DISSERTATION

Answer-Set Programming for the Semantic Web

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Doktors der technischen Wissenschaften unter der Leitung von

O.Univ.Prof. Dipl.-Ing. Dr.techn. Thomas Eiter

und

Ao.Univ.Prof. Mag.rer.nat. Dr.techn. Hans Tompits
als verantwortlich mitwirkendem Universitätsassistenten

184/3
Institut für Informationssysteme
Abteilung für Wissensbasierte Systeme

eingereicht an der Technischen Universität Wien
Fakultät für Informatik

von

Roman Schindlauer
9425558
Glockengasse 6/19, A-1020 Wien

Wien, am 14. Dezember 2006

Abstract

This thesis makes a contribution to the research efforts of integrating rule-based inference methods with current knowledge representation formalisms in the Semantic Web. Ontology languages such as OWL and RDF Schema seem to be widely accepted and successfully used for semantically enriching knowledge on the Web and thus prepare it for machine-readability. However, these languages are of restricted expressivity if it comes to inferring new from existing knowledge. On the other side, rule formalisms have a long tradition in logic programming, being a common and intuitive tool for problem specifications. It is evident that the Semantic Web needs a powerful rule language complementing its ontology formalisms in order to facilitate sophisticated reasoning tasks. Ontology languages commonly derive from Description Logics. As a fragment of first-order logic, their semantics diverge significantly from logic programming languages like Datalog and its various descendants — especially if we consider the powerful category of non-monotonic logic programming. In order to overcome this gap, different approaches have been presented how to combine Description Logics with rules, varying in the degree of integration.

Answer-set programming (ASP) is one of the most prominent and successful semantics for non-monotonic logic programs. The specific treatment of default negation under ASP allows for the generation of multiple models for a single program, which in this respect can be seen as the encoding of a problem specification. Highly efficient reasoners for ASP are available, each extending the core language by various sophisticated features such as aggregates or weak constraints.

In the first part of this thesis, we propose a combination of logic programming under the answer-set semantics with the description logics $\mathcal{SHIF}(\mathbf{D})$ and $\mathcal{SHOIN}(\mathbf{D})$, which underly the Web ontology languages OWL Lite and OWL DL, respectively. This combination allows for building rules on top of ontologies but also, to a limited extent, building ontologies on top of rules. We introduce *description logic programs* (*dl-programs*), which consist of a description logic knowledge base L and a finite set of *description logic rules* (*dl-rules*) P. Such rules are similar to usual rules in logic programs with negation as failure, but may also contain *queries to* L, possibly default-negated, in their bodies. We show that consistent stratified dl-programs can be associated with a unique minimal Herbrand model that is characterized through iterative least Herbrand models. We then define *strong* and *weak answer-set semantics* which both properly generalize answer sets of ordinary normal logic programs, based on a reduction to the least model semantics of positive dl-programs and to the answer-set semantics of ordinary logic programs respectively. We also present a definition of the well-founded semantics for dl-programs, based on a generalization of the notion of unfounded sets. We then give fixpoint characterizations for the (unique) minimal Herbrand model semantics of positive and stratified dl-programs as well as for the well-founded semantics, and show how to compute these models by finite fixpoint iterations. Furthermore, we give a precise picture of the complexity of deciding answer set existence for a dl-program, and of brave, cautious, and well-founded reasoning. We lay out possible

applications of dl-programs and present the implementation of a prototype reasoner.

In the second part of the thesis, we generalize our approach to HEX-*programs*, which are nonmonotonic logic programs under the answer-set semantics admitting *higher-order atoms* as well as *external atoms*. Higher-order features are widely acknowledged as useful for performing meta-reasoning, among other tasks. Furthermore, the possibility to exchange knowledge with external sources in a fully declarative framework such as ASP is particularly important in view of applications in the Semantic Web area. Through external atoms, HEX-programs can model some important extensions to ASP, and are a useful KR tool for expressing various applications. We define syntax and semantics of HEX-programs and show how they can be deployed in the context of the Semantic Web. We give a picture of the computation method of HEX-programs based on the theory of splitting sets, followed by a discussion on complexity. Then, the implementation of a prototype reasoner for HEX-programs is outlined, along with a description how to extend this framework by custom modules. Eventually, we show the usability and versatility of HEX-programs and our prototype implementation on the basis of concrete, real-world scenarios.

Kurzfassung

Diese Dissertation ist ein Beitrag zu aktuellen Forschungsbestrebungen Bereich des Semantic Web, speziell bezüglich der Möglichkeiten, regelbasierte Schlussmethoden mit bereits im Einsatz befindlichen Wissensrepräsentationen zu kombinieren. Formalismen und Sprachen zur Darstellung von ontologischen Wissensbasen scheinen weitgehend akzeptiert zu sein, um Wissen im World Wide Web semantisch zu annotieren und damit maschinell verarbeitbar zu machen. Diese Ontologie-Sprachen sind jedoch nur von begrenzter Expressivität, wenn neues Wissen aus vorhandenem geschlossen werden soll. Diese Aufgabenestellung ist seit langem eine Domäne von logikbasierten Regelsprachen, die das Spezifizieren von Problemen auf intuitive Weise erlauben. Es ist allgemein anerkannt, daß zusätzlich zu Ontologien ein ausdrucksstarker Regelformalismus für komplexes Schliessen innerhalb des Semantic Web unerläßlich ist. Ontologie-Sprachen stammen ursprünglich von Beschreibungslogiken ab und sind ein Fragment der Prädikatenlogik erster Stufe. Dadurch unterscheiden sie sich wesentlich von der Semantik von logischen Programmiersprachen, wie z.B. Datalog und speziell von nichtmonotononen logischen Programmiersprachen. Verschiedene Ansätze wurden vorgeschlagen, um diese Diskrepanz zu überwinden und Beschreibungslogiken mit Regelsprachen zu kombinieren.

Answer-Set Programmierung (ASP) ist einer der bekanntesten und erfolgreichsten Vertreter aus der Familie der nichtmonotononen logischen Programmiersprachen. ASP erlaubt es, durch eine spezifische Interpretation der schwachen Negation mehrere Modelle zu einem logischen Programm — in diesem Zusammenhang auch als Kodierung eines Problems gesehen — zu generieren. Mehrere effiziente Inferenzmaschinen für ASP sind verfügbar, wobei jede davon die Kernsprache von ASP mit eigenen Konstrukten, z.B. Aggregaten oder schwachen Constraints, erweitert.

Der erste Teil dieser Arbeit ist einer speziellen Kombination von logischer Programmierung unter der Answer-Set Semantik mit den Beschreibungslogiken $\mathcal{SHIF}(\mathbf{D})$ und $\mathcal{SHOIN}(\mathbf{D})$ gewidmet, die jeweils den Ontologiesprachen OWL Lite und OWL DL zugrunde liegen. Diese Kombination erlaubt sowohl das Aufsetzen von Regeln auf Ontologien, aber auch bis zu einem gewissen Grad das Aufsetzen von Ontologien auf Regeln. Wir führen sogenannte *description logic programs* (*dl-programs*) ein, die aus einer Beschreibungslogik-Wissensbasis L und einer finiten Anzahl von *description logic rules* (*dl-rules*) P bestehen. Diese Regeln basieren auf herkömmlichen Regeln in logischer Programmierung mit schwacher Negation, können zusätzlich jedoch Abfragen von Informationen aus L in ihren Regelrümpfen enthalten. Wir zeigen, daß konsistente und stratifizierte dl-Programme mit einem eindeutigen minimalen Herbrand-Modell assoziierbar sind, das durch iterative kleinste Herbrand-Modelle charakterisiert ist. Wir definieren *Starke* und *Schwache Answer-Set Semantik*, beide als Verallgemeinerung von Answer Sets herkömmlicher normaler logischer Programme, basierend auf einer Reduktion auf die Semantik kleinster Modelle von positiven dl-Programmen bzw. der Answer-Set Semantik von gewöhnlichen logischen Programmen. Weiters definieren wir eine Well-Founded Semantik für dl-Programme, basierend auf

einer Verallgemeinerung des Begriffes des Unfounded Sets. Wir charakterisieren sowohl die (eindeutige) minimale Herband-Modell Semantik von positiven und stratifizierten dl-Programmen als auch die Well-Founded Semantik über einen Fixpunkt und zeigen, wie solche Modelle über endliche Fixpunkt-Iterationen zu berechnen sind. Darüber hinaus zeichnen wir ein präzises Bild der Komplexität der Entscheidbarkeit der Existenz von Answer-Sets eines dl-Programms. Wir präsentieren mögliche Anwendungen von dl-Programmen und stellen die Implementierung eines Prototyps zur Evaluierung von dl-Programmen vor.

Im zweiten Teil dieser Arbeit verallgemeinern wir diesen Ansatz zu sogenannten HEX-*Programmen*, nichtmonotononen logischen Programmen unter der Answer-Set Semantik, die sowohl *Higher-Order Atome* als auch *Externe Atome* zur Verfügung stellen. *Higher-Order Atome* ermöglichen die Spezifikation von Meta-Regeln, während Externe Atome eine Schnittstelle zum Austausch von Informationen mit externen Quellen von Wissen darstellen. Letzteres ist speziell im Hinblick auf das Semantic Web von großem Interesse, da somit verschiedenartige Informationen in einem einzigen deklarativen Formalismus zusammengeführt und verarbeitet werden können. Wir definieren Syntax und Semantik von HEX-Programmen und zeigen, auf welche Weise diese im Kontext des Semantic Web eingesetzt werden können. Wir beschreiben Methoden zur Evaluierung von HEX-Programmen und untersuchen ihre Komplexität. Weiters stellen wir eine Applikation zur Auswertung von HEX-Programmen vor, ergänzt durch eine Beschreibung, wie dieses System mit individuellen externe Schnittstellen erweitert werden kann. Schlussendlich demonstrieren wir die Sinnhaftigkeit und Flexibilität von HEX-Programmen anhand von konkreten, realistischen Szenarien.

Acknowledgements

It is a pleasure for me to thank the many people who made this thesis possible. First and foremost, I must express my gratitude to my professor Thomas Eiter, who never ceased to actively motivating and supporting me throughout the entire time of my PhD studies. One could not think of a better academic and personal guidance, combining an amicable working environment with extraordinary scientific proliferation, yet never too busy not to have an open ear for my questions. Everybody who has the chance to work with Thomas will count oneself lucky, as I have been during the last three years. A great deal of our publications regarding this project are owed to his efforts and ambitions. Hans Tompits, a co-author and colleague in our group, always had an open ear for my countless questions, even when I asked the same ones repeatedly. He taught me a lot about preciseness and diligence, which are so indispensable in any scientific work. Another co-author, Thomas Lukasiewicz, was of invaluable help in working out numerous proofs. Giovambattista Ianni from the University of Calabria, who collaborated with us closely, was not only a patient and helpful colleague, but became also a good friend. He enabled me to visit his department several times, which I always enjoyed professionally as well as personally. Of course, these visits to Calabria wouldn't have been possible without the great hospitality and support of Nicola Leone, the project leader of `DLV`. Nicola also kindly accepted to be my second examinator without hesitation. Among the people in Italy, I further need to thank Wolfgang Faber, who always answered promptly to my inquiries concerning `DLV` and its pitfalls. Axel Polleres and Michael Fink helped me by providing some starting points for the preliminaries section. Moreover, Axel realized a very attractive project using our results, which eventually found its way into this work.

Apart from my colleagues and superiors, I as well benefited much from students I have been supervising or otherwise collaborating with. Jürgen Bock, who did his honours thesis at Griffith University in Brisbane, also designed an appealing application for our work, which I gladly incorporated in this thesis. Thomas Krennwallner taught me at least as much as I taught him as his supervisor, being constantly available for my programming questions — even if completely unrelated to his work.

Our secretary, Elfi Nedoma, deserves special thanks, being extremely efficient in all administrative tasks and letting me fully concentrate on my work. From the arrangement of trips to conferences to the mind-numbing preparations of financial reports for our sponsors, she took care of it all before I even noticed.

Finally, I want to thank my family, who fostered my studies unconditionally from the beginning, without ever questioning my career choice. Also, my friends as well as Caroline continuously supported me, (almost) never complaining about my unusual schedules, especially before submission deadlines.

This thesis was supported by the Austrian Science Fund (FWF) under the project number P17212-N04 and by the European Commission through the IST Networks of Excellence REWERSE (IST-2003-506779).

Contents

Abstract iii

Abstract in German v

Acknowledgements vii

Contents ix

1 Introduction 1
- 1.1 Answer-Set Programming . . . 1
- 1.2 The Semantic Web . . . 2
- 1.3 Combining Rules and Ontologies . . . 4
- 1.4 Thesis Organization . . . 4

2 Preliminaries 7
- 2.1 Declarative Logic Programming . . . 7
- 2.2 Logic Programs under the Answer-Set Semantics . . . 8
  - 2.2.1 Syntax . . . 8
  - 2.2.2 Semantics . . . 9
  - 2.2.3 Available Systems: Restrictions and Extensions . . . 12
- 2.3 Computational Complexity . . . 15
  - 2.3.1 Complexity Classes . . . 15
  - 2.3.2 Complexity of Logic Programming . . . 16
- 2.4 Description Logics . . . 18
  - 2.4.1 $\mathcal{SHOIN}(\mathbf{D})$ and $\mathcal{SHIF}(\mathbf{D})$ . . . 19
  - 2.4.2 Syntax . . . 20
  - 2.4.3 Semantics . . . 21
  - 2.4.4 OWL . . . 22

3 dl-Programs 27
- 3.1 Introduction . . . 27
  - 3.1.1 Logic Programming vs. Classical Logic . . . 27
  - 3.1.2 Strategies for Combining Rules and Ontologies . . . 29
- 3.2 dl-Program Syntax . . . 31
- 3.3 Least Model Semantics for dl-Programs . . . 34
  - 3.3.1 Least Model Semantics of Positive dl-Programs . . . 35
  - 3.3.2 Iterative Least Model Semantics of Stratified dl-Programs . . . 36
- 3.4 Answer-Set Semantics for dl-Programs . . . 37
  - 3.4.1 Strong Answer-Set Semantics of dl-Programs . . . 37
  - 3.4.2 Weak Answer-Set Semantics of dl-Programs . . . 39

3.5 Well-Founded Semantics for dl-Programs . . . 41
3.5.1 Original Definition of the Well-Founded Semantics . . . 41
3.5.2 Generalizing Unfounded Sets . . . 42
3.5.3 Semantic Properties . . . 44
3.5.4 Relationship to Strong Answer-Set Semantics . . . 46
3.6 Computation . . . 48
3.6.1 General Algorithm for Computing Weak Answer Sets . . . 48
3.6.2 Fixpoint Semantics . . . 49
3.6.3 Computing the Well-Founded Model . . . 51
3.7 Complexity . . . 51
3.7.1 Deciding Answer Set Existence . . . 51
3.7.2 Brave and Cautious Reasoning . . . 53
3.7.3 Well-Founded Reasoning . . . 54
3.8 Reasoning Applications . . . 55
3.8.1 Closed-World Reasoning . . . 56
3.8.2 Default Reasoning . . . 58
3.8.3 DL-Safe Rules . . . 62
3.9 Implementing a Solver for dl-Programs . . . 63
3.9.1 Splitting the Input Program . . . 64
3.9.2 Well-Founded Semantics . . . 69
3.9.3 Efficient dl-Atom Evaluation and Caching . . . 72
3.9.4 Prototype . . . 73
3.10 Related Work . . . 74

4 HEX-Programs 79
4.1 Introduction . . . 79
4.2 HEX-Program Syntax . . . 82
4.3 Semantics of HEX-Programs . . . 82
4.4 Modeling ASP Extensions by External Atoms . . . 85
4.4.1 Programs with Aggregates . . . 85
4.4.2 dl-Programs . . . 86
4.4.3 Programs with Monotone Cardinality Atoms . . . 87
4.4.4 Agent Programs . . . 87
4.5 Application Examples . . . 88
4.5.1 Semantic Web Applications . . . 88
4.5.2 Closed-World and Default Reasoning . . . 90
4.6 Computation of HEX-programs . . . 90
4.6.1 Dependency Information . . . 91
4.6.2 Infinite Domains . . . 94
4.6.3 Splitting Algorithm . . . 96
4.6.4 Evaluation Algorithm . . . 99
4.7 Complexity . . . 103
4.8 An Extension: Weak Constraints . . . 104

5 Implementation of a HEX-Reasoner 107
5.1 Architecture . . . 107
5.1.1 Code Layout . . . 107
5.1.2 Plugin Integration . . . 109
5.2 Using dlvhex . . . 110

5.2.1 Installation . . . 110
5.2.2 Calling dlvhex on the Command Line . . . 111
5.2.3 Writing HEX-Programs . . . 111
5.3 Available Plugins . . . 114
5.3.1 The Description Logic Plugin . . . 114
5.3.2 The String Plugin . . . 120
5.3.3 The RDF Plugin . . . 121
5.3.4 The WordNet Plugin . . . 122
5.4 Writing a Plugin . . . 123
5.4.1 The External Atom . . . 123
5.4.2 Registering the Atoms . . . 125
5.4.3 Importing the Plugin . . . 125
5.4.4 The Rewriter . . . 126
5.4.5 Command Line Option Handling . . . 126
5.4.6 Building the Plugin . . . 127

6 HEX-Program Applications **129**
6.1 Optimizing Trust Negotiations . . . 129
6.1.1 The Credential Selection Problem . . . 129
6.1.2 Aggregation by External Computation . . . 131
6.2 Implementing a SPARQL Reasoning Engine . . . 132
6.2.1 The RDF Query Language SPARQL . . . 132
6.2.2 Rewriting SPARQL to Rules . . . 133
6.3 Ontology Merging . . . 134
6.3.1 Merging Algorithm . . . 135
6.3.2 Implementation . . . 136

7 Conclusion **139**

Appendix **143**
A Proofs . . . 143
B String Plugin Source Code . . . 150
C Policy Example Source Code . . . 153

Bibliography **155**

Chapter 1

Introduction

The title of this thesis brings together two prominent themes in the field of knowledge representation and reasoning. Answer-set programming on one hand is a fairly young, yet very successful descendant of a long tradition of logic programming formalisms. The Semantic Web on the other hand, even younger, denotes the current efforts of injecting machine intelligence into the Internet by semantic enrichment in a structured way. Roughly put, in this work we will investigate different approaches how these two lines of research can benefit from each other.

