :- &dlC["file:/home/travel.owl",dl_pc_1,dl_mc_0,dl_pr_0,
    dl_mr_0,"http://www.owl-ontologies.com/travel.owl#Destination"](X),
    domain(X).
dl_pc_1("http://www.owl-ontologies.com/travel.owl#Destination",X)
    :- destination(X), domain(X).
domain("http://www.owl-ontologies.com/travel.owl#BlueMountains").
domain("http://www.owl-ontologies.com/travel.owl#Coonabarabran").
domain("http://www.owl-ontologies.com/travel.owl#Warrumbungles").
domain("http://www.owl-ontologies.com/travel.owl#Woomera").
domain("http://www.owl-ontologies.com/travel.owl#CapeYork").
domain("http://www.owl-ontologies.com/travel.owl#Canberra").
domain("http://www.owl-ontologies.com/travel.owl#Cairns").
domain("http://www.owl-ontologies.com/travel.owl#BondiBeach").
domain("http://www.owl-ontologies.com/travel.owl#FourSeasons").
domain("http://www.owl-ontologies.com/travel.owl#Sydney").
domain("http://www.owl-ontologies.com/travel.owl#CurrawongBeach").
domain("http://www.owl-ontologies.com/travel.owl#TwoStarRating").
domain("http://www.owl-ontologies.com/travel.owl#ThreeStarRating").
domain("http://www.owl-ontologies.com/travel.owl#OneStarRating").

where domain(X) was added to the rules and all individuals of this Mdl-program were added to the program as domain facts. In this case, all domain facts are individuals from the travel.owl ontology. The answer set that is shown to the user is:

{destination(CurrawongBeach),
destination(Sydney),
destination(BondiBeach),
destination(Cairns),
destination(Canberra),
destination(CapeYork),
destination(Woomera),
destination(Warrumbungles),
destination(Coonabarabran),
destination(BlueMountains)}

Interestingly, domain predicates were a necessary ingredient that was already needed in, but missing from, the original version of the DL-plugin. In particular, the example of closed-world reasoning (Example 5.9 of [11]) is not supported by DL-plugin; private communication with the developers of the DL-plugin confirmed that this was indeed missing from the implementation.

Figure 5.2: Brief overview of DL-plugin with the new DOMAIN CHECKER module
As mentioned in Section 3.1, the dl-program from the Example does not run on the original version of this tool, as the rule \( \text{overloaded}(X) :\neg \text{goodManager}(X) \) is not dl-safe. With the inclusion of this new module, this rule is changed to: \( \text{overloaded}(X) :\neg \text{domain}(X), \neg \text{goodManager}(X) \). In this way, the negation as failure in the logic program works as expected. Thus, this example now runs in the DL-plugin as it should and produces the expected output.

Figure 5.2 shows that this new module processes the program after the HEX CONVERTER, as all previous modules may add new rules to the input program and the DOMAIN CHECKER has to process all rules.

One can turn off this module using the parameter --nodomain in the command line when invoking dlvhex.

5.4 Lifting

Our group proposed in [6] a new mechanism called lifting in dl-programs, to obtain a complete two-way integration between the description logic knowledge base and the rule-oriented program.

The separation between these two worlds is seen as a positive aspect of dl-programs, but it has the disadvantage that the flow of information is not symmetric. The effects of the answers from the knowledge base are permanent on the program, whereas no effect is seen in the knowledge base, as the extensions of concepts or roles in queries are local and meant to extend the knowledge base in the context of one query (the one which contains the given DL-atom).

With this lifting mechanism, a concept or a role from the knowledge base can be lifted to the program and thus be accessible both in the program and in the knowledge base. In this way, lifting identifies a predicate from the program with a concept or a role from the knowledge base (in the sense that they become “the same”). In other words, the deductions one makes are automatically reflected globally on both levels.

The dl-program with lifting \( KB_{\Gamma} \), where \( KB = \langle L, P \rangle \) is a dl-program and \( \Gamma = \{Q_1, \ldots, Q_m\} \) is a finite set of \( L \)-predicates, is the dl-program \( \langle L, P_{\Gamma} \rangle \) where \( P_{\Gamma} \) is obtained from \( P \) by:

- for every \( Q \in \Gamma \), adding the rules
  \[
  q^+(X) \leftarrow DL;[Q](X) \\
  q^-(X) \leftarrow DL;[\neg Q](X)
  \]
- replacing every dl-query \( DL[\chi; R](t) \) (including those added in the previous step) with
  \[
  DL[\chi, Q_1 \cup q^+_1, Q_1 \cup q^-_1, \ldots, Q_m \cup q^+_m, Q_m \cup q^-_m; R](t)
  \]
where \( \chi \equiv S_1 op_1 p_1, \ldots, S_n op_n p_n \) corresponds to the original query’s input. We will call the query above a \( \Gamma \)-extended query and abbreviate it to \( DL_{\Gamma}[\chi; R](t) \).
In practice, one can define a dl-program with lifting simply by giving \( KB \) and \( \Gamma \). The symbols \( q^+ \) and \( q^- \) represent \( Q \) and \( \neg Q \) in the logic program.

Lifting in dl-programs was implemented by Daniel Santos, another member of the group. He programmed a new module, LIFTING, that processes a dl-program with lifting. The set \( \Gamma \) is written as special “lifting declarations” of the form

\[
\text{Lfc}(Q, q^+, q^-) \quad \text{or} \quad \text{Lfr}(Q, q^+, q^-)
\]

according to whether \( Q \) is a concept or a role.

This new module does the parsing of the new lifting declarations, and translates a dl-program with lifting to a dl-program without lifting, following the transformation defined above. The LIFTING module has as input a \texttt{std::string}, that contains the dl-program with the lifting clauses, and returns a \texttt{std::string} with the processed dl-program.

My contribution in this new mechanism was the integration of the LIFTING module in the DL-plugin for dlvhex. First, the input program is processed by the LIFTING module, and then by the DL-CONVERTER module, which converts dl-programs to HEX-programs. In this way, the dl-program resulting from this new module is then subject to the normal processing by DL-plugin, which has not been modified. In particular, if the input program does not have lifting, then the lifting module passes it without changes to the next module, so that the DL-plugin works exactly as without this module.

One technical aspect emerged also during the implementation. Lifting has always the problem of failing the strong-safety check performed by the dlvhex. Therefore, the DOMAIN CHECKER module was also required in this process.

The following example illustrates this new mechanism for dl-programs.

**Example 15.** This dl-program uses the same travel ontology travel.owl which provides the RuralArea concept. This concept has some individuals as CapeWork and Woomera and the complement of this concept has some cities as Canberra and Sydney. We want to share this concept between both components of the dl-program and add some new information. Then, we have the following program with a lifting declaration to lift the concept RuralArea to the predicates ruralPlus and ruralMinus of the program:

\[
\text{Lfc}(\text{RuralArea}, \text{ruralPlus}, \text{ruralMinus})
\]

\[
\text{ruralMinus}(X) :- \text{australianCity}(X).
\]

\[
\text{australianCity}("Melbourne").
\]

Here, we add the information that an Australian city is not in a rural area; we achieve this by adding the predicate \text{australianCity} to the predicate \text{ruralMinus}, which is identified with the concept \( \neg \text{RuralArea} \). The definition of \text{RuralArea} is truly split between the ontology (which contains all the relationships between this concept and the
others) and the program (which is where the information about Australian cities, which are not a rural area, is fed into it).

This dl-program is processed by the LIFTING module, translating the lifting declaration as described above. The HEX CONVERTER converts this program without lifting to the corresponding HEX-program. The answer set returned by the \texttt{dlvhex} is:

\{australianCity(Melbourne),
    ruralMinus(Melbourne),
    ruralMinus(Sydney),
    ruralMinus(Cairns),
    ruralMinus(Canberra),
    ruralMinus(Coonabarabran),
    ruralPlus(CapeYork),
    ruralPlus(Woomera),
    ruralPlus(Warrumbungles),
    ruralPlus(BlueMountains)\}

where the Australian city Melbourne was added to the complement of RuralArea.

Using Mdl-programs with observers, the same construction can also be mimicked: instead of using the lifting declaration, we can use the same program and take the observers to be the sets \(\Lambda_1 = \{(\text{RuralArea}, \text{ruralPlus}), (\neg \text{RuralArea}, \text{ruralMinus})\}\) and \(\Psi_1 = \{(\text{ruralPlus}, \text{RuralArea}), (\text{ruralMinus}, \neg \text{RuralArea})\}\). So, the corresponding Mdl-program is

1: ruralPlus <- RuralArea, ruralPlus -> RuralArea.
1: ruralMinus <- -RuralArea, ruralMinus -> -RuralArea.

ruralMinus(X) :- australianCity(X).
australianCity("Melbourne").

and it has the same answer set as the lifting construction.