1.1 Answer-Set Programming

In recent years, the *answer-set programming* (ASP) paradigm [Gelfond and Lifschitz, 1991] has emerged as an important tool for declarative knowledge representation and reasoning. This approach is rooted in semantic notions and is based on methods to compute models. More specifically, problems are represented in terms of (finite) theories such that the models of the latter determine the solutions of the original problem.

Among the different ASP techniques proposed in literature, logic programming under the stable model semantics and, as a generalization thereof, the answer-set semantics are two of the most widely used approaches. Both semantics are inherently nonmonotonic, i.e., the set of logical consequences does, in general, not necessarily grow monotonically with increasing information, due to the use of the negation-as-failure operator; moreover, in contrast to procedural semantics like Prolog, they are fully *declarative*.

The answer-set semantics extends the stable model semantics in that the former is defined on a syntactically richer class of programs than the latter. More specifically, the answer-set semantics is defined for *extended logic programs* (ELPs), in which not only negation as failure may occur in program rules, but also *strong negation* (also often referred to as *classical negation*) and disjunctions. On the other hand, the stable model semantics is associated with *normal logic programs* (NLPs), in which only negation as failure occurs as basic operator. The answer-set semantics has recently been extended by Lifschitz et al. [1999] also to a more general class of programs (so-called *nested logic programs*), in which arbitrary Boolean expressions may occur in program rules, albeit this does not yield an enhanced expressibility compared to extended logic programs [Pearce et al., 2001].

Generally speaking, the answer-set semantics is a suitable formalism for handling incomplete and inconsistent information, as well as for expressing non-deterministic features. One of the main reasons for the increasing popularity of both the answer-set semantics as well as the stable model semantics is in large part due to the availability of sophisticated solvers for these languages. On the one hand, the system `DLV` [Leone et al., 2002] is a state-

of-the-art implementation for the answer-set semantics for ELPs, and, on the other hand, the SMODELS system [Niemelä and Simons, 1997] implements the stable model semantics.

Furthermore, in view of its inherent expressibility, the answer-set semantics is a suitable tool to serve as a host language for capturing specialized advanced reasoning tasks. Consequently, in accordance to the general methodology of the answer-set programming paradigm, ASP solvers can be used as underlying reasoning engines for evaluating such dedicated tasks. Different such tasks have been implemented, e.g., on top of the `DLV` system. In particular, `DLV` provides front-ends for planning [Eiter et al., 2001a] and diagnostic reasoning [Eiter et al., 1999a], as well as computing the semantics of updates of nonmonotonic knowledge bases represented as logic programs [Eiter et al., 2001b], or the semantics of inheritance programs [Buccafurri et al., 2002]. Furthermore, the `plp` front-end [Delgrande et al., 2001] to `DLV` allows the computation of different preference approaches under the answer-set semantics, and extensions of the core `DLV` syntax allow the natural formalization of optimization problems, in terms of so-called *weak constraints* [Buccafurri et al., 2000]. A similar flexibility and applicability applies to the SMODELS system as well, which can also be used as a C++-library called from user programs, or as a stand-alone program together with suitable front-ends.

The increasing interest in ASP is also documented by the establishment of the "Working Group on Answer Set Programming (WASP)" which was supported by the European Commission (IST-2001-37004) from 2002 to 2005.

1.2 The Semantic Web

The World Wide Web is impressively successful. Both the information that is stored in the Web and the number of its human users have been growing exponentially in the recent years, now being by far the largest and most frequently-accessed data repository available. For many people, the Web has started to play a fundamental role as a means of providing and searching for information. However, searching the Web in its current form is not always a joyful experience, since today's search engines often return a huge number of answers, many of which are completely irrelevant, while some relevant answers are not returned. One of the main reasons for this problem is that the current Web is designed for human consumption, but not for automated processing through machines, since the HTML standard only allows for describing the layout of Web pages, but not their semantic content. This shortcoming is clearly recognized by the scientific community and there is currently extensive work under way to build the foundations of the next-generation Web addressing these issues.

The *Semantic Web* [Berners-Lee, 1999, Berners-Lee et al., 2001, Fensel et al., 2002] is an extension of the current Web by standards and technologies that help machines to "understand" the information on the Web so that they can support richer discovery, data integration, navigation, and automation of tasks. The Semantic Web will not only allow for more exact answers when we search for information, but also provide knowledge necessary for integrating and comparing information from different sources, and allow for various forms of automated services. Roughly, the main idea behind the Semantic Web is to add a machine-readable meaning to Web pages, to use ontologies for a precise definition of shared terms in Web resources, to make use of knowledge representation technology for automated reasoning from Web resources, and to apply cooperative agent technology for processing the information of the Web.

The development of the Semantic Web proceeds in layers of Web technologies and standards, where every layer is lying on top of lower layers, as shown in Figure 1.1. According

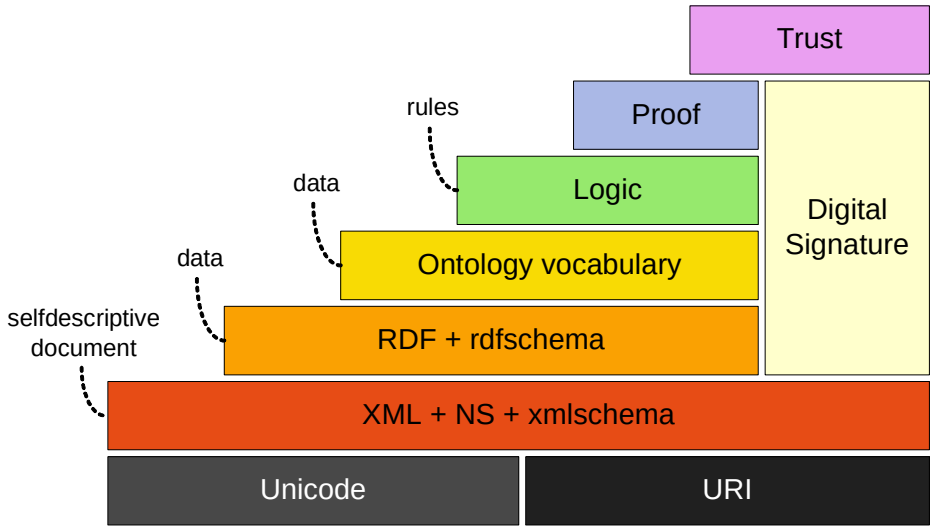


Figure 1.1: The Semantic Web Layer Cake.

to Berners-Lee [1998], and following Hendler [2002], the Semantic Web can be divided into the following layers:

- At the bottom layer, we find standards for identifying resources (URI: uniform resource identifier) and for representing typed text (Unicode);
- the next layer contains languages for annotating information items, like XML (including namespaces and XML Schema);
- the third layer provides formalisms to express meta-data expressions by means of the *Resource Description Framework* (RDF) and its extension *RDF Schema*;
- the fourth layer contains ontology vocabularies to express relative semantics to concepts (like "flying is a form of traveling"); and
- the final layers deal with *logic*, *proof*, and *trust* issues.

The Digital Signature layer is supposed to provide means to identify the proper origin of a specific resource.

The highest layer that has currently reached a sufficient maturity is the ontology layer in the form of the *OWL Web Ontology Language* [McGuinness and van Harmelen, 2004, Horrocks et al., 2003], which has been accepted as standard by the W3C. However, still an open issue is the specification of the final layers of the Semantic Web. To quote Berners-Lee et al. [2001]:

> *The challenge of the Semantic Web, therefore, is to provide a language that expresses both data and rules for reasoning about the data and that allows rules from any existing knowledge-representation system to be exported onto the Web.*

The next and ongoing step in the development of the Semantic Web is the realization of the rules, logic, and proof layers, which will be developed on top of the ontology layer, and which should offer sophisticated representation and reasoning capabilities. A first, yet mostly syntactical effort in this direction was *RuleML* (Rule Markup Language) [Boley

et al., 2001], fostering an XML-based markup language for rules and rule-based systems, while the OWL Rules Language [Horrocks and Patel-Schneider, 2004] is a first proposal for extending OWL by Horn clause rules. Other contributions in this field mostly stem from attempts to combine Description Logics, the theoretical underpinning of OWL, with rules.

1.3 Combining Rules and Ontologies

As we have seen, a key requirement of the layered architecture of the Semantic Web is to integrate the rules and the ontology layer. In particular, it is crucial to allow for building rules on top of ontologies, that is, for rule-based systems that use vocabulary specified in ontology knowledge bases. Another type of combination is to build ontologies on top of rules, which means that ontological definitions are supplemented by rules or imported from rules. In the first part of this thesis, we propose a combination of nonmonotonic logic programming under the answer-set semantics (and partly also under well-founded semantics) with Description Logics, focusing here on $\mathcal{SHIF}(\mathbf{D})$ and $\mathcal{SHOIN}(\mathbf{D})$, allowing for both strategies within a single logical framework. Importantly, we adopt the view of a loose interface between a logic program and a DL knowledge base, keeping both semantics strictly separated. This eases difficulties that arise when the two formalisms are tightly integrated, such as rule safety, infinite domains and undecidability. On the other hand it puts the responsibility of coupling two different semantics in a single framework in the hands of the user. This awareness is significant, recalling the fundamental differences between description logics as a fragment of first-order logic and logic programming. For instance, weak negation in rule bodies allows for nonmonotonic inferences, while ontologies in general behave in a monotonic way. Closely related is the difference between the closed-world assumption taken in logic programming, whereas reasoning in Description Logics happens in an open domain. These differences result in a semantic gap which is not straightforward to bridge. In this work we will closely examine these problems and show how we circumnavigated them.

As it turns out in practice, it is desirable for a rule formalism for the Semantic Web to be versatile enough not only to integrate ontological inferences, but also other kinds of external knowledge. The second part of the thesis presents a further extension of the answer-set semantics towards a more general approach of importing knowledge from arbitrary external sources while at the same time facilitate higher order reasoning, which lets one specify rules for meta-reasoning in a very concise and intuitive way. Here, we generalized the aforementioned interface, yet still keeping the loose integration paradigm.

1.4 Thesis Organization

The rest of the work is organized as follows: In Chapter 2 we introduce some preliminaries about the domain of logic programming in general and answer-set program in particular as well as the language OWL and its underlying theory of Description Logics. Chapter 3 describes a novel type of answer-set programs, so-called *dl-programs*, which provide means to interface ontological knowledge bases. We introduce the syntax and different semantics of this formalism and investigate possible ways of computation as well as the complexity of various reasoning tasks regarding dl-programs. In this part, we also consider possible application scenarios for dl-programs in the domain of the Semantic Web, making use of their nonmonotonic semantics. The chapter concludes with a presentation of a prototype reasoner for this type of programs. Chapter 4 introduces another type of extended answer-set programs, the so-called HEX-*programs*, which can be seen as a generalization of the

previous chapter's approach. Again, we describe syntax and semantics and survey possible application scenarios for HEX-programs. Also, we investigate their complexity and finally lay out various computation procedures to evaluate them. The description of an implemented solver for HEX-programs is given in Chapter 5. There, we also explain how to make use of the flexibility of this reasoning framework and create tailored extensions. Real-world applications of HEX-programs and the respective solver, demonstrating their usefulness and versatility, are presented in Chapter 6. Chapter 7 eventually concludes this thesis.

The results contained in this thesis have been published as refereed articles in the proceedings of several international conferences and workshops. We introduced our novel *dl-programs* first in the proceedings of the *"9th International Conference on of Knowledge Representation and (KR 2004)"* [Eiter et al., 2004a] (also available as technical report [Eiter et al., 2003]) and extended the answer-set semantics to well-founded semantics in the proceedings of the *"3rd International Workshop on Rules and Rule Markup Languages for the Semantic Web (RuleML 2004)"* [Eiter et al., 2004b]. A survey on this formalism was given as an invited paper in the *"Annals of Mathematics, Computing and Teleinformatics"* [Antoniou et al., 2004]. Computation and optimization techniques for dl-programs were presented in the proceedings of the *"12th International Conference on for Programming Artificial Intelligence Reasoning (LPAR 2004)"* [Eiter et al., 2005a]. The second extension to answer-set programming, our HEX-program formalism, was first published in the proceedings of the *"19th International Joint on Artificial Intelligence (IJCAI 2005)"* [Eiter et al., 2005d]. Reasoning prototypes for both languages were presented in the poster session of the *"4th International Semantic Web Conference (ISWC 2005)"* [Eiter et al., 2005b,c]. A survey on both reasoning frameworks was also given at the doctoral consortium of the *"21st International Conference Logic Programming (ICLP 2005)"* [Schindlauer, 2005]. A first account on the techniques implemented by the solver dlvhex was first given at the *"20th Workshop on Logic Programming and Systems (WLP 06)"* [Eiter et al., 2006b], which was succeeded by the more detailed descriptions in the proceedings of the *"11th International Workshop on Reasoning (NMR-2006, Answer Set Track)"* [Eiter et al., 2006c] and the proceedings of the *"3rd European Conference on Semantic Web (ESWC 2006)"* [Eiter et al., 2006f], where our paper received the Best Paper Award. A presentation of dlvhex and its possibilities was given in the informal proceedings of the *"Workshop on Applications of Logic Programming in the Semantic Web and Semantic Web Services (ALPSWS 2006)"* at the ICLP 2006 [Eiter et al., 2006d]. A survey on reasoning with dl- and HEX-programs appeared in the lecture notes of the *"Reasoning Web Second International Summer School"* [Eiter et al., 2006a]. A concrete example of using dlvhex in a Semantic Web reasoning context was presented at the poster track of the *"5th International Semantic Web Conference (ISWC 2006)"* [Polleres and Schindlauer, 2006]. A practical demonstration of dlvhex will be given at the *"IEEE Web Intelligence (WI 2006)"* [Eiter et al., 2006e].

Chapter 2

Preliminaries

In this chapter, we will outline the basics and principles of answer-set programming, its complexity and existing implementations. Moreover, we will introduce Description Logics as a formal base of various knowledge representation languages in the area of the Semantic Web.

2.1 Declarative Logic Programming

For central reasoning sub-tasks that can be identified when looking for solutions to the main problems of advanced information access such as priority handling and dealing with incomplete information, methods of *declarative logic programming* constitute a promising approach. In the logic programming model the programmer is responsible for specifying the basic logical relationships and does not specify the manner in which the inference rules are applied. An example shall illustrate the difference between a procedural and a logic-based approach.

Example 2.1.1 The following procedure represents knowledge about birds:

```
function bird(x): boolean;
    if x = 'tweety' then return true;
    else if x = 'sam' then return true;
    else if penguin(x) then return true;
    else return false;
```

The corresponding logic program would be

$$P = \{bird(tweety);\ bird(sam);\ bird(X) \leftarrow penguin(X);\}$$

◇

If it comes to extending the knowledge (e.g., by the fact that ostriches are birds), the logic program is obviously more modular and flexible. Furthermore, the reasoning capabilities of the procedural method are restricted to the information if an object is a bird. The implicit knowledge of the procedure cannot be used to answer explicit queries, like "which birds do you know?". The same question posed to the logic program would yield all known birds of the representation.

The most widespread tool for programming in logic is Prolog, which processes first-order predicate logic expressed by Horn clauses. Prolog fulfills many of the requirements for a high-level programming language, but it has some drawbacks when it comes to expressing

pure declarative semantics. Prolog is implemented as a sequential programming language by processing goals from left to right and selecting rules in textual order. This means that the rule order as well as the predicate order within a rule can influence the program's result. Furthermore Prolog provides extra-logical features to control the execution of the program. For example, the cut rule " ! " does not have a logical meaning. The dependence of the result on the rule order and non-logical predicates cause Prolog semantics to depart from the pure declarative meaning.

A different paradigm of logic programming that will be presented in this chapter is the *answer-set semantics* [Gelfond and Lifschitz, 1991], which is based on view of program statements as constraints on the solution of a given problem. Subsequently, each model of the program encodes a solution to the program itself.

2.2 Logic Programs under the Answer-Set Semantics

Answer-set programming has its roots in the stable model semantics of normal logic programs [Gelfond and Lifschitz, 1988] (also known as *general logic programs*), which are characterized by the occurrence of negation as failure. This kind of negation is closely related to Reiter's *Default Logic* [Reiter, 1980], hence it is also know as *default negation*. Since negation as failure is different from classical negation in propositional logic, Gelfond and Lifschitz proposed a logic programming approach that allows for both negations [Gelfond and Lifschitz, 1990]. Moreover, Gelfond and Lifschitz [1991] extended their semantics to disjunction in rule heads. Similar definitions for general logic programs and other classes of programs can be found in the literature (cf. e.g., [Lifschitz and Woo, 1992]). For an overview on other semantics for extended logic programs, see also [Dix, 1995].

2.2.1 Syntax

Let σ^{pred}, σ^{con} and σ^{var} be disjoint sets of predicate, constant, and variable symbols from a first-order vocabulary Φ, respectively, where σ^{var} is infinite and σ^{pred} and σ^{con} are countable. In accordance with common ASP solvers such as `DLV`, we assume that elements from σ^{con} and σ^{pred} are string constants that begin with a lowercase letter or are double-quoted, where elements from σ^{con} can also be integer numbers. Elements from σ^{var} begin with an uppercase letter. A *term* is either a constant or a variable. Given $p \in \sigma^{pred}$ an *atom* is defined as $p(t_1, \ldots, t_k)$, where k is called the arity of p and each $t_1, \ldots, t_k$ are terms. Atoms of arity $k = 0$ are called *propositional atoms*.

A *classical literal* (or simply *literal*) l is an atom p or a negated atom $\neg p$, where "$\neg$" is the symbol for true (classical) negation. Its *complementary* literal is $\neg p$ (resp., p). A *negation as failure literal* (or *NAF-literal*) is a literal l or a default-negated literal $not\, l$. Negation as failure is an extension to classical negation, denoting a fact as false if all attempts to prove it fail. Thus, $not\, L$ evaluates to *true* if it cannot be foundedly demonstrated that L is true, i.e., if either L is false or we do not know whether L is true or false.

A *rule* r is an expression of the form

$$a_1 \vee \ldots \vee a_n \leftarrow b_1, \ldots, b_k, not\, b_{k+1}, \ldots, not\, b_m\,, \qquad n \geq 0\,, m \geq k \geq 0\,, \tag{2.1}$$

where $a_1, \ldots, a_n, b_1, \ldots, b_m$ are classical literals. We say that $a_1, \ldots, a_n$ is the *head* of r, while the conjunction $b_1, \ldots, b_k$, $not\, b_{k+1}, \ldots, not\, b_m$ is the *body* of r, where $b_1, \ldots, b_k$ (resp., $not\, b_{k+1}, \ldots, not\, b_m$) is the *positive* (resp., *negative*) *body* of r. We use $H(r)$ to denote its head literals, and $B(r)$ to denote the set of all its body literals $B^+(r) \cup B^-(r)$, where $B^+(r) = \{b_1, \ldots, b_k\}$ and $B^-(r) = \{b_{k+1}, \ldots, b_m\}$. A rule r without head literals

(i.e., $n = 0$) is an *integrity constraint*. A rule r with exactly one head literal (i.e., $n = 1$) is a *normal rule*. If the body of r is empty (that is, $k = m = 0$), then r is a *fact*, and we often omit "$\leftarrow$".[1] An *extended disjunctive logic program* (EDLP, or simply *program*) P is a finite set of rules r of the form (2.1).

Programs without disjunction in the heads of rules are called *extended logic programs* (ELPs). A Program P without negation as failure, i.e., for all $r \in P$, $B^-(r) = \emptyset$ is called *positive logic program*. If, additionally, no strong negation occurs in P, i.e., the only form of negation is default negation in rule bodies, then P is called a *normal logic program* (NLP). The generalization of an NLP by allowing default negation in the heads of rules is called *generalized logic program* (GLP).

2.2.2 Semantics

The semantics of extended disjunctive logic programs is defined for variable-free programs. Thus, we first define the *ground instantiation* of a program that eliminates its variables.

The *Herbrand universe* of a program P, denoted HU_P, is the set of all constant symbols $C \subset \sigma^{con}$ appearing in P. If there is no such constant symbol, then $HU_P = \{c\}$, where c is an arbitrary constant symbol from Φ. As usual, terms, atoms, literals, rules, programs, etc. are *ground* iff they do not contain any variables. The *Herbrand base* of a program P, denoted HB_P, is the set of all ground (classical) literals that can be constructed from the predicate symbols appearing in P and the constant symbols in HU_P. A *ground instance* of a rule $r \in P$ is obtained from r by replacing every variable that occurs in r by a constant symbol from HU_P. We use $ground(P)$ to denote the set of all ground instances of rules in P.

The semantics for EDLPs is defined first for positive ground programs. A set of literals $X \subseteq HB_P$ is *consistent* iff $\{p, \neg p\} \not\subseteq X$ for every atom $p \in HB_P$. An *interpretation* I relative to a program P is a consistent subset of HB_P. We say that a set of literals S *satisfies* a rule r if $H(r) \cap S \neq \emptyset$ whenever $B^+(r) \subseteq S$ and $B^-(r) \cap S = \emptyset$. A *model* of a positive program P is an interpretation $I \subseteq HB_P$ such that I satisfies all rules in P. An *answer set* of a positive program P is the least model of P w.r.t. set inclusion.

To extend this definition to programs with negation as failure, we define the *Gelfond-Lifschitz transform* (also often called the *Gelfond-Lifschitz reduct*) of a program P relative to an interpretation $I \subseteq HB_P$, denoted P^I, as the ground positive program that is obtained from $ground(P)$ by

(i) deleting every rule r such that $B^-(r) \cap I \neq \emptyset$, and

(ii) deleting the negative body from every remaining rule.

An *answer set* of a program P is an interpretation $I \subseteq HB_P$ such that I is an answer set of P^I.

Example 2.2.1 Consider The following program P:

$$p \leftarrow not\, q.$$
$$q \leftarrow not\, p.$$

Let $I_1 = \{p\}$; then, $P^{I_1} = \{p \leftarrow\}$ with the unique model $\{p\}$ and thus I_1 is an answer set of P. Likewise, P has an answer set $\{q\}$. However, the empty set $\emptyset$ is not an answer set of P, since the respective reduct would be $\{p \leftarrow; q \leftarrow\}$ with the model $\{p, q\}$. $\diamond$

[1] In this thesis, we will use both forms "$a \leftarrow$" and "$a.$" to denote that a is a fact in a logic program.

Example 2.2.2 Let P be the following program:

$$\begin{aligned} &path(X,Y) \leftarrow arc(X,Y).\\ &path(X,Y) \leftarrow path(X,Z), arc(Z,Y).\\ &arc(a,b). \quad arc(b,c). \quad arc(b,d). \end{aligned}$$

The grounding of P contains these rules:

$$\begin{aligned} path(a,a) &\leftarrow arc(a,a).\\ path(a,b) &\leftarrow arc(a,b).\\ path(a,c) &\leftarrow arc(a,c).\\ path(a,a) &\leftarrow path(a,a), arc(a,a).\\ path(a,b) &\leftarrow path(a,b), arc(b,a).\\ path(a,c) &\leftarrow path(a,c), arc(c,c).\\ &\dots\\ arc(a,b). &\quad arc(b,c). \quad arc(b,d). \end{aligned}$$

From all models that satisfy these ground rules, the minimal model contains the following facts:

$$\{arc(a,b), arc(b,c), arc(b,d), path(a,b), path(b,c), path(b,d), path(a,c), path(a,d)\}$$

◇

A constraint is used to eliminate "unwanted" models from the result, since its head is implicitly assumed to be *false*. A model that satisfies the body of a constraint is hence discarded from the set of answer sets.

The main reasoning tasks that are associated with EDLPs under the answer-set semantics are the following:

- decide whether a given program P has an answer set;
- given a program P and a ground formula ϕ, decide whether ϕ holds in every (resp., some) answer set of P (*cautious* (resp., *brave*) *reasoning*);
- given a program P and an interpretation $I \subseteq HB_P$, decide whether I is an answer set of P (*answer set checking*); and
- compute the set of all answer sets of a given program P.

In the following, we will define specific syntactical restrictions with beneficial semantic properties on EDLPs.

Head-Cycle-Free Logic Programs

In [Ben-Eliyahu and Dechter, 1994] an alternative definition for answer sets is given for so called *head-cycle-free* EDLPs (HEDLPs). For that, we first have to define the dependency graph of an EDLP: The *dependency graph* of an EDLP P is a directed graph where each predicate occurring in P is a node and there is an edge from p to p' if there is a rule in P such that in $p \in H(r)$ and $p' \in B^+(r)$. We now say that p is head-cycle-free iff its dependency graph does not contain directed cycles that go through two literals occurring in the same rule head. For such HEDLPs Ben-Eliyahu and Dechter [1994] showed the following:

Theorem 2.2.1 (*cf. [Ben-Eliyahu and Dechter, 1994]*) *Given a HEDLP P, a consistent set $S \subseteq Lit(P)$ is an answer set iff*

1. *S satisfies each rule in P, and*
2. *there is a function $\phi : Lit(P) \mapsto \mathbb{N}^+$ such that for each literal l in S there is a rule r in P with*
   (a) $B^+(r) \subseteq S$
   (b) $B^-(r) \cap S = \emptyset$
   (c) $l \in H(r)$
   (d) $S \cap (H(r) \setminus \{l\}) = \emptyset$
   (e) $\phi(l') < \phi(l)$ *for each* $l' \in B^+(r)$

The key essence of this theorem is that HEDLPs (in contrast to EDLPs in general) allow for a leveled evaluation of the logic program if we know the function ϕ. Later results by Babovich et al. [2000] which generalize Fages' theorem [Fages, 1994] resemble this theorem resulting in similar alternative definitions of answer sets for normal logic programs. Head-cycle-free disjunction in fact does not increase the expressive power compared with normal logic programs, as head-cycle-free negation can be shifted to the rule bodies by a semantically equivalent rewriting where any disjunctive rule:

$$h_1 \vee \ldots \vee h_l \leftarrow Body.$$

is substituted by l normal rules:

$$\begin{aligned} h_1 &\leftarrow \ not\, h_2, not\, h_3, \ldots, not\, h_l, Body. \\ h_2 &\leftarrow \ not\, h_1, not\, h_3, \ldots, not\, h_l, Body. \\ &\vdots \\ h_l &\leftarrow \ not\, h_1, not\, h_2, \ldots, not\, h_{l-1}, Body. \end{aligned}$$

For programs with head-cycles, this rewriting is not possible which can be easily shown by the following example:

Example 2.2.3 Let P_2 be the following simple DLP:

$$p \leftarrow q. \quad q \leftarrow p. \quad p \vee q.$$

Obviously, P_2 has the single answer set $S = \{p, q\}$. On the other hand, when substituting the last rule with the pair of rules $\{p \leftarrow not\, q.\ q \leftarrow not\, p.\}$, the resulting program has no answer sets at all. ◇

Stratified Logic Programs

An even stronger restriction than head-cycle-freeness is stratification. The concept of stratification was introduced for logic programs independently by Apt et al. [1988] and by van Gelder [1988]. Przymusinski generalized it to constraint-free DLPs [Przymusinski, 1988, 1991].

We say that a constraint-free DLP P is *stratified* iff there exists a function $Strat : Lit(P) \mapsto \mathbb{N}^+$ such that for every rule r of the form (2.1) there exists a $c \in \mathbb{N}$ with

1. $Strat(h) = c$ for all $h \in H(r)$
2. $Strat(b) \leq c$ for all $b \in B^+(r)$
3. $Strat(b) < c$ for all $b \in N^-(r)$

It is well-known that such a stratification *Strat* can efficiently be found, if existent. In particular, positive programs are always stratified. Note that stratification does not imply head-cycle-freeness or vice versa. However, stratified programs also allow for an even more efficient evaluation. In case P is free of integrity constraints, stratified programs always have at least one answer set. Note that EDLPs are not considered, since extended programs with classical negation always contain "implicit" integrity constraints $\leftarrow a, \neg a.$ for any complementary pair of literals.

2.2.3 Available Systems: Restrictions and Extensions

Among the available systems for computing answer sets of logic programs the two most successful over the past years have been DLV [Eiter et al., 1998, 2000a, Leone et al., 2002] and SMODELS [Niemelä, 1999, Simons et al., 2002] which allow for efficient declarative problem solving.

DLV

The DLV system[2] has been developed for several years as joint work of the University of Calabria and Vienna University of Technology and is still actively maintained. It is an efficient engine for computing answer sets accepting as core input language logic programs as defined above which fulfill the following *safety restriction* (cf. [Ullman, 1989]):

Definition 2.2.1 *A rule r of the form (2.1) is called* safe *if every variable X occurring in literals in $H(r) \cup B^-(r)$ also occurs in at least one literal $B^+(r)$. A logic Program P is* safe *if all of its rules are safe.*

Note that this restriction is only syntactical but does not really affect the expressive power of the language in any way. We refer for instance to [Leone et al., 2006] for a detailed discussion.

Weak Constraints Furthermore, DLV extends the logic programs by so-called *weak constraints*, cf. [Buccafurri et al., 1997, 2000]:

Definition 2.2.2 *A* weak constraint *is a construct*

$$:\sim\ b_1, \ldots, b_k,\ not\ b_{k+1}, \ldots,\ not\ b_m.\ [w : l] \tag{2.2}$$

where w (weight) and l (level) are integer constants or variables occurring in $b_1, \ldots, b_k$ and all b_i are classical literals. If l is not specified, it defaults to 1, and we can just write $[w :]$. The body of a weak constraint c, $B(c)$ is defined as for (2.1).

The level l intuitively allows to specify a priority layer after the colon, where 1 is the lowest priority.

The syntactical safety restriction from above is extended to weak constraints as follows: A weak constraint c is *safe* if in addition to the conditions above whenever w (or l, resp.) is a variable it has to occur in at least one literal $B^+(c)$.

[2] http://www.dlvsystem.com

An *extended disjunctive logic program with weak constraints* (EDLP^w) is then a finite set of rules, constraints and weak constraints.

The *ground instantiation* for an EDLP^w is defined like in Subsection 2.2.2 with the obvious extension to weak constraints. Furthermore, we impose another syntactical restriction on weak constraints related to safety mentioned above: A weak constraint c is only admissible, if all possible weights and levels are integers. Thus, if either w or l is a variable, then P must guarantee that w, l can only be bound to integers. This restriction can easily be checked (for instance during grounding), which is done by `DLV`.

The answer sets of an EDLP^w P without weak constraints are defined as above. The answer sets of a program P with weak constraints are defined by selection of so called *optimal answer sets* from the answer sets S of the weak-constraint free part P' of P (referred to as *candidate answer sets*).

Again, we will define the semantics of optimal answer sets in terms of the ground instantiation of a program. A weak constraint c of the form (2.2) is *violated*, if it is satisfied with respect to the candidate answer set S, i.e., $\{b_1, \ldots, b_k\} \subseteq S$ and $\{b_{k+1}, \ldots, b_m\} \cap S = \emptyset$. We are interested in those answer sets with minimal weights of the violated weak constraints in the highest priority level. Among those, the ones with the minimal weight on the next lower level are considered as optimal, etc. This can be expressed by an objective function $H^P(A)$ for a program P with weak constraints WC and an answer set A as follows, where f_P is an auxiliary function that guarantees the prioritization of levels over weights (cf. [Leone et al., 2006]):

$$\begin{aligned} f_P(1) &= 1, \\ f_P(n) &= f_P(n-1) \cdot |WC(P)| \cdot w^P_{max} + 1, \quad n > 1, \\ H^P(A) &= \Sigma_{i=1}^{l^P_{max}} (f_P(i) \cdot \Sigma_{w \in N_i^P(A)} \mathit{weight}(w)), \end{aligned}$$

where $|WC(P)|$ denotes the number of weak constraints in P, and w^P_{max} and l^P_{max} denote the maximum weight and maximum level over the weak constraints in P, respectively. $N_i^P(A)$ denotes the set of weak constraints in level i that are violated by A, and $\mathit{weight}(w)$ denotes the weight of the weak constraint w.

Example 2.2.4 Let us consider the following program P:

$p \vee q.$
$\neg q \leftarrow p.$
$:\sim p.\ [3:1] \quad :\sim q.\ [1:2]$

This program has the two answer sets $A_1 = \{p, \neg q\}$ and $A_2 = \{q\}$. Having defined two levels by weak constraints, $f(1) = 1$ and $f(2) = 7$ and thus $H^P(A_1) = 3$ and $H^P(A_2) = 7$. It follows that A_2 is the single optimal answer set of P. ◇

Example 2.2.5 The following program computes the minimum spanning trees of a weighed directed graph.

$\mathit{root}(a).$
$\mathit{node}(a).\ \mathit{node}(b).\ \mathit{node}(c).\ \mathit{node}(d).\ \mathit{node}(e).$
$\mathit{edge}(a,b,4).\ \mathit{edge}(a,c,3).\ \mathit{edge}(c,b,2).\ \mathit{edge}(c,d,3).\ \mathit{edge}(b,e,4).\ \mathit{edge}(d,e,5).$
$\mathit{in\_tree}(X,Y,C) \vee \mathit{out\_tree}(X,Y) \leftarrow \mathit{edge}(X,Y,C), \mathit{reached}(X).$

$\leftarrow root(X), in\_tree(\_, X, C).$
$\leftarrow in\_tree(X, Y, C), in\_tree(Z, Y, C), X \neq Z.$
$reached(X) \leftarrow root(X).$
$reached(Y) \leftarrow reached(X), in\_tree(X, Y, C).$
$\leftarrow node(X), not\ reached(X).$
$:\sim in\_tree(X, Y, C).\ [C : 1]$

The single optimal answer set of this program (modulo the original facts) is:

$\{reached(a), reached(b), reached(c), reached(d), reached(e),$
$in\_tree(a, c, 3), in\_tree(b, e, 4), in\_tree(c, b, 2), in\_tree(c, d, 3),$
$out\_tree(a, b), out\_tree(d, e)\}$

with the cost $[12 : 1]$. ◇

Built-In Predicates The built-in predicates "`A` $<$ `B`", "`A` $<=$ `B`", "`A` $>$ `B`", "`A` $>=$ `B`", and "`A` $!=$ `B`" with the obvious meanings of less-than, less-or-equal, greater-than, greater-or-equal, and inequality for strings and numbers can be used in the positive bodies of `DLV` rules and constraints.

`DLV` currently does not support full arithmetics but provides some built-in predicates, which can be used to "emulate" range restricted integer arithmetics: the arithmetic built-ins "`A` $=$ `B` $+$ `C`" and "`A` $=$ `B` $*$ `C`" which stand for integer addition and multiplication, and the predicate "`#int(X)`" which which holds for all nonnegative integers (up to a user-defined limit).

Aggregates Furthermore, borrowing from database query languages, `DLV` has been extended by *aggregate predicates* [Dell'Armi et al., 2003]. Aggregate predicates allow to express properties over a set of elements, such as *sum* or *count*. They can occur in the bodies of rules and constraints, possibly negated using negation-as-failure. Aggregates often allow clean and concise problem encodings by minimizing the use of auxiliary predicates and recursive programs, and and foster the depiction of problems in a more natural way. From the point of efficiency, encodings using aggregates often outperform those without, reducing the size of the ground instantiation of the program.

Aggregate predicates operate over so-called *symbolic sets*. A symbolic set is a pair $\{V : C\}$, where V is a list of variables and C is a conjunction of literals with V among their arguments. The semantics of a symbolic set w.r.t. an interpretation I contains all ground pairs $\{V' : C'\}$ such that $I \models C'$. The aggregate function, such as `#count`, `#sum`, `#min`, and `#max`, is then applied to the symbolic set. The returned value is compared to the guards, specified in the entire aggregate predicate. For instance, the predicate

$$0 \leq \#\texttt{count}\{Y : person(X), ownsCar(X, Y)\} \leq 1$$

is true for all interpretations where no person owns more than one car.