5.5 Performance analysis

We now provide some experimental results about Mdl-programs and Mdl-programs with observers. We have tested the performance of Mdl-programs using the DL-plugin for \texttt{dlvhex} (version 1.7.3) with all new modules that we implemented for Mdl-programs and RacerPro (version 1.9.2).

The tests were performed on an Intel Core 2 Duo 3.00GHz PC with 4GB RAM within a virtual machine running Ubuntu 10.10 with 1GB RAM dedicated. As an ontology benchmark, we used the testsuite about wines described in \cite{30} \footnote{Available at \url{http://kaon2.semanticweb.org/download/test_ontologies.zip}}. It uses the original
wine.rdf ontology, denoted by wine_00 and wine_n (with 01 ≤ n ≤ 10), which is obtained by replicating $2^n$ times the ABox of wine_00. Some statistical information about these ontologies is listed in [30]. To test Mdl-programs, we used the program from Example [13] and for testing Mdl-programs with observers, we used Example [14]. We also tested with or without domain predicates; since these example programs are dl-safe, we can compare the computation time between both.

The travel.owl ontology was not changed; therefore the only variable in these programs is the ontology wine_n. For each test, a new instance of RacerPro was opened. To perform these tests we ran ten times the command (DOMAIN CHECKER module is on):

dlvhex --ontology=travel.owl,$wine $dlp --out

for each program, and ten times the command (here DOMAIN CHECKER module is off):

dlvhex --ontology=travel.owl,$wine $dlp --out --nodomain

for each program, where $wine varies over the ontologies wine_n (with 00 ≤ n ≤ 10) and $dlp is either the Mdl-program program.dlp or the Mdl-program with observers programObs.dlp.

![Figure 5.3: Performance analysis: Mdl-programs vs Mdl-programs with observers (with or without DOMAIN)](image-url)
Figure 5.3 presents the results, where the horizontal axis shows the used ontologies and the vertical axis displays the used time in seconds in a logarithmic scale. As can be seen from the graph, an Mdl-program with observers has almost the same performance as an Mdl-program with the same properties. So, there is not a significant overhead using observers, comparing with similar Mdl-programs.

On the other hand, regarding the domain predicates, we can see that for ontologies with a large ABox, for example the wine_09.owl, we have a significant difference between using or not domain predicates. This is mainly due to the processing overhead by the addition of the predicate domain to all rules in a program and by the addition of the domain facts for all individuals in the Mdl-program.

For fully detailed test results see Table C.1 in Appendix C.
Chapter 6
Conclusions

In this dissertation, we have studied a generalization of dl-programs – Mdl-programs, proposed by the ITSWeb group [7]. An Mdl-program accommodates several description logic knowledge bases and a generalized program, which may contain queries to these ontologies. Particularly, a dl-program can be seen as an Mdl-program with only one knowledge base. These programs have the advantage of keeping ontologies separated: these can be physically apart, independently managed and not available for modifications.

One of the goals of my participation in the ITSWeb group was to contribute with a survey of available tools in this area. Two tools were analysed in more detail: the NLP-DL, a reasoner for dl-programs available through a web prototype, and dlvhex, a tool for computing semantics of HEX-programs. This tool has some available plugins, such as the DL-plugin. This plugin interprets dl-programs in terms of HEX-programs and calculates the answers set of the given program. Comparing these tools, we chose to work with dlvhex, particularly the DL-plugin.

Although dlvhex and the DL-plugin have some years of development, they lack of documentation. With the purpose of understanding these tools it was necessary to do reverse engineering, analysing the structure and the source code of both components. Chapter 3 summarizes all the information retrieved from this process.

The DL-plugin tool allows the computation of dl-programs by means of a concrete syntax for the dl-atoms. This plugin converts a dl-program to an HEX-program translating the dl-atoms into the corresponding HEX-atoms, which are provided by the DL-plugin. Then dlvhex performs the evaluation and, when an external atom is found, it is passed to DL-plugin to handle the query to the ontology, communicating with RacerPro.