SMODELS and GNT

SMODELS [Niemelä, 1999, Simons et al., 2002][3] allows for the computation of answer sets for normal logic programs. However, there is an extended prototype version for the evaluation of disjunctive logic programs as well, called GNT [Janhunen et al., 2000].[4]

[3] http://www.tcs.hut.fi/Software/smodels/
[4] http://www.tcs.hut.fi/Software/gnt/

Syntactically, SMODELS imposes an even stronger restriction than rule safety in DLV by demanding that any variable in a rule of the form (2.1) is bounded to a so-called *domain predicate* $d \in B^+(r)$ which is, intuitively, a predicate that is defined only positively (cf. [Niemelä, 1999] for details). Again, this restriction does not affect the expressive power of the language itself, but in some cases the weaker safety restriction of DLV allows for more concise problem encodings.

SMODELS also allows for model optimization, but the syntactic and semantic concept here slightly differs from weak constraints in DLV: SMODELS supports another extension to pure answer-set programming allowing to minimize over sets of predicates (cf. [Simons et al., 2002] for details) by adding statements of the form:

$$\texttt{minimize}\{b_1 = w_1, \ldots, b_m = w_m, not\, b_{m+1} = w_{m+1}, \ldots, not\, b_n = w_n\}.$$

where $b_1, \ldots, b_n$ are ground literals and $w_1, \ldots, w_n$ are constants. Here, similarly to weak constraints, an answer set S of a program P is considered to be optimal if

$$\mathsf{cost}_P(S) = \sum\{w \mid ((b_i = w) \in min \wedge b_i \in S) \vee (not\, b_i = w) \in min \wedge b_i \notin S)\}$$

is minimal, where *min* is the union of all **minimize** statements. If there are more than one **minimize** statements SMODELS considers them in fixed order, the last one being the strongest, similar to levels of DLV weak constraints, but missing full declarativity in some sense (since rule order has a semantic impact here).

For **minimize** statements with variables, SMODELS offers the following shorter notation:

$$\begin{aligned}\texttt{minimize}[\, & a_1(\vec{X}_1) : b(\vec{Y}_1) = C_1, \ldots, a_m(\vec{X}_m) : b(\vec{Y}_m) = C_m, \\ & not\, a_{m+1}(\vec{X}_{m+1}) : b(\vec{Y}_{m+1}) = C_{m+1}, \ldots, not\, a_n(\vec{X}_n) : b(\vec{Y}_n) = C_n].\end{aligned}$$

where $\vec{X}_i$, $\vec{Y}_i$ are lists of variables or constants, and all variables in $\vec{X}_i$ have to occur in $\vec{Y}_i$. C_i is either a variable from $\vec{Y}_i$ or a constant, and b_i is a domain predicate, for $i \in \{1, \ldots, n\}$.

During model computation SMODELS does not compute only optimal answer sets, but first evaluates an arbitrary model and then incrementally only returns "better" answer sets, such that the last answer set found by SMODELS is the optimal one. As an additional feature SMODELS also provides a dual **maximize** statement with the obvious semantics. Moreover, similar to DLV, SMODELS allows for a restricted form of integer arithmetics and lexicographic comparison predicates.

2.3 Computational Complexity

We will now review the most important problem classes for the computational complexity of the problems addressed in the course of this work. Furthermore, we will review some results on the computational complexity of answer-set programming.

2.3.1 Complexity Classes

We assume that the reader is familiar with the concept of Turing Machines and basic notions of complexity theory, such as problem reductions and completeness; see e.g., [Papadimitriou, 1994] and references therein. We recall that P, resp. NP, is the class of decision problems (i.e., problems where the answer is "yes" or "no") computable on a deterministic, resp. nondeterministic, Turing Machine in polynomial time. Further, PSPACE is the class of problems computable on deterministic Turing Machines with polynomial storage space.

The classes Σ_k^P (resp. Π_k^P, Δ_k^P), $k \geq 0$ of the so called Polynomial Hierarchy $\mathsf{PH} = \bigcup_{k \geq 0} \Sigma_k^P$ are defined by $\Sigma_0^P = \Pi_0^P = \Delta_0^P = \mathsf{P}$ and $\Sigma_k^P = \mathsf{NP}^{\Sigma_{k-1}^P}$ (resp. $\Pi_k^P = \text{co-}\Sigma_k^P$, $\Delta_k^P = \mathsf{P}^{\Sigma_{k-1}^P}$), for $k \geq 1$. The latter model nondeterministic polynomial-time computation with an oracle for problems in Σ_{k-1}^P. Here, co- stands for the class of complementary problems. In particular, $\Sigma_1^P = \mathsf{NP}$, $\Pi_1^P = \mathsf{coNP}$ and $\Delta_2^P = \mathsf{P}^{\mathsf{NP}}$.

Furthermore, $\mathsf{DP} = \{L \cap L' \mid L \in \mathsf{NP}, L' \in \mathsf{coNP}\}$ is the logical "conjunction" of NP and coNP.[5] Finally, NEXPTIME and NEXPSPACE denote the class of problems decidable by nondeterministic Turing machines in exponential time, resp. space.

We recall that $\mathsf{NP} \subseteq \mathsf{DP} \subseteq \mathsf{PH} \subseteq \mathsf{PSPACE} = \mathsf{NPSPACE} \subseteq \mathsf{NEXPTIME}$ holds, where NPSPACE is the nondeterministic analog of PSPACE. It is generally believed that these inclusions are strict, and that PH is a true hierarchy of problems with increasing difficulty. Note that NEXPTIME-complete problems are *provably intractable*, i.e., exponential lower bounds can be proved, while no such proofs for problems in PH or PSPACE are known today.

While many interesting problems are decision problems, computing answer sets are *search problems*, where for each problem instance I a (possibly empty) finite set $S(I)$ of solutions exists. To solve such a problem, a (possibly nondeterministic) algorithm must compute the alternative solutions from this set in its computation branches, if $S(I)$ is not empty. More precisely, search problems are solved by transducers, i.e., Turing machines equipped with an output tape. If the machine halts in an accepting state, then the content of the output tape is the result of the computation. Observe that a nondeterministic machine computes a (partial) multi-valued function.

As an analog to NP, the class NPMV contains those search problems where $S(I)$ can be computed by a nondeterministic Turing machine in polynomial time; for a precise definition, see [Selman, 1994]. In analogy to Σ_{i+1}^P, by $\Sigma_{i+1}^P\text{MV} = \mathsf{NPMV}^{\Sigma_i^P}$, $i \geq 0$, we denote the generalization of NPMV where the machine has access to a Σ_i^P oracle.

Analogs to the classes P and Δ_{i+1}^P, $i \geq 0$, are given by the classes FP and $\text{F}\Delta_{i+1}^P$, $i \geq 0$, which contain the partial single-valued functions (that is, $|S(I)| \leq 1$ for each problem instance I) computable in polynomial time possibly using a Σ_i^P oracle. We say, abusing terminology, that a search problem A is in FP (or $\text{F}\Delta_{i+1}^P$), if there is a partial (single-valued) function $f \in \mathsf{FP}$ (or $f \in \text{F}\Delta_{i+1}^P$) such that $f(I) \in S(I)$ and $f(I)$ is undefined iff $S(I) = \emptyset$. For example, computing a satisfying assignment for a propositional CNF (FSAT) and computing an optimal tour in the Traveling Salesperson Problem (TSP) are in $\text{F}\Delta_2^P$ under this view, cf. [Papadimitriou, 1994].

A partial function f is polynomial-time reducible to another partial function g, if there are polynomial-time computable functions h_1 and h_2 such that $f(I) = h_2(I, g(h_1(I)))$ for all I and $g(h_1(I))$ is defined whenever $f(I)$ is defined. Hardness and completeness are defined as usual.

2.3.2 Complexity of Logic Programming

We will now consider the following problems: Given a logic program P, decide whether P has a model under the answer-set semantics.

We restrict ourselves to finite propositional, i.e. ground, (function-free) EDLPs as defined above. Non-ground programs are not considered as grounding might already be exponential and deciding answer set existence thus becomes provably intractable even for simple positive normal programs (cf. [Immerman, 1987, Vardi, 1982, Dantsin et al., 2001]).

[5] Note that DP is *not* NP ∩ coNP (cf. [Papadimitriou, 1994]).

Furthermore, note that when allowing function symbols most of the problems outlined in this section become undecidable in general which is basically explained by the undecidability of first-order logic.

Theorem 2.3.1 (*cf. [Eiter et al., 1997, Dantsin et al., 2001]*) *Deciding whether a propositional EDLP has an answer set is Σ_2^P-complete. Computing such an answer set is Σ_2^PMV-complete.*

Proof. *Membership:* Given a program P, an answer set S can be guessed and checked in polynomial time by an NP oracle: In particular, the reduct P^S can clearly be computed in polynomial time. Since P^S is a positive program, its answer sets coincide with its minimal models. Testing whether S is a minimal model is in coNP (cf. [Cadoli, 1992]) and therefore decidable in polynomial time by a single call to an NP oracle.

Hardness: For the hardness proof we will review an encoding of deciding the satisfiability of a Quantified Boolean Formula (QBF)

$$F = \exists x_1 \ldots \exists x_m \forall y_1 \ldots \forall y_n \Phi$$

with one quantifier alternation in an answer-set program, which is a well known reference problem hard for the class Σ_2^P. Here, $\Phi = c_1 \vee \ldots \vee c_k$ is a propositional formula over $x_1, \ldots, x_m, y_1, \ldots, y_n$ in disjunctive normal form, i.e. each $c_i = a_{i,1} \wedge \ldots \wedge a_{i,l_i}$ and $|a_{i,j}| \in \{x_1, \ldots, x_m, y_1, \ldots, y_n\}$. Satisfiability is here defined as the existence of an assignment to the variables $x_1, \ldots, x_m$ which witness that F evaluates to true.

We will now present an encoding of this formula as an answer-set program P_{QBF} such that P_{QBF} has an answer set if and only if F is satisfiable:

$$
\begin{aligned}
&x_1 \vee nx_1. \;\; \ldots \;\; x_m \vee nx_m.\\
&y_1 \vee ny_1. \;\; \ldots \;\; y_n \vee ny_n.\\
&sat \leftarrow a_{1,1}, \ldots, a_{1,l_1}.\\
&\quad \vdots\\
&sat \leftarrow a_{k,1}, \ldots, a_{k,l_k}.\\
&y_1 \leftarrow sat. \quad ny_1 \leftarrow sat. \quad \ldots \quad y_n \leftarrow sat. \quad ny_n \leftarrow sat.\\
&\leftarrow not\ sat.
\end{aligned}
$$

This encoding is maybe not intuitive at first sight, but in principle can be explained as follows: For any assignment guessed for $x_1, \ldots, x_m$, *sat* will *not* be derived if there is a bad assignment for $y_1, \ldots, y_n$ such that all clauses are unsatisfied. However, since all answer sets not containing *sat* are invalidated by the last constraint, only those assignments for $x_1, \ldots, x_m$ "survive" which do not allow for such a bad assignment for $y_1, \ldots, y_n$. This can be argued by minimality of answer sets together with rules in the one but last line, which "saturate" any good assignment for $y_1, \ldots, y_n$ to an answer set uniquely determined by $x_1, \ldots, x_m$. Note that this encoding does not only represent the satisfiability problem, but moreover the answer sets of P_{QBF} uniquely encode the valid assignments for variables $x_1, \ldots, x_m$, which proofs hardness for Σ_2^P and Σ_2^PMV, respectively. For the details of this encoding we refer to [Eiter and Gottlob, 1995]. □

This result shows that the computational power of answer-set solvers such as DLV and GNT which support full disjunctive logic programming is indeed higher than solvers for propositional Satisfiability (SAT) (unless the PH collapses). Hence, using answer-set programming we can encode and solve hard problems not expressible as a simple propositional logic formula in polynomial time.

Note that the complexity boils down to lower complexity classes as soon as we impose specific syntactical restrictions:

Proposition 2.3.2 (*cf. [Ben-Eliyahu and Dechter, 1994, Fages, 1994]*) *For head-cycle free (resp. normal) logic programs deciding answer set existence is* NP*-complete.*

The essence of this result is that head-cycle free and normal logic programs can intuitively be evaluated by guessing an order of rule evaluation. Moreover, it states that SMODELS without its disjunctive extension GNT, i.e., answer-set solvers which only accept normal or head-cycle-free logic programs cover the same class of problems as SAT solvers.

Furthermore, for *stratified* (especially positive) DLPs the answer sets correspond to the minimal models of a program, i.e., answer set existence is trivial (cf. [Przymusinski, 1988, 1991]). For non-disjunctive programs this model even is unique and by well-known results computable in polynomial time.

Finally, as for *optimal answer sets* w.r.t. to weak constraints in DLV, we know from [Buccafurri et al., 2000] that deciding whether a query q is true in some optimal answer set of an EDLP^w P is Δ_3^P-complete and Δ_2^P-complete for head-cycle free programs. The respective class for computing such an optimal answer set is $\mathrm{F}\Delta_3^P$, and $\mathrm{F}\Delta_2^P$ for head-cycle free programs. These results equally apply to minimization in SMODELS with minor adaptions.

2.4 Description Logics

The line of research that eventually led to the term "Description Logics" has its roots in the 1970s, when a crucial distinction between two directions of knowledge representation developments emerged: Logic-based formalisms, using predicate calculus to draw implicit conclusions from explicitly represented knowledge, and non-logic-based representations, which built mostly on cognitive notions and were created from specific methods of thinking regarding problem solving. Evidently, the former approach was more of general-purpose than the latter, adopting variants of first-order predicate calculus, such that the process of reasoning amounts to verifying logical consequence. In the non-logical formalisms, inferring knowledge was achieved by manipulating ad-hoc data structures. This approach led to the development of *frames* and *semantic networks*, where a network structure represents sets of individuals and their relationships.

Soon, it became clear that a major deficiency of such network-based systems is their lacking of a precise semantic characterization. Nevertheless, it was shown that frames can be given a semantics which is based on first-order logic, by unary predicates representing sets of individuals and binary predicates representing relationships between them [Hayes, 1979]. From this viewpoint it appeared that frames and semantic networks require only a fragment of first-order predicate logic [Brachman and Levesque, 1985], making it unnecessary to rely on full first-order theorem provers. Instead, specialized reasoning techniques were more suitable to carry out reasoning in these formalisms, which were then subsumed under the term *terminological systems* and later *Description Logics*.

The basic building blocks of a network representation, as shown in Figure 2.1, are *nodes* and *links*. Nodes denote concepts, i.e., classes of individuals, while links characterize relationships between concepts. A specific type of link is the "IS-A" relationship, which defines a hierarchy between classes, such as the link between Student and Person in the example figure. Such subclass-relationships facilitate the inheritance of properties. Naturally, implicit relationships between concepts exist, such as the fact that PhD Student is also a subclass of Person. Apart from such *atomic* concepts, classes can be specified by

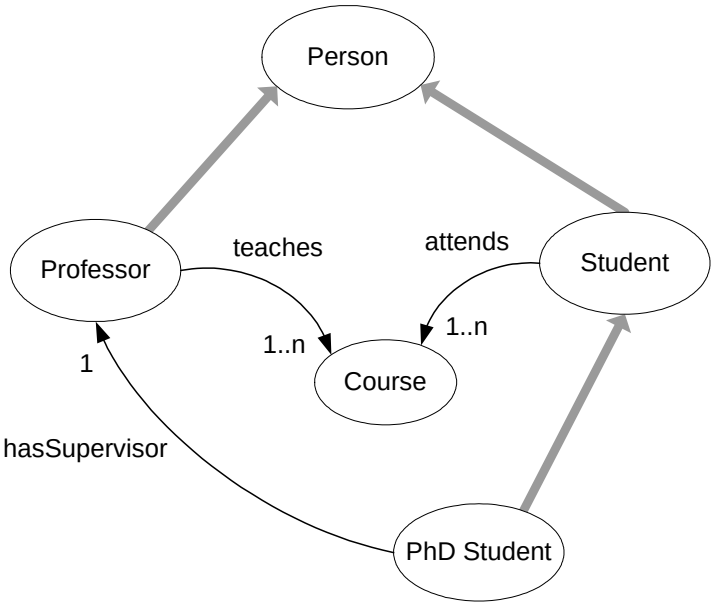


Figure 2.1: An example network.

means of union and intersection of other classes. A specific feature of Description Logics is the definition of relationships other than IS-A, such as hasSupervisor in the example, which are commonly denoted as *roles*. Moreover, they allow for so called *value-restrictions*, i.e., stating conditions like "each PhD student must have exactly one supervisor".

Research in this area has generated a variety of different Description Logics formalisms, each with specific expressivity and complexity, offering a distinct set of language constructs. Two of them, which play a major role in the theoretical foundations of knowledge representation on the Semantic Web, will be presented in the subsequent sections.

For an excellent introduction to Description Logics, we refer to [Baader et al., 2003].

2.4.1 $\mathcal{SHOIN}(\mathbf{D})$ and $\mathcal{SHIF}(\mathbf{D})$

In this section, we recall the foundations of two Description Logics, which are the underpinning of Semantic Web ontology languages and basis of the novel type of answer-set programs that will be presented in Chapter 3.

The naming of specific Description Logics languages usually corresponds to the constructors they provide (in addition to the basic ones like concept union, concept disjunction, etc.). In the case of $\mathcal{SHOIN}(\mathbf{D})$ these are:

- $\mathcal{S}$ Role transitivity.
- $\mathcal{H}$ Role hierarchy.
- $\mathcal{O}$ Nominals ("one-of"-constructor).
- $\mathcal{I}$ Role inverses.
- $\mathcal{N}$ Unqualified number restrictions.
- D Datatypes.

The logic $\mathcal{SHIF}(\mathrm{D})$ is slightly less expressive:

- $\mathcal{S}$ Role transitivity.

$\mathcal{H}$ Role hierarchy.
$\mathcal{I}$ Role inverses.
$\mathcal{F}$ Functionality.
D Datatypes.

Here, functionality stands for the specific number restriction $\leq 1R$, which is subsumed by the unqualified number restrictions of $\mathcal{SHOIN}(\mathbf{D})$. Evidently, $\mathcal{SHOIN}(\mathbf{D})$ is a restriction of $\mathcal{SHOIN}(\mathbf{D})$, which is closely related to the Description Logic $\mathcal{SHOQ}(\mathbf{D})$ [Horrocks and Sattler, 2001] The reason for presenting these two Description Logics here is that they are the formal counterparts of sublanguages of the Web Ontology Language OWL which will be discussed at the end of this chapter.

2.4.2 Syntax

We first describe the syntax of the Description Logic $\mathcal{SHOIN}(\mathbf{D})$. We assume a set $\mathbf{D}$ of *elementary datatypes*. Every $d \in \mathbf{D}$ is associated with a set of *data values*, called the *domain* of d, denoted $\mathrm{dom}(d)$. We use $\mathrm{dom}(\mathbf{D})$ to denote $\bigcup_{d\in\mathbf{D}} \mathrm{dom}(d)$. A *datatype* is either an element of $\mathbf{D}$ or a subset of $\mathrm{dom}(\mathbf{D})$ (called *datatype oneOf*). Let $\mathbf{A}$, $\mathbf{R}_A$, $\mathbf{R}_D$, and $\mathbf{I}$ be nonempty finite and pairwise disjoint sets of *atomic concepts*, *abstract roles*, *datatype roles*, and *individuals*, respectively. We use $\mathbf{R}_A^-$ to denote the set of all inverses R^- of abstract roles $R \in \mathbf{R}_A$.

A *role* is an element of $\mathbf{R}_A \cup \mathbf{R}_A^- \cup \mathbf{R}_D$. *Concepts* are inductively defined as follows. Every atomic concept from $\mathbf{A}$ is a concept. If $o_1, o_2, \ldots$ are individuals from $\mathbf{I}$, then $\{o_1, o_2, \ldots\}$ is a concept (called *oneOf*). If C and D are concepts, then also $(C \sqcap D)$, $(C \sqcup D)$, and $\neg C$ (called *conjunction*, *disjunction*, and *negation*, respectively). If C is a concept, R is a role from $\mathbf{R}_A \cup \mathbf{R}_A^-$, and n is a nonnegative integer, then $\exists R.C$, $\forall R.C$, $\geq nR$, and $\leq nR$ are concepts (called *exists*, *value*, *atleast*, and *atmost restriction*, respectively). If U is a datatype role from $\mathbf{R}_D$, n is a nonnegative integer, and d is a datatype from $\mathbf{D}$, then $\exists U.d$, $\forall U.d$, $\geq nU$, and $\leq nU$ are concepts (called *datatype exists*, *value*, *atleast*, and *atmost restriction*, respectively). We write $\top$ (resp., $\bot$) to abbreviate $C \sqcup \neg C$ (resp., $C \sqcap \neg C$), and we eliminate parentheses as usual.

Axioms are expressions of the following forms:

(1) $C \sqsubseteq D$, where C and D are concepts (*concept inclusion*);

(2) $R \sqsubseteq S$, where either $R, S \in \mathbf{R}_A$ or $R, S \in \mathbf{R}_D$ (*role inclusion*);

(3) $\mathrm{Trans}(R)$, where $R \in \mathbf{R}_A$ (*transitivity*);

(4) $C(a)$, where C is a concept and $a \in \mathbf{I}$ (*concept membership*);

(5) $R(a, b)$ (resp., $U(a, v)$), where $R \in \mathbf{R}_A$ (resp., $U \in \mathbf{R}_D$) and $a, b \in \mathbf{I}$ (resp., $a \in \mathbf{I}$ and $v \in \mathrm{dom}(\mathbf{D})$) (*role membership axiom*); and

(6) $a = b$ (resp., $a \neq b$), where $a, b \in \mathbf{I}$ (*equality* resp. *inequality*).

A *knowledge base* DL is a finite set of axioms.

For decidability reasons, number restrictions in a knowledge base DL are restricted to simple abstract roles [Horrocks et al., 1999]: A role R is called *simple* w.r.t. DL iff for each role S such that $S \sqsubseteq^\star R$, it holds that $\mathrm{Trans}(S) \notin DL$, where $\sqsubseteq^\star$ is the transitive and reflexive closure of $\sqsubseteq$ on DL, that is, $S \sqsubseteq^\star R$ iff either (i) $S \sqsubseteq R$ is in DL, or (ii) $S = R$, or (iii) $S \sqsubseteq^\star Q$ and $Q \sqsubseteq^\star R$, for some role Q.

The syntax of $\mathcal{SHIF}(\mathbf{D})$ is as the above syntax of $\mathcal{SHOIN}(\mathbf{D})$, but without the oneOf constructor and with the atleast and atmost constructors limited to 0 and 1.

2.4.3 Semantics

The meaning of a Description Logic is usually defined by a model theoretic semantics, relating the syntax of the language to the intended models of the domain. An *interpretation* $\mathcal{I} = (\Delta, \cdot^{\mathcal{I}})$ w.r.t. $\mathbf{D}$ consists of a nonempty (*abstract*) *domain* Δ, which is disjoint from the datatype domain $\mathrm{dom}(\mathbf{D})$, and a mapping $\cdot^{\mathcal{I}}$ that assigns to each atomic concept from $\mathbf{A}$ a subset of Δ, to each individual $o \in \mathbf{I}$ an element of Δ, to each abstract role from $\mathbf{R}_A$ a subset of $\Delta \times \Delta$, and to each datatype role from $\mathbf{R}_D$ a subset of $\Delta \times \mathrm{dom}(\mathbf{D})$. The mapping $\cdot^{\mathcal{I}}$ is extended by induction to all concepts and roles as follows (where $\sharp S$ denotes the cardinality of a set S):

- $(\{o_1, o_2, \ldots\})^{\mathcal{I}} = \{o_1^{\mathcal{I}}, o_2^{\mathcal{I}}, \ldots\}$, $(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}$, $(C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}}$, and $(\neg C)^{\mathcal{I}} = \Delta \setminus C^{\mathcal{I}}$,
- $(\exists R.C)^{\mathcal{I}} = \{x \in \Delta \mid \exists y\colon (x, y) \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$,
- $(\forall R.C)^{\mathcal{I}} = \{x \in \Delta \mid \forall y\colon (x, y) \in R^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}}\}$,
- $(\geq n R)^{\mathcal{I}} = \{x \in \Delta \mid \sharp(\{y \mid (x, y) \in R^{\mathcal{I}}\}) \geq n\}$,
- $(\leq n R)^{\mathcal{I}} = \{x \in \Delta \mid \sharp(\{y \mid (x, y) \in R^{\mathcal{I}}\}) \leq n\}$,
- $(\exists U.d)^{\mathcal{I}} = \{x \in \Delta \mid \exists y\colon (x, y) \in U^{\mathcal{I}} \wedge y \in \mathrm{dom}(d)\}$,
- $(\forall U.d)^{\mathcal{I}} = \{x \in \Delta \mid \forall y\colon (x, y) \in U^{\mathcal{I}} \rightarrow y \in \mathrm{dom}(d)\}$.
- $(\geq n U)^{\mathcal{I}} = \{x \in \Delta \mid \sharp(\{y \mid (x, y) \in U^{\mathcal{I}}\}) \geq n\}$,
- $(\leq n U)^{\mathcal{I}} = \{x \in \Delta \mid \sharp(\{y \mid (x, y) \in U^{\mathcal{I}}\}) \leq n\}$,
- $(R^-)^{\mathcal{I}} = \{(a, b) \mid (b, a) \in R^{\mathcal{I}}\}$.

The *satisfaction* of an axiom F in an interpretation $\mathcal{I} = (\Delta, \cdot^{\mathcal{I}})$, denoted $\mathcal{I} \models F$, is defined as follows:

(1) $\mathcal{I} \models C \sqsubseteq D$ iff $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$,

(2) $\mathcal{I} \models R \sqsubseteq S$ iff $R^{\mathcal{I}} \subseteq S^{\mathcal{I}}$,

(3) $\mathcal{I} \models \mathrm{Trans}(R)$ iff $R^{\mathcal{I}}$ is transitive,

(4) $\mathcal{I} \models C(a)$ iff $a^{\mathcal{I}} \in C^{\mathcal{I}}$,

(5) $\mathcal{I} \models R(a, b)$ iff $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}$,

(6) $\mathcal{I} \models U(a, v)$ iff $(a^{\mathcal{I}}, v) \in U^{\mathcal{I}}$,

(7) $\mathcal{I} \models a = b$ iff $a^{\mathcal{I}} = b^{\mathcal{I}}$, and

(8) $\mathcal{I} \models a \neq b$ iff $a^{\mathcal{I}} \neq b^{\mathcal{I}}$.

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

Typical important reasoning tasks related to Description Logic knowledge bases L are the following:

(1) decide whether a given L is satisfiable;

(2) given DL and a concept C, decide whether $DL \not\models C \sqsubseteq \bot$;

(3) given DL and two concepts C and D, decide whether $DL \models C \sqsubseteq D$;

(4) given DL, $a \in \mathbf{I}$, and a concept C, decide whether $DL \models C(a)$; and

(5) given DL, $a, b \in \mathbf{I}$ (resp., $a \in \mathbf{I}$ and $v \in \mathrm{dom}(\mathbf{D})$), and $R \in \mathbf{R}_A$ (resp., $U \in \mathbf{R}_D$), decide whether $DL \models R(a, b)$ (resp., $DL \models U(a, v)$).

Here, (1) is a special case of (2), since DL is satisfiable iff $DL \not\models \top \sqsubseteq \bot$. Moreover, (2) and (3) can be reduced to each other, since $DL \models C \sqcap \neg D \sqsubseteq \bot$ iff $DL \models C \sqsubseteq D$. Finally, in $\mathcal{SHOIN}(\mathbf{D})$, (4) and (5) are special cases of (3).

2.4.4 OWL

OWL [Bechhofer et al., 2004] is an ontology language for the Semantic Web, developed by the World Wide Web Consortium (W3C) Web Ontology Working Group. The language emerged from its predecessors SHOE [Heflin and Hendler, 2000], a frame-based language with XML-syntax, and DAML+OIL [Horrocks, 2002b,a], which is highly integrated with RDF and itself was a combination of OIL [Fensel et al., 2001] and DAML [Hendler and McGuinness, 2000]. OWL became a W3C Recommendation in February 2004 and is as such understood by the industry and the web community as a web standard.

The role of ontologies in the Semantic Web is to provide a means for representing knowledge in both human- and machine-readable format and thus fostering the automation of information access and retrieval. More specifically, they are supposed to provide structured vocabularies than can be used to describe the relationship between different terms. Being an effort of the W3C, OWL represents a layer in the so-called Semantic Web Cake, which visualizes the stack of semantic technologies as a series of strata (as we have shown in the introduction in Figure 1.1). OWL is build on top of XML and RDF resp. RDF Schema and thus further extends the ability of stating facts and class- resp. property hierarchies.

The language OWL provides the three increasingly expressive sublanguages *OWL Lite*, *OWL DL*, and *OWL Full*, where OWL DL basically corresponds to DAML+OIL. The languages OWL Lite and OWL DL are essentially very expressive Description Logics with an RDF/XML syntax and an abstract frame-like syntax [Horrocks et al., 2003]. One can therefore exploit a large body of existing previous work on Description Logic research, for example, to define the formal semantics of the languages, to understand their formal properties (in particular, the decidability and the complexity of key inference problems), and for an automated reasoning support. In fact, as shown by Horrocks and Patel-Schneider [2003], ontology entailment in OWL Lite and OWL DL reduces to knowledge base (un)satisfiability in the Description Logics $\mathcal{SHIF}(\mathbf{D})$ and $\mathcal{SHOIN}(\mathbf{D})$, respectively.

OWL DL allows for the specification of classes, datatypes, individuals and data values using the constructs shown in Table 2.1. There, the first column shows the OWL abstract syntax for the construction, while the second column gives the standard Description Logic syntax. OWL DL uses these description-forming constructs in axioms that provide information about classes, properties, and individuals, as shown in Table 2.2. Again, the frame-like abstract syntax is given in the first column, and the standard Description Logic syntax is given in the second column.

Syntax

Since an OWL ontology is in principle just an RDF graph, it can also be represented by RDF triples and hence be written in a variety of different syntactic forms. The most common, albeit not too readable, is RDF/XML.

Example 2.4.1 As an example, consider the following Description Logics axiom:

$$\geq 2\ supplier \sqsubseteq Discount$$

This expression states that all individuals x of pairs $\langle x, y\rangle \in supplier$ that have at least two different y are also members of the class *Discount*. For instance, If *supplier* contains the tuples $\langle shop_1, part_a\rangle, \langle shop_1, part_b\rangle, \langle shop_2, part_c\rangle$, then $shop_1$ will be in *Discount*.

Next, we formulate this axiom in RDF/XML syntax of OWL:[6]

```
<owl:Class rdf:ID="Discount">
  <owl:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#supplier"/>
      <owl:minCardinality rdf:datatype="xsd:nonNegativeInteger">2
        </owl:minCardinality>
    </owl:Restriction>
  </owl:subClassOf>
</owl:Class>
```

Expressed in the OWL abstract syntax the same fragment becomes much more concise:

```
Class(Discount partial restriction(supplier minCardinality(2)))
Class(Part partial)
Class(Shop partial)
```

◇

Semantics

The Semantics of OWL DL, as we already pointed out, corresponds to $\mathcal{SHOIN}(\mathbf{D})$ and is summarized in the third column of Table 2.1. The particular datatypes used in OWL are taken from RDF and XML Schema Datatypes. $\Delta^{\mathcal{I}}$ is the domain of individuals in a model and $\Delta^{\mathcal{I}}_{\mathbf{D}}$ is the domain of data values.

OWL Lite, being closely related to $\mathcal{SHIF}(\mathbf{D})$, prohibits unions and complements, restricts intersections to the implicit intersections in the frame-like class axioms, limits all embedded descriptions to concept names, does not allow individuals to show up in descriptions or class axioms, and limits cardinalities to 0 or 1. It therefore represents a subset of OWL DL, reducing its expressivity and hence its complexity. $\mathcal{SHOIN}(\mathbf{D})$ has a time complexity of NEXP for central reasoning problems, which is in $\mathcal{SHIF}(\mathbf{D})$ reduced to EXP in the worst case.

[6] We assume the proper definitions of the used namespaces in the preamble of the respective XML-file.

Abstract Syntax	DL Syntax	Semantics
Descriptions (C)		
A (URI reference)	A	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
`owl:Thing`	$\top$	$\texttt{owl:Thing}^{\mathcal{I}} = \Delta^{\mathcal{I}}$
`owl:Nothing`	$\bot$	$\texttt{owl:Nothing}^{\mathcal{I}} = \{\}$
`intersectionOf`(C_1 C_2 ...)	$C_1 \sqcap C_2$	$(C_1 \sqcap C_2)^{\mathcal{I}} = C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}}$
`unionOf`(C_1 C_2 ...)	$C_1 \sqcup C_2$	$(C_1 \sqcup C_2)^{\mathcal{I}} = C_1^{\mathcal{I}} \cup C_2^{\mathcal{I}}$
`complementOf`(C)	$\neg C$	$(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
`oneOf`(o_1 ...)	$\{o_1, \ldots\}$	$\{o_1, \ldots\}^{\mathcal{I}} = \{o_1^{\mathcal{I}}, \ldots\}$
`restriction`(R `someValuesFrom`(C))	$\exists R.C$	$(\exists R.C)^{\mathcal{I}} = \{x \mid \exists y.\langle x,y\rangle \in R^{\mathcal{I}} \text{ and } y \in C^{\mathcal{I}}\}$
`restriction`(R `allValuesFrom`(C))	$\forall R.C$	$(\forall R.C)^{\mathcal{I}} = \{x \mid \forall y.\langle x,y\rangle \in R^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}}\}$
`restriction`(R `hasValue`(o))	$R : o$	$(\forall R.o)^{\mathcal{I}} = \{x \mid \langle x, o^{\mathcal{I}}\rangle \in R^{\mathcal{I}}\}$
`restriction`(R `minCardinality`(n))	$\geq nR$	$(\geq nR)^{\mathcal{I}} = \{x \mid \sharp(\{y.\langle x,y\rangle \in R^{\mathcal{I}}\}) \geq n\}$
`restriction`(R `maxCardinality`(n))	$\leq nR$	$(\leq nR)^{\mathcal{I}} = \{x \mid \sharp(\{y.\langle x,y\rangle \in R^{\mathcal{I}}\}) \leq n\}$
`restriction`(U `someValuesFrom`(D))	$\exists U.D$	$(\exists U.D)^{\mathcal{I}} = \{x \mid \exists y.\langle x,y\rangle \in U^{\mathcal{I}} \text{ and } y \in D^{\mathbf{D}}\}$
`restriction`(U `allValuesFrom`(D))	$\forall U.D$	$(\forall U.D)^{\mathcal{I}} = \{x \mid \forall y.\langle x,y\rangle \in U^{\mathcal{I}} \rightarrow y \in D^{\mathbf{D}}\}$
`restriction`(U `hasValue`(v))	$U : v$	$(U : v)^{\mathcal{I}} = \{x \mid \langle x, v^{\mathcal{I}}\rangle \in U^{\mathcal{I}}\}$
`restriction`(U `minCardinality`(n))	$\geq nU$	$(\geq nU)^{\mathcal{I}} = \{x \mid \sharp(\{y.\langle x,y\rangle \in U^{\mathcal{I}}\}) \geq n\}$
`restriction`(U `maxCardinality`(n))	$\leq nU$	$(\leq nU)^{\mathcal{I}} = \{x \mid \sharp(\{y.\langle x,y\rangle \in U^{\mathcal{I}}\}) \leq n\}$
Data Ranges (D)		
D (URI reference)	D	$D^{\mathbf{D}} \subseteq \Delta_{\mathbf{D}}^{\mathcal{I}}$
`oneOf`(v_1 ...)	$\{v_1, \ldots\}$	$\{v_1, \ldots\}^{\mathcal{I}} = \{v_1^{\mathcal{I}}, \ldots\}$
Object Properties (R)		
R (URI reference)	R	$R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
	R^-	$(R^-)^{\mathcal{I}} = (R^{\mathcal{I}})^-$
Datatype Properties (U)		
U (URI reference)	U	$U^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta_{\mathbf{D}}^{\mathcal{I}}$
Individuals (o)		
o (URI reference)	o	$o^{\mathcal{I}} \in \Delta^{\mathcal{I}}$
Data Values (v)		
v (RDF literal)	v	$v^{\mathcal{I}} = v^{\mathbf{D}}$

Table 2.1: OWL DL Descriptions, Data Ranges, Properties, Individuals, and Data Values.