We have extended the DL-plugin to allow the processing of Mdl-programs, where one can work with more than one ontology at same time, keeping them completely separated. Through the command line the user can identify which ontologies he wants to work with. In an Mdl-atom inside the logic program, the user specifies the ontology that he wants to extend and query this extension. The flow of information among ontologies is achieved through the logic program, in particular by the input context in each Mdl-atom.
Mdl-programs can be extended by mean of observers, a syntactic construction that allows one to extend concepts or roles from a knowledge base (in the program’s view of that knowledge base) automatically with all instances of a predicate in the logic program or reciprocally. This syntactic construction also was implemented in the DL-plugin as annotations in the logic program. A new module was built that processes these annotations, translating an Mdl-program with observers to a (standard) Mdl-program.

Some technical issues arose during the implementation, which we now briefly summarize.

- When an individual passes from the logic program to the ontology through a dl-atom, it automatically receives the namespace corresponding to that knowledge base. In Mdl-programs it is important to identify individuals with the same name in different ontologies with different namespaces. So, we allow the user to choose if he wants to keep the namespaces or not.

- The output of dlvhex was not shown in a human readable way to the user. So, we have changed this to show one fact per line and we allow the user to choose if he wants to see the namespaces in the answer set or not.

- The dlvhex tool performs a strong-safety check on the input program, identifying circular dependencies between predicates. An example of this circularity arises when one has a predicate and a concept mutually observing each other. To circumvent this issue, we introduce a special predicate domain, which is added to the rules in the program. All the individuals are also added to the program as domain facts. This was already needed in the original version of the DL-plugin; the developers confirmed that it was missing from the implementation.

Having an implementation for Mdl-programs, we analysed the performance of these programs, using a testsuite already available. We can say that an Mdl-program with observers has almost the same performance as a similar Mdl-program. Mdl-programs with observers have the advantage of being shorter and more legible, because all global observations are clearly marked. More importantly, the program is more robust with respect to future changes: with observers, it is guaranteed that every Mdl-atom’s input context is always adequately extended. Without observers, this would have to be ensured by hand, which might lead to inconsistencies.

Ontologies are considered a convenient tool for specifying knowledge in several areas. Also, reusing ontologies and the knowledge associated to them is a key technology for the feasibility of the Semantic Web. So, another of the goals of my participation in the ITSWeb group was to contribute with some case-studies. We wanted to base our examples on ontologies used in real projects, as these seem to reflect more accurate use cases. Most ontologies currently used in practice either have a complex TBox, but no ABox – as the
Hontology [5], the Volkswagen Vehicles Ontology [22], the Volkswagen Sales Ontology [21], The Accommodation Ontology [20] and Tourism Ontology [11] – or they have a large ABox, but a simple TBox – as the Travel Guide System [9], the Travel Ontology [28], the Wine Ontology [32] and the Food Ontology [31]. Throughout this document, we use some of these ontologies that are freely available on the internet.

Appendix A contains the tentative work plan for this dissertation. We deviated slightly from this work plan. The analysis of the dlvhex tool and its plugin, the DL-plugin, took longer than we were expecting as we had to do the reverse engineering. It was also necessary to provide some support to Daniel, an undergraduate student in Applied Mathematics. Daniel and I found a small bug in the interpretation of negated roles by the DL-plugin, making RacerPro process a negated role with negation as failure instead of true negation. This was fixed and reported to the developers of the DL-plugin.

As result this thesis provides some documentation about the dlvhex tool and its plugin, the DL-plugin, which were poorly documented. We also provide an extended version of this tool that can interpret Mdl-programs, Mdl-programs with observers and dl-programs with lifting. Another contribution was the case studies that allow to illustrate and test the implementation of Mdl-programs.

The practical aspects included in this thesis were also included in some publications: a paper about the lifting construction and its implementation [6] was accepted in a national conference (Inforum 2013); a paper about Mdl-programs, including case studies, was submitted to an international conference and it is available as a technical report [8]; and we have a journal article in preparation that includes the details about the implementation of Mdl-programs.
Appendix A

Work plan

1. Familiarization with working context (2 months)
   (a) Study of specific papers in the relevant subject
   (b) Presentation sessions by the members of ITSWeb group
   (c) Survey of available tools

2. Implementation and study of solutions already proposed by the ITSWeb group (4 months)
   (a) Extension of DL-plugin with new syntactic operators and generalizations of DL-programs
   (b) Case-studies
   (c) Performance analysis

3. Development of new solutions (2 months), possibly including:
   (a) Alternative forms of combination and interaction of ontologies
   (b) Active integrity constraints

4. Writing the final report (1 month)
Appendix B

OWL

The abstract syntax for class descriptions and axioms in OWL DL ontologies is given in the first column of Table B.1 and Table B.2 respectively. The second column maps OWL abstract syntax to the corresponding DL syntax and the third column summarizes the semantics of OWL DL, that corresponds to $\mathcal{SHOIN}(D)$.

<table>
<thead>
<tr>
<th>Abstract Syntax</th>
<th>DL Syntax</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Descriptions (C)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A (URI Reference)</td>
<td>A</td>
<td>$A^2 \subseteq \Delta^2$</td>
</tr>
<tr>
<td>owl:Thing</td>
<td>T</td>
<td>owl:Thing$^x = \Delta^2$</td>
</tr>
<tr>
<td>owl:Nothing</td>
<td>1</td>
<td>owl:Nothing$^x = \emptyset$</td>
</tr>
<tr>
<td>intersectionOf($C_1$ $C_2$ ... )</td>
<td>$C_1 \cap C_2$</td>
<td>$C_1^2 \cap C_2^2$</td>
</tr>
<tr>
<td>unionOf($C_1$ $C_2$ ... )</td>
<td>$C_1 \cup C_2$</td>
<td>$C_1^2 \cup C_2^2$</td>
</tr>
<tr>
<td>complementOf($C$)</td>
<td>$\neg C$</td>
<td>$\Delta^2 \setminus C^2$</td>
</tr>
<tr>
<td>oneOf($a_1$ ... )</td>
<td>${a_1, \ldots }$</td>
<td>${a_1^2, \ldots }$</td>
</tr>
<tr>
<td>restriction($R$ someValuesFrom($C$))</td>
<td>$\exists RC$</td>
<td>${x \mid \exists y. (x, y) \in R^2 \land y \in C^2}$</td>
</tr>
<tr>
<td>restriction($R$ allValuesFrom($C$))</td>
<td>$\forall RC$</td>
<td>${x \mid \forall y. (x, y) \in R^2 \land y \in C^2}$</td>
</tr>
<tr>
<td>restriction($R$ minCardinality($n$))</td>
<td>$\geq nR$</td>
<td>${a \in \Delta^2 \mid</td>
</tr>
<tr>
<td>restriction($R$ maxCardinality($n$))</td>
<td>$\leq nR$</td>
<td>${a \in \Delta^2 \mid</td>
</tr>
<tr>
<td>restriction($U$ someValuesFrom($D$))</td>
<td>$\exists UD$</td>
<td>${x \mid \exists y. (x, y) \in U^2 \land y \in D^2}$</td>
</tr>
<tr>
<td>restriction($U$ allValuesFrom($D$))</td>
<td>$\forall UD$</td>
<td>${x \mid \forall y. (x, y) \in U^2 \land y \in D^2}$</td>
</tr>
<tr>
<td>restriction($U$ minCardinality($n$))</td>
<td>$\geq nU$</td>
<td>${a \in \Delta^2 \mid</td>
</tr>
<tr>
<td>restriction($U$ maxCardinality($n$))</td>
<td>$\leq nU$</td>
<td>${a \in \Delta^2 \mid</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>D (URI reference)</th>
<th>D</th>
<th>$D^2 \subseteq \Delta^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>oneOf($v_1$ ... )</td>
<td>${v_1, \ldots }$</td>
<td>${v_1^2, \ldots }$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Object Properties (R)</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>R (URI reference)</td>
<td>R</td>
<td>$\Delta^2 \times \Delta^2$</td>
</tr>
<tr>
<td></td>
<td>$R^r$</td>
<td>$(R^r)^2$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Datatype Properties (U)</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>U (URI reference)</td>
<td>U</td>
<td>$U^2 \subseteq \Delta^2 \times \Delta^2$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Individuals (o)</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>o (URI reference)</td>
<td>o</td>
<td>$o^2 \in \Delta^2$</td>
</tr>
</tbody>
</table>

| Data Values (v) | v | $v^{U^2}$ |

Table B.1: OWL DL syntax vs. DL syntax and semantics

Here, we list the full OWL concrete syntax of Example [ ] from Section 2.1.1.
### Appendix B. OWL