Abstract Syntax	DL Syntax	Semantics
`Class(`A `partial` C_1 ... C_n`)`	$A \sqsubseteq C_1 \sqcap \ldots \sqcap C_n$	$A^{\mathcal{I}} \subseteq C_1^{\mathcal{I}} \cap \ldots \cap C_n^{\mathcal{I}}$
`Class(`A `complete` C_1 ... C_n`)`	$A = C_1 \sqcap \ldots \sqcap C_n$	$A^{\mathcal{I}} = C_1^{\mathcal{I}} \cap \ldots \cap C_n^{\mathcal{I}}$
`EnumeratedClass(`A o_1 ... o_n`)`	$A = \{o_1, \ldots, o_n\}$	$A^{\mathcal{I}} = \{o_1^{\mathcal{I}}, \ldots, o_n^{\mathcal{I}}\}$
`SubClassOf(`C_1 C_2`)`	$C_1 \sqsubseteq C_2$	$C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$
`EquivalentClasses(`C_1 ... C_n`)`	$C_1 = \ldots = C_n$	$C_1^{\mathcal{I}} = \ldots = C_n^{\mathcal{I}}$
`DisjointClasses(`C_1 ... C_n`)`	$C_i \sqcap C_j = \bot, i \neq j$	$C_i^{\mathcal{I}} \cap C_j^{\mathcal{I}}\{\}, i \neq j$
`Datatype(`D`)`		$D^{\mathcal{I}} \subseteq \Delta_{\mathbf{D}}^{\mathcal{I}}$
`DatatypeProperty(`U `super(`U_1`)`...`super(`U_n`)`	$U \sqsubseteq U_i$	$U^{\mathcal{I}} \subseteq U_i^{\mathcal{I}}$
`domain(`C_1`)`...`domain(`C_m`)`	$\geq 1\, U \sqsubseteq C_i$	$U^{\mathcal{I}} \subseteq C_i^{\mathcal{I}} \times \Delta_{\mathbf{D}}^{\mathcal{I}}$
`range(`D_1`)`...`range(`D_l`)`	$\top \sqsubseteq \forall U.D_i$	$U^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times D_i^{\mathcal{I}}$
`[Functional])`	$\top \sqsubseteq \leq 1\, U$	$U^{\mathcal{I}}$ is functional
`SubPropertyOf(`U_1 U_2`)`	$U_1 \sqsubseteq U_2$	$U_1^{\mathcal{I}} \subseteq U_2^{\mathcal{I}}$
`EquivalentProperties(`U_1 ... U_n`)`	$U_1 = \ldots = U_n$	$U_1^{\mathcal{I}} = \ldots = U_n^{\mathcal{I}}$
`ObjectProperty(`R `super(`R_1`)`...`super(`R_n`)`	$R \sqsubseteq R_i$	$R^{\mathcal{I}} \subseteq R_i^{\mathcal{I}}$
`domain(`C_1`)`...`domain(`C_m`)`	$\geq 1\, R \sqsubseteq C_i$	$R^{\mathcal{I}} \subseteq C_i^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
`range(`C_1`)`...`range(`C_l`)`	$\top \sqsubseteq \forall R.C_i$	$R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times C_i^{\mathcal{I}}$
`[inverseOf(`R_0`)]`	$R = ({}^{-}R_0)$	$R^{\mathcal{I}} = (R_0^{\mathcal{I}})^{-}$
`[Symmetric)]`	$R = ({}^{-}R)$	$R^{\mathcal{I}} = (R^{\mathcal{I}})^{-}$
`[Functional]`	$\top \sqsubseteq \leq 1\, R$	$R^{\mathcal{I}}$ is functional
`[InverseFunctional]`	$\top \sqsubseteq \leq 1\, R^{-}$	$(R^{\mathcal{I}})^{-}$ is functional
`[Transitive)]`	$Tr(R)$	$R^{\mathcal{I}} = (R^{\mathcal{I}})^{+}$
`SubPropertyOf(`R_1 R_2`)`	$R_1 \sqsubseteq R_2$	$R_1^{\mathcal{I}} \subseteq R_2^{\mathcal{I}}$
`EquivalentProperties(`R_1 ... R_n`)`	$R_1 = \ldots = R_n$	$R_1^{\mathcal{I}} = \ldots = R_n^{\mathcal{I}}$
`AnnotationProperty(`S`)`		
`Individual(`o `type(`C_1`)`...`type(`C_n`)`	$O \in C_i$	$o^{\mathcal{I}} \in C_i^{\mathcal{I}}$
`value(`R_1 o_1`)`...`value(`R_n o_n`)`	$\langle o, o_i \rangle \in R_i$	$\langle o^{\mathcal{I}}, o_i^{\mathcal{I}} \rangle \in R_i^{\mathcal{I}}$
`value(`U_1 v_1`)`...`value(`U_n v_n`))`	$\langle o, v_i \rangle \in R_i$	$\langle o^{\mathcal{I}}, v_i^{\mathcal{I}} \rangle \in R_i^{\mathcal{I}}$
`SameIndividual(`o_1 ... o_n`)`	$o_1 = \ldots = o_n$	$o_1^{\mathcal{I}} = \ldots = o_n^{\mathcal{I}}$
`DifferentIndividuals(`o_1 ... o_n`)`	$o_i \neq o_j, i \neq j$	$o_i^{\mathcal{I}} \neq o_j^{\mathcal{I}}, i \neq j$

Table 2.2: OWL DL Axioms and Facts.

Chapter 3

dl-Programs

3.1 Introduction

In this chapter we present our approach of combining terminological reasoning in form of ontologies that are based on certain Description Logics with logic programs under the answer-set semantics. The original motivation for this work was the current effort in the Semantic Web and logic programming communities of creating a suitable formalism to build rules on top of ontologies, hence further shaping the realization of the Semantic Web. Several obstacles arise for finding the right combination of rich ontology languages such as OWL which are based on classical logic with logic programming based languages such as answer-set programming, since these formalisms are semantically separated by a considerable gap. In the following introduction, we will point out the relation of different flavors of logic programming based rule languages to classical first-order logic and show how their disparity hamper an easy solution to the combination of rules and ontologies.

3.1.1 Logic Programming vs. Classical Logic

As it is well-known, the core of logic programming, i.e., definite positive programs, has a direct correspondence with the Horn subset of classical first-order logic, i.e., a definite ($n = 1$), *not*-free ($k = m$) rule of the form (2.1) can be read as a first-order sentence of the form

$$(\forall) a_1 \leftarrow b_1 \wedge \ldots \wedge b_k \tag{3.1}$$

where ($\forall$) denotes the universal closure of all free variables in the formula. This subset of first-order logic allows for a sound and complete decision procedure for entailment of ground atomic formulae, which — in case of a function-free theory (which corresponds to Datalog) — is computable in finite polynomial time.

However, it is important to point out some slight but decisive differences between the logic programming (LP) view and the first-order view of definite programs.

Non-ground entailment. The first divergence becomes apparent already in case of positive programs. LP semantics is defined in terms of minimal Herbrand models, i.e., sets of ground facts. Take for example the logic program

$$\begin{aligned} \mathit{PotableLiquid}(X) &\leftarrow \mathit{Wine}(X) \\ \mathit{Wine}(X) &\leftarrow \mathit{Whitewine}(X) \\ \mathit{Whitewine}(\text{“}\mathit{Welschriesling}\text{”}). & \end{aligned}$$

Both the logic program reading and the Horn reading of this program following (3.1) above yield the ground entailment of facts

$$\{Whitewine(\text{“}Welschriesling\text{”}), Wine(\text{“}Welschriesling\text{”}), \\ PotableLiquid(\text{“}Welschriesling\text{”})\}.$$

The first-order reading of the program would allow further non-factual inferences such as $PotableLiquid(\text{“}Welschriesling\text{”}) \leftarrow Wine(\text{“}Welschriesling\text{”})$ or $\forall x.PotableLiquid(x) \leftarrow Whitewine(x)$, which are not entailed by the logic program. Logic programs and minimal Herbrand models (and answer sets as their extension) are only concerned with facts.

Negation as failure vs. classical negation. Divergences become more severe when considering programs with negation. Negation as failure *not* is evaluated with respect to a closed world assumption (CWA) whereas negation in Description Logics and thus in OWL (`owl:complementOf`) is interpreted classically. Let us again demonstrate this with a small example:

$$\begin{aligned} Wine(X) &\leftarrow Whitewine(X) \\ NonWhite(X) &\leftarrow not\ Whitewine(X) \\ Wine(myDrink)&. \end{aligned}$$

Not given any additional information, under the answer-set semantics this program entails both bravely and cautiously the fact $NonWhite(myDrink)$. However, this conclusion would not be justified in a first-order reading of the above program, like

$$\begin{aligned} &(\forall x.Whitewine(x) \supset Wine(x)) \wedge \\ &(\forall x.\neg NonWhite(x) \supset not\ Whitewine(x)) \wedge \\ &Wine(myDrink). \end{aligned}$$

or Description Logics reading, such as:

$$\begin{aligned} Whitewine &\sqsubseteq Wine \\ \neg Whitewine &\sqsubseteq NonWhite \\ myDrink &\in Wine. \end{aligned}$$

The reason for this is the different purposes which classical negation and negation as failure are serving, the latter rather to be understood as modeling (defeasible) default assumptions with non-monotonic behavior. While some people argue that such kind of non-monotonic negation is unsuitable for an open environment such as the Web, there are several applications from e.g., information integration where negation as failure has been shown to be particularly useful.

Strong negation vs. classical negation. Note that also strong negation, as used in answer-set programming, has a slightly different flavor than its classical counterpart. That is, the following two representations of a logic program and a description logic theory again slightly diverge:

$$\begin{aligned} Wine(X) &\leftarrow Whitewine(X). \\ \neg Wine(myDrink)&. \end{aligned} \qquad\qquad \begin{aligned} Whitewine &\sqsubseteq Wine \\ myDrink &\in \neg Wine. \end{aligned}$$

Whereas the DL knowledge base would entail that $myDrink \in \neg Whitewine$ the corresponding fact $\neg Whitewine(myDrink)$ is not a justified conclusion in an LP setting i.e., the law of the excluded middle or contraposition do not hold upfront in ASP. Nonetheless, one can "emulate" classical behavior of certain predicates in ASP by, for instance, adding a rule $\neg Whitewine(X) \vee Whitewhine(X)$ in the above example.

Logic Programming and equality. Answer-set programming engines typically deploy a unique name assumption (UNA) and do not support real equality reasoning, i.e., equality in the head of rules. This does not necessarily comply with the view in classical logic, and thus RDF Schema and OWL where no such assumption is made. While equality "$=$" and inequality "$\neq$" predicates are allowed in rule bodies, these represent syntactic equality and (default) negation thereof only. This must not be confused with the OWL directives `sameAs` and `differentFrom` directives. Let us consider the following rule base:

$$knows(\text{“}Bob\text{”}, \text{“}Alice\text{”}).$$
$$knowsOtherPeople(X) \leftarrow knows(X, Y), X \neq Y.$$

Under standard ASP semantics where UNA is deployed, "$\neq$" amounts to "$not =$", and thus $knowsOtherPeople(\text{“}Bob\text{”})$ would be entailed.

Decidability. Finally, the probably largest obstacle for combining the Description Logics world of OWL with the logic programming world of answer-set programming stems from the fact that both face undecidability issues from two completely different angles.

As mentioned above, decidability of answer-set programming follows from the fact that it is based on function-free Horn logic where ground entailment can be determined by checking the finite subsets of the Herbrand base, i.e., decidability and termination of evaluation strategies is guaranteed by finiteness of the domain.

Decidability of Description Logics reasoning tasks such as satisfiability, class subsumption, or class membership relies on the so-called *tree model property*. This property basically says that a description logic knowledge base has a model iff it has a tree-shaped model, whose depth and branching factor are bounded by the size of the knowledge base [Baader et al., 2003].

It becomes clear that the property of decidability stems from different conditions, such that LP and Description Logics cannot be combined immediately. As shown in [Levy and Rousset, 1998] the naive combination of even a very simple DL with arbitrary Horn Logic is undecidable which they proof by a reduction of Turing machines.

3.1.2 Strategies for Combining Rules and Ontologies

As one can expect by the above-mentioned problems, combining logic programming and Description Logics is not straightforward. If decidability is not an issue, a naive combination of Description Logics and Horn rules could be a possible approach for the Semantic Web rules layer. The Semantic Web Rule Language (SWRL) [Horrocks et al., 2004] proposal, a recent W3C member submission, straightforwardly extends OWL in this spirit. Nonetheless, in the following we want to take a closer look at approaches which still retain decidability in a more cautious integration.

On the other extreme, the overcautious approach of allowing interoperability only on the intersection of Description Logics and Horn-Logics seems to be too restricted. Grosof et al. [2003] have defined this intersection where the LP and DL world coincide which they call DLP. However, such an approach leaves a rule and ontology language with very

restrictive expressivity. Layering several extensions in the direction of logic programming and ASP on top of the DLP fragment have lead to the Web Rule Language (WRL) proposal by Angele et al. [2005], an alternative W3C member submission.

When we want to combine full Description Logics with full answer-set programming, however, things become more involved. In principle the different proposals in the literature can be divided into two major approaches of rules and ontologies interaction:

Strict semantic integration. Rules are introduced by adapting existing semantics for rule languages directly in the Ontology layer. The restrictive DLP fragment on the one end and the undecidable SWRL approach on the other mark two extremes of this approach. Nonetheless, recently several proposals have been made to extend expressiveness while still retaining decidability, remarkably several attempts in the ASP field. Common to these approaches is the restriction of the combined language in a way that guarantees "safe interaction" of the LP and DL parts of the language, usually by introducing a safety condition in rules. Approaches described in [Levy and Rousset, 1998, Motik et al., 2005, Rosati, 2005, 2006a] fall under this category.

Strict semantic separation. In this setting ASP should play a central role in the rule layer, while OWL/RDF Schema flavors would keep their purpose of description languages, not aimed at intensive reasoning tasks, in the underlying ontology layer. The two layers are kept strictly distinguished and only communicate via a "safe interface": From the rule layer point of view, ontologies are dealt with as an external source of information whose semantics is treated independently. Non-monotonic reasoning and rules are allowed in a decidable setting, as well as arbitrary mixing of closed and open world reasoning. This approach is typical of [Eiter et al., 2004a, 2005d, Heymans et al., 2005b] and [Lukasiewicz, 2005b].

A more detailed account on contributions to each of these categories will be given in Subsection 3.10. For excellent surveys which classify the above-mentioned and other approaches we refer the interested reader to [Antoniou et al., 2005, Pan et al., 2004].

The remainder of this chapter is dedicated to the theory and application of one approach of the latter type, which was first presented in [Eiter et al., 2004a]. In particular, we will address the following issues:

- We introduce *description logic programs* (*dl-programs*), which consist of a knowledge base L in a description logic and a finite set of description logic rules (*dl-rules*) P. Such rules are similar to usual rules in logic programs with negation as failure, but may also contain *queries to* L, possibly default-negated, in their bodies. As an important feature, such queries also allow for specifying an input from P, and thus for a *flow of information from* P *to* L, besides the flow of information from L to P, given by any query to L. For example, concepts and roles in L may be enhanced by facts generated from dl-rules, possibly involving heuristic knowledge and other concepts and roles from L.

- The queries to L are treated, fostering an encapsulation view, in a way such that logic programming and description logic inference are technically separated; mainly interfacing details need to be known. Compared to other similar work, this increases flexibility and is also amenable to privacy aspects for L and P. Furthermore, the nondeterminism inherent in answer sets is retained, supporting brave reasoning and the answer-set programming paradigm in which solutions of problems are represented by answer sets of a logic program.

- We define Herbrand models for dl-programs, and show that satisfiable positive dl-programs, in which default-negation does not occur and all queries to L are monotonic, have a unique least model. Furthermore, we show that more general stratified dl-programs can be associated, if consistent, with a unique minimal Herbrand model that is characterized through iterative least Herbrand models.

- We define *strong answer sets* for all dl-programs, based on a reduction to the least model semantics of positive dl-programs. For positive and stratified dl-programs, the strong answer-set semantics coincides with the (unique) minimal Herbrand model semantics associated. We also consider *weak answer sets* based on a reduction to the answer sets of ordinary logic programs. Strong answer sets are also weak answer sets, and both properly generalize answer sets of ordinary normal logic programs.

- We present fixpoint characterizations for the least model of a positive dl-program and the canonical minimal model of a stratified dl-program, and show how to compute these models by finite fixpoint iterations.

- We give a precise picture of the complexity of deciding strong and weak answer set existence for a dl-program KB. From this, the complexity of brave and cautious reasoning is easily derived. We consider the general case as well as the restrictions where KB is (a) positive, (b) stratified and contains only monotonic queries, and (c) stratified. We consider $\mathcal{SHIF}(\mathbf{D})$ and $\mathcal{SHOIN}(\mathbf{D})$, but most of our results can be easily transferred to other Description Logics having the same complexity (EXP and NEXP, respectively).

- Finally, we describe a prototype implementation of a reasoner for dl-programs under the different aforementioned semantics. We present the algorithms that are used for the evaluation as well as general techniques to increase its efficiency.