#### Table B.2: OWL DL axioms and facts

<table>
<thead>
<tr>
<th>Abstract Syntax</th>
<th>DL Syntax</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Classes</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Class(A partial $C_1 ... C_n$)</td>
<td>$A \subseteq C_1 \cap ... \cap C_n$</td>
<td>$A^2 \subseteq C_1^2 \cap ... \cap C_n^2$</td>
</tr>
<tr>
<td>Class(A complete $C_1 ... C_n$)</td>
<td>$A = C_1 \cup ... \cup C_n$</td>
<td>$A^2 = C_1^2 \cup ... \cup C_n^2$</td>
</tr>
<tr>
<td>EnumeratedClass($A$ $a_1 ... a_n$)</td>
<td>$A \equiv {a_1, ..., a_n}$</td>
<td>$A^2 \equiv {a_1^2, ..., a_n^2}$</td>
</tr>
<tr>
<td>SubClassOf($C_1$ $C_2$)</td>
<td>$C_1 \subseteq C_2$</td>
<td>$C_1^2 \subseteq C_2^2$</td>
</tr>
<tr>
<td>EquivalentClasses($C_1 ... C_n$)</td>
<td>$C_1 \equiv ... \equiv C_n$</td>
<td>$C_1^2 = ... = C_n^2$</td>
</tr>
<tr>
<td>DisjointClasses($C_1 ... C_n$)</td>
<td>$C_i \cap C_j = \bot, i \neq j$</td>
<td>$C_i^2 \cap C_j^2 = \emptyset, i \neq j$</td>
</tr>
<tr>
<td><strong>Datatype</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DatatypeProperty</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{U}$ super($U_1$) ... super($U_n$)</td>
<td>$U \subseteq U_i$</td>
<td>$U^2 \subseteq U_i^2$</td>
</tr>
<tr>
<td>domain($C_1$) ... domain($C_m$)</td>
<td>$\ni \exists U \subseteq C_i$</td>
<td>$U^i \subseteq C_i^2 \times D_i$</td>
</tr>
<tr>
<td>range($D_1$) ... range($D_l$)</td>
<td>$\ni \forall U : D_i$</td>
<td>$U^i \subseteq D_i^2 \times D_i^2$</td>
</tr>
<tr>
<td>[Functional]</td>
<td>$U \ni \ni U$</td>
<td>$U_i$ is functional</td>
</tr>
<tr>
<td>SubPropertyOf($U_1$ $U_2$)</td>
<td>$U_1 \subseteq U_2$</td>
<td>$U_1^2 \subseteq U_2^2$</td>
</tr>
<tr>
<td>EquivalentProperties($U_1 ... U_n$)</td>
<td>$U_1 \equiv ... \equiv U_n$</td>
<td>$U_1^2 = ... = U_n^2$</td>
</tr>
<tr>
<td><strong>Object</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ObjectProperty</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{H}$ super($H_1$) ... super($H_n$)</td>
<td>$R \subseteq R_i$</td>
<td>$R^2 \subseteq R_i^2$</td>
</tr>
<tr>
<td>domain($C_1$) ... domain($C_m$)</td>
<td>$\ni \exists R \subseteq C_i$</td>
<td>$R^2 \subseteq C_i \times \Delta_i$</td>
</tr>
<tr>
<td>range($C_1$) ... range($C_l$)</td>
<td>$\ni \forall R : C_i$</td>
<td>$R^2 \subseteq \Delta_i^2 \times C_i^2$</td>
</tr>
<tr>
<td>[inverseOf($R_0$)]</td>
<td>$R \equiv (R_0)$</td>
<td>$R^2 = (R_0)^2$</td>
</tr>
<tr>
<td>[symmetric]</td>
<td>$R \equiv (R^-)$</td>
<td>$R^2 = (R^-)^2$</td>
</tr>
<tr>
<td>[Functional]</td>
<td>$U \ni \ni 1R$</td>
<td>$R^2 = (R^-)^2$ is functional</td>
</tr>
<tr>
<td>[InverseFunctional]</td>
<td>$\ni \ni 1R^{-1}$</td>
<td>$(R^2)^{-1}$ is functional</td>
</tr>
<tr>
<td>[Transitive]</td>
<td>$\ni \ni 1R^2$</td>
<td>$(R^2)^2$ is functional</td>
</tr>
<tr>
<td>SubPropertyOf($R_1$ $R_2$)</td>
<td>$R_1 \subseteq R_2$</td>
<td>$R_1^2 \subseteq R_2^2$</td>
</tr>
<tr>
<td>EquivalentProperties($R_1 ... R_n$)</td>
<td>$R_1 \equiv ... \equiv R_n$</td>
<td>$R_1^2 = ... = R_n^2$</td>
</tr>
<tr>
<td><strong>Annotation</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AnnotationProperty($S$)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Individual</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$o$ type($C_1$) ... type($C_n$)</td>
<td>$o \in C_i$</td>
<td>$o^i \in C_i^2$</td>
</tr>
<tr>
<td>value($R_1 o_1$) ... value($R_n o_n$)</td>
<td>${o, o_1} \in R_i$</td>
<td>${o, o_1^i} \in R_i^2$</td>
</tr>
<tr>
<td>value($U_1 o_1$) ... value($U_n o_n$)</td>
<td>${o_1, o_1^i} \in U_i$</td>
<td>${o_1^i, o_1^i} \in U_i^2$</td>
</tr>
<tr>
<td>SameIndividual($o_1 ... o_n$)</td>
<td>$o_1 = ... = o_n$</td>
<td>$o_1^i = ... = o_n^i$</td>
</tr>
<tr>
<td>DifferentIndividual($o_1 ... o_n$)</td>
<td>$o_1 \neq o_j, i \neq j$</td>
<td>$o_1^i \neq o_j^i, i \neq j$</td>
</tr>
</tbody>
</table>

```xml
<?xml version="1.0"?>
<!DOCTYPE rdf:RDF [ 
  <!ENTITY owl "http://www.w3.org/2002/07/owl#" >  
  <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >  
  <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >  
  <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" > ]>

<rdf:RDF xmlns="http://basenamespace.owl#"  
xml:base="http://basenamespace.owl#"  
xml:srdfs="http://www.w3.org/2000/01/rdf-schema#" >
```


<hasFather rdf:resource="#John"/>
</owl:Thing>

<owl:Thing rdf:about="#Susan">
  <rdf:type rdf:resource="&owl;NamedIndividual"/>
  <isMotherOf rdf:resource="#Katty"/>
</owl:Thing>
</rdf:RDF>
Appendix C

Experimental Results

The following table shows the outcome of the experiments described in Section 5.5.

<table>
<thead>
<tr>
<th>ontology</th>
<th>Mdl-program with DOMAIN</th>
<th>Mdl-program without DOMAIN</th>
<th>Mdl-program with observers with DOMAIN</th>
<th>Mdl-program with observers without DOMAIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>wine_00.owl</td>
<td>0.8152</td>
<td>0.7903</td>
<td>0.8727</td>
<td>0.9870</td>
</tr>
<tr>
<td>wine_01.owl</td>
<td>1.3216</td>
<td>1.2402</td>
<td>1.4160</td>
<td>1.4217</td>
</tr>
<tr>
<td>wine_02.owl</td>
<td>1.6597</td>
<td>1.6736</td>
<td>1.7350</td>
<td>1.6781</td>
</tr>
<tr>
<td>wine_03.owl</td>
<td>2.1403</td>
<td>1.9686</td>
<td>2.1752</td>
<td>1.9864</td>
</tr>
<tr>
<td>wine_04.owl</td>
<td>2.6592</td>
<td>2.2981</td>
<td>2.7200</td>
<td>2.5256</td>
</tr>
<tr>
<td>wine_05.owl</td>
<td>3.2022</td>
<td>2.7359</td>
<td>3.2708</td>
<td>2.8564</td>
</tr>
<tr>
<td>wine_06.owl</td>
<td>6.0342</td>
<td>4.9395</td>
<td>6.1490</td>
<td>5.1808</td>
</tr>
<tr>
<td>wine_08.owl</td>
<td>38.2662</td>
<td>18.7943</td>
<td>38.6106</td>
<td>19.4540</td>
</tr>
<tr>
<td>wine_09.owl</td>
<td>139.4109</td>
<td>43.7885</td>
<td>141.4371</td>
<td>44.2773</td>
</tr>
<tr>
<td>wine_10.owl</td>
<td>597.9283</td>
<td>104.4720</td>
<td>602.5614</td>
<td>110.5519</td>
</tr>
</tbody>
</table>

Table C.1: Mdl-program and Mdl-program with observers experiment results (time in seconds)
Bibliography


71