The rest of this chapter is organized as follows. In Section 3.2 we introduce the syntax of dl-programs. We then define Herbrand models of dl-programs, unique minimal Herbrand models of positive and stratified dl-programs, and the least model semantics for these kinds of dl-programs in Section 3.3. The strong and weak answer-set semantics for general dl-programs are finally presented in Section 3.4. Additionally, we introduce the well-founded semantics for dl-programs in Section 3.5. Section 3.6 shows how the unique minimal Herbrand models of positive and stratified dl-programs can be computed through fixpoint iterations. Section 3.7 provides a precise picture of the complexity of deciding strong and weak answer set existence for a dl-program. In Section 3.9 we present a prototype implementation of a dl-program reasoner, called NLP-DL. Section 3.10 eventually gives an overview on related work.

3.2 dl-Program Syntax

Informally, a dl-program consists of a description logic knowledge base L and a generalized normal program P, which may contain queries to L. Roughly, such a query asks whether a specific description logic axiom is entailed by L or not.

We first define dl-queries and dl-atoms, which are used to express queries to the description logic knowledge base L. A *dl-query* $Q(\mathbf{t})$ is either[1]

- a concept inclusion axiom F or its negation $\neg F$, or

[1] In Subsection 5.3.1 we will extend these queries by the possibility to state also conjunctive queries.

- 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* has the form

$$DL[S_1 op_1 p_1, \ldots, S_m op_m\ p_m; Q](\mathbf{t}), \qquad m \geq 0, \tag{3.2}$$

where each S_i is either a concept or a role, $op_i \in \{\uplus, \cup, \cap\}$, p_i is a unary resp. binary predicate symbol, and $Q(\mathbf{t})$ is a dl-query. We call $p_1, \ldots, p_m$ its *input predicate symbols*. Intuitively, $op_i = \uplus$ (resp., $op_i = \cup$) increases S_i (resp., $\neg S_i$) by the extension of p_i, while $op_i = \cap$ constrains S_i to p_i.

A *dl-rule* r has the form

$$a \leftarrow b_1, \ldots, b_n, not\, b_{n+1}, \ldots, not\, b_m, \tag{3.3}$$

where any literal $b_1, \ldots, b_m \in B(r)$ may be a dl-atom. We define $H(r) = a$ and $B(r) = B^+(r) \cup B^-(r)$, where $B^+(r) = \{b_1, \ldots, b_n\}$ and $B^-(r) = \{b_{n+1}, \ldots, b_m\}$. If $B(r) = \emptyset$ and $H(r) \neq \emptyset$, then r is a *fact*. A *dl-program* $KB = (L, P)$ consists of a description logic knowledge base L and a finite set of dl-rules P.

We use the following example to illustrate our main ideas.

Example 3.2.1 Suppose we want to assign reviewers to papers, based on certain information about the papers and available persons, using a description logic knowledge base L_R containing knowledge about scientific publications.

We assume not to be aware of the entire structure and contents of L_R, but of the following aspects. L_R classifies papers into research areas, stored in a concept *Area*, depending on keyword information. The roles *keyword* and *inArea* associate with each paper its relevant keywords and the areas it is classified into (obtained, e.g., by reification of the classes). Furthermore, a role *expert* relates persons to their areas of expertise, and a concept *Referee* contains all referees. Eventually, a role *topicOf* associates with a cluster of similar keywords all its members. Consider then the following dl-program P_R:

(1) $paper(p_1)$; $kw(p_1, Semantic\_Web)$;
(2) $paper(p_2)$; $kw(p_2, Bioinformatics)$; $kw(p_2, Answer\_Set\_Programming)$;
(3) $kw(P, K_2) \leftarrow kw(P, K_1)$, $DL[topicOf](S, K_1)$, $DL[topicOf](S, K_2)$;
(4) $paperArea(P, A) \leftarrow DL[keywords \uplus kw; inArea](P, A)$;
(5) $cand(X, P) \leftarrow paperArea(P, A)$, $DL[Referee](X)$, $DL[expert](X, A)$;
(6) $assign(X, P) \leftarrow cand(X, P), not\, \neg assign(X, P)$;
(7) $\neg assign(Y, P) \leftarrow cand(Y, P), assign(X, P)$, $X \neq Y$;
(8) $a(P) \leftarrow assign(X, P)$;
(9) $error(P) \leftarrow paper(P), not\, a(P)$.

Intuitively, lines (1) and (2) specify the keyword information of two papers, p_1 and p_2, which should be assigned to reviewers. The rule (3) augments, by choice of the designer, the keyword information with similar ones (hoping for good). The rule (4) queries the augmented L_S to retrieve the areas each paper is classified into, and the rule (5) singles out review candidates based on this information from experts among the reviewers according to L_R. Rules (6) and (7) pick one of the candidate reviewers for a paper (multiple reviewers

can be selected similarly). Finally, (8) and (9) check if each paper is assigned; if not, an error is flagged. Note that, in view of rules (3)–(5), information flows in both directions between the description logic knowledge base L_R and the knowledge represented by the above dl-program.

To illustrate the use of $\uplus$, a predicate *poss_Referees* may be defined in P_R, and "*Referee* $\uplus$ *poss_Referees*" may be added in the first dl-atom of (5), which thus constrains the set of referees.

The dl-rule below shows in particular how dl-rules can be used to encode certain qualified number restrictions, which are not available in $\mathcal{SHOIN}(\mathbf{D})$ It defines an *expert* as an author of at least three papers of the same area:

$$\begin{aligned} \mathit{expert}(X, A) \leftarrow\ & DL[\mathit{isAuthorOf}](X, P_1), \\ & DL[\mathit{isAuthorOf}](X, P_2), \\ & DL[\mathit{isAuthorOf}](X, P_3), \\ & DL[\mathit{inArea}](P_1, A), \\ & DL[\mathit{inArea}](P_2, A), \\ & DL[\mathit{inArea}](P_3, A), \\ & P_1 \neq P_2,\ P_2 \neq P_3,\ P_3 \neq P_1. \end{aligned}$$

◇

Example 3.2.2 A small computer store obtains its hardware from several vendors. It uses the following description logic knowledge base L_S, which contains information about the product range that is provided by each vendor and about possible rebate conditions (we assume here that choosing two or more parts from the same seller results in a discount). For some parts, a shop may be already contracted as supplier.

$$\begin{aligned} & \geq 1\ \mathit{supplier} \sqsubseteq \mathit{Shop};\ \ \top \sqsubseteq \forall \mathit{supplier}.\mathit{Part};\ \ \geq 2\ \mathit{supplier} \sqsubseteq \mathit{Discount}; \\ & \mathit{Part}(\mathit{harddisk});\ \ \mathit{Part}(\mathit{cpu});\ \ \mathit{Part}(\mathit{case}); \\ & \mathit{Shop}(s_1);\ \ \mathit{Shop}(s_2);\ \ \mathit{Shop}(s_3); \\ & \mathit{provides}(s_1, \mathit{case});\ \ \mathit{provides}(s_2, \mathit{cpu});\ \ \mathit{provides}(s_3, \mathit{case}); \\ & \mathit{provides}(s_1, \mathit{cpu});\ \ \mathit{provides}(s_2, \mathit{harddisk});\ \ \mathit{provides}(s_3, \mathit{harddisk}); \\ & \mathit{supplier}(s_3, \mathit{case}). \end{aligned}$$

Here, the first two axioms determine *Shop* and *Part* as domain and range of the property *supplier*, respectively, while the third axiom constitutes the concept *Discount* by putting a cardinality constraint on *supplier*.

Consider now the dl-program $KB_S = (L_S, P_S)$, with L_S as above and P_S given as follows, choosing vendors for needed parts w.r.t. possible rebates:

$$\begin{aligned} &(1) \quad \mathit{vendor}(s_2);\quad \mathit{vendor}(s_1);\quad \mathit{vendor}(s_3); \\ &(2) \quad \mathit{needed}(\mathit{cpu});\quad \mathit{needed}(\mathit{harddisk});\quad \mathit{needed}(\mathit{case}); \\ &(3) \quad \mathit{avoid}(V) \leftarrow \mathit{vendor}(V), \mathit{not}\ \mathit{rebate}(V); \\ &(4) \quad \mathit{rebate}(V) \leftarrow \mathit{vendor}(V), DL[\mathit{supplier} \uplus \mathit{buy\_cand}; \mathit{Discount}](V); \\ &(5) \quad \mathit{buy\_cand}(V, P) \leftarrow \mathit{vendor}(V), \mathit{not}\ \mathit{avoid}(V), DL[\mathit{provides}](V, P), \mathit{needed}(P), \\ &\qquad\qquad\qquad\qquad \mathit{not}\ \mathit{exclude}(P) \\ &(6) \quad \mathit{exclude}(P) \leftarrow \mathit{buy\_cand}(V_1, P), \mathit{buy\_cand}(V_2, P), V_1 \neq V_2; \end{aligned}$$

(7) $exclude(P) \leftarrow DL[supplier](V, P), needed(P);$

(8) $supplied(V, P) \leftarrow DL[supplier \uplus buy\_cand; supplier](V, P), needed(P).$

Rules (3)–(5) choose a possible vendor (*buy_cand*) for each needed part, taking into account that the selection might affect the rebate condition (by feeding the possible vendor back to L_S, where the discount is determined). Rules (6) and (7) assure that each hardware part is bought only once, considering that for some parts a supplier might already be chosen. Rule (8) eventually summarizes all purchasing results. ◇

Example 3.2.3 An existing network must be extended by new nodes. The knowledge base L_N contains information about existing nodes and their interconnections as well as a definition of "overloaded" nodes (concept *HighTrafficNode*), which depends on the number of connections of the respective node (here, all nodes with more than three connections belong to *HighTrafficNode*):

$\geq 1\ wired \sqsubseteq Node;\ \ \top \sqsubseteq \forall wired.Node;\ \ wired = (^{-}wired);$
$\geq 4\ wired \sqsubseteq HighTrafficNode;$
$Node(n_1);\ \ Node(n_2);\ \ Node(n_3);\ \ Node(n_4);\ \ Node(n_5);$
$wired(n_1, n_2);\ \ wired(n_2, n_3);\ \ wired(n_2, n_4);$
$wired(n_2, n_5);\ \ wired(n_3, n_4);\ \ wired(n_3, n_5).$

To evaluate possible combinations of connecting the new nodes, the following program P_N is specified:

(1) $newnode(add_1);$

(2) $newnode(add_2);$

(3) $overloaded(X) \leftarrow DL[wired \uplus connect; HighTrafficNode](X);$

(4) $connect(X, Y) \leftarrow newnode(X), DL[Node](X), not\ overloaded(Y), not\ excl(X, Y);$

(5) $excl(X, Y) \leftarrow connect(X, Z), DL[Node](Y), Y \neq Z;$

(6) $excl(X, Y) \leftarrow connect(Z, Y), newnode(Z), newnode(X), Z \neq X;$

(7) $excl(add_1, n_4).$

Rules (1)–(2) define the new nodes to be added. Rule (3) imports knowledge about overloaded nodes in the existing network, taking new connections already into account. Rule (4) connects a new node to an existing one, provided the latter is not overloaded and the connection is not to be disallowed, which is specified by Rule (5) (there must not be more than one connection for each new node) and Rule (6) (two new nodes cannot be connected to the same existing one). Rule (7) states a specific condition: Node add_1 must not be connected with n_4. ◇

3.3 Least Model Semantics for dl-Programs

We first define Herbrand interpretations and the truth of dl-programs in Herbrand interpretations. In the sequel, let $KB = (L, P)$ be a dl-program.

The *Herbrand base* of P, denoted HB_P, is the set of all ground literals with a standard predicate symbol that occurs in P and constant symbols in Φ. An *interpretation* I relative to P is a consistent subset of HB_P. We say I is a *model* of $l \in HB_P$ under L, or I

satisfies l under L, denoted $I \models_L l$, iff $l \in I$. It is a *model* of a ground dl-atom $a = DL[S_1 op_1\ p_1, \ldots, S_m op_m p_m; Q](\mathbf{c})$ under L, or I *satisfies* a under L, denoted $I \models_L a$, iff $L \cup \bigcup_{i=1}^{m} A_i(I) \models Q(\mathbf{c})$, where

- $A_i(I) = \{S_i(\mathbf{e}) \mid p_i(\mathbf{e}) \in I\}$, for $op_i = \uplus$;
- $A_i(I) = \{\neg S_i(\mathbf{e}) \mid p_i(\mathbf{e}) \in I\}$, for $op_i = \cup$;
- $A_i(I) = \{\neg S_i(\mathbf{e}) \mid p_i(\mathbf{e}) \in I \text{ does not hold}\}$, for $op_i = \cap$.

We say I is a *model* of a ground dl-rule r iff $I \models_L l$ for all $l \in B^+(r)$ and $I \not\models_L l$ for all $l \in B^-(r)$ implies $I \models_L H(r)$. It is a *model* of a dl-program $KB = (L, P)$, or I *satisfies* KB, denoted $I \models KB$, iff $I \models_L r$ for all $r \in \mathit{ground}(P)$. We say KB is *satisfiable* (resp., *unsatisfiable*) iff it has some (resp., no) model.

3.3.1 Least Model Semantics of Positive dl-Programs

We now define positive dl-programs, which are "*not*"-free dl-programs that involve only monotonic dl-atoms. Like ordinary positive programs, every positive dl-program that is satisfiable has a unique least model, which naturally characterizes its semantics.

A ground dl-atom a is *monotonic* relative to $KB = (L, P)$ iff $I \subseteq I' \subseteq HB_P$ implies that if $I \models_L a$ then $I' \models_L a$. A dl-program $KB = (L, P)$ is *positive* iff (i) P is "*not*"-free, and (ii) every ground dl-atom that occurs in $\mathit{ground}(P)$ is monotonic relative to KB.

Observe that a dl-atom containing $\cap$ may fail to be monotonic, since an increasing set of $p_i(\mathbf{e})$ in P results in a reduction of $\neg S_i(\mathbf{e})$ in L, whereas dl-atoms containing $\uplus$ and $\cup$ only are always monotonic.

For ordinary positive programs P, the intersection of two models of P is also a model of P. The following lemma shows that a similar result holds for positive dl-programs KB.

Lemma 3.3.1 *Let $KB = (L, P)$ be a positive dl-program. If the interpretations $I_1, I_2 \subseteq HB_P$ are models of KB, then $I_1 \cap I_2$ is also a model of KB.*

Proof. Suppose that $I_1, I_2 \subseteq HB_P$ are models of KB, that is, $I_i \models_L r$ for every $r \in \mathit{ground}(P)$ and $i \in \{1, 2\}$. We now show that $I = I_1 \cap I_2$ is also a model of KB, that is, $I \models_L r$ for every $r \in \mathit{ground}(P)$. Consider any $r \in \mathit{ground}(P)$, and assume that $I \models_L l$ for all $l \in B^+(r) = B(r)$. That is, $I \models_L l$ for all classical literals $l \in B(r)$ and $I \models_L a$ for all dl-atoms $a \in B(r)$. Hence, $I_i \models_L l$ for all classical literals $l \in B(r)$, for every $i \in \{1, 2\}$. Furthermore, since every dl-atom in $\mathit{ground}(P)$ is monotonic relative to KB, it holds that $I_i \models_L a$ for all dl-atoms $a \in B(r)$, for every $i \in \{1, 2\}$. Since I_1 and I_2 are models of KB, it follows that $I_i \models_L H(r)$, for every $i \in \{1, 2\}$, and thus $I \models_L H(r)$. This shows that $I \models_L r$. Hence, I is a model of KB. □

As an immediate corollary of this result, every satisfiable positive dl-program KB has a unique least model, denoted M_{KB}, which is contained in every model of KB.

Corollary 3.3.2 *Let $KB = (L, P)$ be a positive dl-program. If KB is satisfiable, then there exists a unique model $I \subseteq HB_P$ of KB such that $I \subseteq J$ for all models $J \subseteq HB_P$ of KB.*

Example 3.3.1 Consider the dl-program KB comprising of the rules (1)–(6) from Example 3.2.1. Clearly, KB is "*not*"-free. Moreover, as the dl-atoms do not contain $\cap$, they are all monotonic. Thus, the dl-program is positive. As well, its unique least model contains all review candidates for the given papers p_1 and p_2. ◇

3.3.2 Iterative Least Model Semantics of Stratified dl-Programs

We next define stratified dl-programs, which are intuitively composed of hierarchic layers of positive dl-programs linked via default-negation and certain dl-atoms. Like for ordinary stratified programs, a canonical minimal model can be singled out by a number of iterative least models, which naturally describes the semantics, provided some model exists. We can accommodate this with possibly non-monotonic dl-atoms by treating them similarly as NAF-literals. This is particularly useful, if we do not know a priori whether some dl-atoms are monotonic, and determining this might be costly; notice, however, that absence of $\uplus$ in (3.2) is a simple syntactic criterion which implies monotonicity of a dl-atom (cf. also Example 3.3.1).

For any dl-program $KB = (L, P)$, we denote by DL_P the set of all ground dl-atoms that occur in $ground(P)$. We assume that KB has an associated set $DL_P^+ \subseteq DL_P$ of ground dl-atoms which are known to be monotonic, and we denote by $DL_P^? = DL_P \setminus DL_P^+$ the set of all others. An *input literal* of $a \in DL_P$ is a ground literal with an input predicate of a and constant symbols in Φ.

Definition 3.3.1 *A* stratification of $KB = (L, P)$ (w.r.t. DL_P^+) *is a mapping* $\lambda\colon HB_P \cup DL_P \to \{0, 1, \ldots, k\}$ *such that*

(i) $\lambda(H(r)) \geq \lambda(l')$ *(resp.,* $\lambda(H(r)) > \lambda(l')$*) for each* $r \in ground(P)$ *and* $l' \in B^+(r)$ *(resp.* $l' \in B^-(r)$*), and*

(ii) $\lambda(a) \geq \lambda(l)$ *(resp.,* $\lambda(a) > \lambda(l)$*) for each input literal* l *of each* $a \in DL_P^+$ *(resp.* $a \in DL_P^?$*),*

and $k \geq 0$ *is its* length. *For* $i \in \{0, \ldots, k\}$*, let* $KB_i = (L, P_i) = (L, \{r \in ground(P) \mid \lambda(H(r)) = i\})$*, and let* HB_{P_i} *(resp.* $HB^\star_{P_i}$*) be the set of all* $l \in HB_P$ *such that* $\lambda(l) = i$ *(resp.* $\lambda(l) \leq i$*).*

A dl-program $KB = (L, P)$ is *stratified* iff it has a stratification λ of some length $k \geq 0$. We define its iterative least models $M_i \subseteq HB_P$ with $i \in \{0, \ldots, k\}$ as follows:

(i) M_0 is the least model of KB_0;

(ii) if $i > 0$, then M_i is the least model of KB_i such that $M_i | HB^\star_{P_{i-1}} = M_{i-1} | HB^\star_{P_{i-1}}$.

We say KB is *consistent*, if every M_i with $i \in \{0, \ldots, k\}$ exists, and KB is *inconsistent* otherwise. If KB is consistent, then M_{KB} denotes M_k. Notice that M_{KB} is well-defined, since it does not depend on a particular λ (cf. Corollary 3.4.4). The following result shows that M_{KB} is in fact a minimal model of KB.

Theorem 3.3.3 *Let* $KB = (L, P)$ *be a stratified dl-program. Then,* M_{KB} *is a minimal model of* KB*.*

Proof. Let $\lambda\colon HB_P \cup DL_P \to \{0, 1, \ldots, k\}$ be a stratification of $KB = (L, P)$ relative to DL_P^+. Recall that M_0 is the least model (and thus in particular a model) of $KB_0 = (L_0, P_0)$ and for every $i \in \{1, \ldots, k\}$, it holds that M_i is the least model (and thus in particular a model) of $KB_i = (L_i, P_i)$ such that $M_i | {HB_{P_{i-1}}}^\star = M_{i-1} | {HB_{P_{i-1}}}^\star$. It thus follows that $M_k = M_{KB}$ is a model of KB. We next show that M_k is also a minimal model of KB. Towards a contradiction, suppose that there exists a model $J \subseteq HB_P$ of KB such that $J \subset M_k$. Hence, there exists some $i \in \{0, 1, \ldots, k\}$ such that $J | {HB_{P_i}}^\star \neq J | {HB_{P_i}}^\star$. Let j be a minimal such i. Then, J is a model of KB_j. Furthermore, if $j > 0$, then $J | HB^\star_{P_{j-1}} = J | {HB_{P_{j-1}}}^\star$. But this contradicts M_j being the least model of KB_j such that $M_j | {HB_{P_{j-1}}}^\star = M_{j-1} | {HB_{P_{j-1}}}^\star$. This shows that M_k is a minimal model of KB. □

Example 3.3.2 Consider the dl-program $KB = (L, P)$ given by the rules and facts from Example 3.2.1, but without the rules (6) and (7). This program has a stratification of length 2, with the associated set DL_P^+ comprising all dl-atoms occurring in P. The least model of P contains all review candidates of the given papers, together with error flags for them, because no paper is assigned so far. ◇

3.4 Answer-Set Semantics for dl-Programs

Having defined the least model semantics for dl-programs, we are able to specify their answer-set semantics, akin to the original definition by Gelfond and Lifschitz, which is also given in such a constructive manner.

3.4.1 Strong Answer-Set Semantics of dl-Programs

We now define the *strong answer-set semantics* of general dl-programs, which is reduced to the least model semantics of positive dl-programs. We use a generalized transformation that removes all NAF-literals and all dl-atoms except for those known to be monotonic. If we ignore this knowledge and remove all dl-atoms, then we arrive at the *weak answer-set semantics* of KB (see next subsection).

In the sequel, let $KB = (L, P)$ be a dl-program, and let DL_P, DL_P^+, and $DL_P^?$ be as above.

Definition 3.4.1 *The* strong dl-transform *of P relative to L and an interpretation $I \subseteq HB_P$, denoted sP_L^I, is the set of all dl-rules obtained from ground(P) by*

(i) *deleting every dl-rule r such that either $I \not\models_L a$ for some $a \in B^+(r) \cap DL_P^?$, or $I \models_L l$ for some $l \in B^-(r)$, and*

(ii) *deleting from each remaining dl-rule r all literals in $B^-(r) \cup (B^+(r) \cap DL_P^?)$.*

Notice that (L, sP_L^I) has only monotonic dl-atoms and no NAF-literals anymore. Thus, (L, sP_L^I) is a positive dl-program, and by Corollary 3.3.2, has a least model if it is satisfiable.

Definition 3.4.2 *Let $KB = (L, P)$ be a dl-program. A* strong answer set *of KB is an interpretation $I \subseteq HB_P$ such that I is the least model of (L, sP_L^I).*

Example 3.4.1 The dl-program $KB_S = (L_S, P_S)$ of Example 3.2.2 has the following three strong answer sets (quoting only relevant atoms):

$\{supplied(s_3, case);\ supplied(s_2, cpu);\ supplied(s_2, harddisk);\ rebate(s_2);\ \ldots\};$
$\{supplied(s_3, case);\ supplied(s_3, harddisk);\ rebate(s_3);\ \ldots\};$
$\{supplied(s_3, case);\ \ldots\}.$

Since the supplier s_3 was already fixed for the part *case*, two possibilities for a discount remain ($rebate(s_2)$ or $rebate(s_3)$; s_1 is not offering the needed part *harddisk*, and the shop will not give a discount only for the part *cpu*). ◇

Example 3.4.2 The dl-program $KB_N = (L_N, P_N)$ of Example 3.2.3 has the following four strong answer sets:

$\{overloaded(n_2);\ connect(add_1, n_5);\ connect(add_2, n_4);\ \ldots\};$
$\{overloaded(n_2);\ connect(add_1, n_1);\ connect(add_2, n_4);\ \ldots\};$
$\{overloaded(n_2);\ connect(add_1, n_1);\ connect(add_2, n_5);\ \ldots\};$
$\{overloaded(n_2);\ connect(add_1, n_5);\ connect(add_2, n_1);\ \ldots\}.$

Node n_2 is already in concept *HighTrafficNode* in the original L_N, so it is contained in each answer set. From the remaining nodes in L_N, n_2 and n_3 would be in *HighTrafficNode* with one additional connection, so they are avoided. This leaves only nodes n_1, n_4, and n_5 to be connection candidates for the new nodes. The connection from add_1 to n_4 is explicitly inhibited in P_N, the remaining connection possibilities are represented by the four answer sets. ◇

The following result shows that the strong answer-set semantics of a dl-program $KB = (L, P)$ without dl-atoms coincides with the ordinary answer-set semantics of P.

Theorem 3.4.1 *Let $KB = (L, P)$ be a dl-program without dl-atoms. Then, $I \subseteq HB_P$ is a strong answer set of KB iff it is an answer set of the ordinary program P.*

Proof. Let $I \subseteq HB_P$. If KB is free of dl-atoms, then $sP_L^I = P^I$. Hence, I is the least model of (L, sP_L^I) iff I is the least model of P^I. Thus, I is a strong answer set of KB iff I is an answer set of P. □

The next result shows that, as desired, strong answer sets of a dl-program KB are also models, and moreover minimal models if all dl-atoms are monotonic (and known as such).

Theorem 3.4.2 *Let $KB = (L, P)$ be a dl-program, and let M be a strong answer set of KB. Then, (a) M is a model of KB, and (b) M is a minimal model of KB if $DL_P = DL_P^+$.*

Proof. (a) Let I be a strong answer set of KB. To show that I is also a model of KB, we have to show that $I \models_L r$ for all $r \in ground(P)$. Consider any $r \in ground(P)$. Suppose that $I \models_L l$ for all $l \in B^+(r)$ and $I \not\models_L l$ for all $l \in B^-(r)$. Then, the dl-rule r' that is obtained from r by removing all the literals in $B^-(r) \cup (B^+(r) \cap DL_P^?)$ is contained in sP_L^I. Since I is a least model of (L, sP_L^I) and thus in particular a model of (L, sP_L^I), it follows that I is a model of r'. Since $I \models_L l$ for all $l \in B^+(r')$ and $I \not\models_L l$ for all $l \in B^-(r') = \emptyset$, it follows that $I \models_L H(r)$. This shows that $I \models_L r$. Hence, I is a model of KB.

(b) By (a), every strong answer set I of KB is a model of KB. Assume that every dl-atom in DL_P is monotonic relative to KB. We now show that then I is also a minimal model of KB. Towards a contradiction, suppose the contrary. That is, there exists a model J of KB such that $J \subset I$. Since J is a model of KB, it follows that J is also a model of (L, sP_L^J). Since every dl-atom in DL_P is monotonic relative to KB, it then follows that $sP_L^I \subseteq sP_L^J$. Hence, J is also a model of (L, sP_L^I). But this contradicts I being the least model of (L, sP_L^I). This shows that I is a minimal model of KB. □

The following theorem shows that positive and stratified dl-programs have at most one strong answer set, which coincides with the canonical minimal model M_{KB}.

Theorem 3.4.3 *Let $KB = (L, P)$ be a (a) positive (resp. (b) stratified) dl-program. If KB is satisfiable (resp. consistent), then M_{KB} is the only strong answer set of KB. If KB is unsatisfiable (resp. inconsistent), then KB has no strong answer set.*

Proof. (a) If $KB = (L, P)$ is satisfiable, then M_{KB} is defined. A strong answer set of KB is an interpretation $I \subseteq HB_P$ such that I is the least model of (L, sP_L^I). Since KB is a positive dl-program, it follows that sP_L^I coincides with $ground(P)$. Hence, $I \subseteq HB_P$ is a strong answer set of KB iff $I = M_{KB}$. If KB is unsatisfiable, then KB has no model. Thus, by Theorem 3.4.2, KB has no strong answer set.

(b) Let λ be a stratification of KB of length $k \geq 0$. Suppose that $I \subseteq HB_P$ is a strong answer set of KB. That is, I is the least model of (L, sP_L^I). Hence,

- $I|HB^\star_{P_0}$ is the least among all models $J \subseteq HB^\star_{P_0}$ of (L, sP_{0L}^I); and
- if $i > 0$, then $I|HB^\star_{P_i}$ is the least among all models $J \subseteq HB^\star_{P_i}$ of (L, sP_{iL}^I) with $J|HB^\star_{P_{i-1}} = I|HB^\star_{P_{i-1}}$.

It thus follows that

- $I|HB^\star_{P_0}$ is the least among all models $J \subseteq HB^\star_{P_0}$ of KB_0; and
- if $i > 0$, then $I|HB^\star_{P_i}$ is the least among all models $J \subseteq HB^\star_{P_i}$ of KB_i with $J|HB^\star_{P_{i-1}} = I|HB^\star_{P_{i-1}}$.

Hence, KB is consistent, and $I = M_{KB}$. Since the above line of argumentation also holds in the converse direction, it follows that $I \subseteq HB_P$ is a strong answer set of KB iff KB is consistent and $I = M_{KB}$. □

Since the strong answer sets of a stratified dl-program KB are independent of the stratification λ of KB, we thus obtain that consistency of KB and M_{KB} are independent of λ.

Corollary 3.4.4 *Let KB be a stratified dl-program. Then, the notion of consistency of KB and the model M_{KB} do not depend on the stratification of KB.*

Example 3.4.3 Consider now the full dl-program from Example 3.2.1. This program is not stratified, in view of rules (6) and (7), which take care of the selection between the different candidates for being reviewers. Each strong answer set containing no error flags corresponds to an acceptable review assignment scenario. ◇

3.4.2 Weak Answer-Set Semantics of dl-Programs

We finally introduce the *weak answer-set semantics* of dl-programs, which associates with a dl-program a larger set of models than the strong answer-set semantics. It is based on a generalized transformation that removes all dl-atoms and NAF-literals, and reduces to the answer-set semantics of ordinary programs.

In the sequel, let $KB = (L, P)$ be a dl-program. The *weak dl-transform* of P relative to L and to an interpretation $I \subseteq HB_P$, denoted wP_L^I, is the ordinary positive program obtained from $ground(P)$ by

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

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

Observe that wP_L^I is an ordinary ground positive program, which does not contain any dl-atoms anymore, and which also does not contain any NAF-literals anymore. We thus define the weak answer-set semantics by reduction to the least model semantics of ordinary ground positive programs as follows.

Definition 3.4.3 *Let $KB = (L, P)$ be a dl-program. A* weak answer set *of KB is an interpretation $I \subseteq HB_P$ such that I is the least model of the ordinary positive program wP_L^I.*

The following result shows that the weak answer-set semantics of a dl-program $KB = (L, P)$ without dl-atoms coincides with the ordinary answer-set semantics of P.

Theorem 3.4.5 *Let $KB = (L, P)$ be a dl-program without dl-atoms. Then, $I \subseteq HB_P$ is a weak answer set of KB iff it is an answer set of the ordinary normal program P.*

Proof. Let $I \subseteq HB_P$. If KB is free of dl-atoms, then $wP_L^I = P^I$. Thus, I is the least model of wP_L^I iff I is the least model of P^I. So, I is a weak answer set of KB iff I is an answer set of P. □

The following result shows that every weak answer set of a dl-program KB is also a model of KB. Note that differently from strong answer sets, the weak answer sets of KB are generally not minimal models of KB, even if KB has only monotonic dl-atoms.

Theorem 3.4.6 *Let KB be a dl-program. Then, every weak answer set of KB is also a model of KB.*

Proof. Let $I \subseteq HB_P$ be a weak answer set of $KB = (L, P)$. To show that I is also a model of KB, we have to show that $I \models_L r$ for all $r \in \mathit{ground}(P)$. Consider any $r \in \mathit{ground}(P)$. Suppose that $I \models_L l$ for all $l \in B^+(r)$ and $I \not\models_L l$ for all $l \in B^-(r)$. Then, the dl-rule r' that is obtained from r by removing all the dl-atoms in $B^+(r)$ and all literals in $B^-(r)$ is in wP_L^I. Since I is the least model of wP_L^I and thus in particular a model of wP_L^I, it follows that $I \models_L r'$. Since $I \models_L l$ for all $l \in B^+(r')$ and $I \not\models_L l$ for all $l \in B^-(r') = \emptyset$, it follows $I \models_L H(r') = H(r)$. This shows that $I \models_L r$. Thus, I is a model of KB. □

The following result shows that the weak answer-set semantics of dl-programs can be expressed in terms of a reduction to the answer-set semantics of ordinary normal programs.

Theorem 3.4.7 *Let $KB = (L, P)$ be a dl-program. Let $I \subseteq HB_P$ and let P_L^I be obtained from $\mathit{ground}(P)$ by (i) deleting every dl-rule r where either $I \not\models_L a$ for some dl-atom $a \in B^+(r)$, or $I \models_L a$ for some dl-atom $a \in B^-(r)$, and (ii) deleting from every remaining dl-rule r every dl-atom in $B^+(r) \cup B^-(r)$. Then, I is a weak answer set of KB iff I is an answer set of P_L^I.*

Proof. Immediate by the observation that $wP_L^I = (P_L^I)^I$. □

Finally, the next result shows that the set of all strong answer sets of a dl-program KB is contained in the set of all weak answer sets of KB. Intuitively, the additional information about the monotonicity of dl-atoms that we use for specifying strong answer sets allows for focusing on a smaller set of models. Hence, the set of all weak answer sets of KB can be seen as an approximation of the set of all strong answer sets of KB. Note that the converse of the following theorem generally does not hold. That is, there exist dl-programs KB, which have a weak answer set that is not a strong answer set.

Theorem 3.4.8 *Every strong answer set of a dl-program KB is also a weak answer set of KB.*

Proof. Let $I \subseteq HB_P$ be a strong answer set of $KB = (L, P)$. That is, I is the least model of (L, sP_L^I). Hence, I is also a model of wP_L^I. We now show that I is in fact the least model of wP_L^I. Towards a contradiction, assume the contrary. That is, there exists a model $J \subset I$ of wP_L^I. Hence, J is also a model of (L, sP_L^I). But this contradicts I being the least model of (L, sP_L^I). This shows that I the least model of wP_L^I. That is, I is a weak answer set of KB. □

3.5 Well-Founded Semantics for dl-Programs

The well-founded semantics, introduced by van Gelder et al. [1991], represents another widely used semantics for ordinary non-monotonic logic programs, choosing a more tentative approach concerning derivable knowledge. It is a skeptical approximation of the answer set semantics in the sense that every well-founded consequence of a given ordinary normal program P is contained in every answer set of P. While the answer-set semantics resolves conflicts by virtue of permitting multiple intended models as alternative scenarios, the well-founded semantics remains agnostic in the presence of conflicting information, assigning the truth value *false* to a maximal set of atoms that cannot become true during the evaluation of a given program. Furthermore, well-founded semantics assigns a coherent meaning to *all* programs, while some programs are not consistent under answer-set semantics, i.e., lack an answer set.

Another important aspect of the well-founded semantics is that it is geared towards efficient *query answering* and also plays a prominent role in deductive databases (see, e.g., [May et al., 1997] for a proposal for object-oriented deductive databases, which is applied to the Florid system implementing F-Logic). As an important computational property, a query to an ordinary normal program is evaluable under well-founded semantics in polynomial time (under data complexity), while the query answering under the answer-set semantics is intractable in general. Finally, efficient implementations of the well-founded semantics exist, such as the XSB system [Rao et al., 1997] and SMODELS [Niemelä et al., 2000].

3.5.1 Original Definition of the Well-Founded Semantics

The well-founded semantics has many different equivalent definitions, cf. [van Gelder et al., 1991, Baral and Subrahmanian, 1993]. We recall here the one based on unfounded sets.

Let P be a program. *Ground terms*, *atoms*, *literals*, etc., are defined as usual. We denote by HB_P the *Herbrand base* of P, i.e., the set of all ground atoms with predicate and constant symbols from P (if no latter exists, with an arbitrary constant symbol c from Φ), and by $ground(P)$ the set of all ground instances of rules in (w.r.t. HB_P).

For literals $l = a$ (resp., $l = \neg a$), we use $\neg.l$ to denote $\neg a$ (resp., a), and for sets of literals S, we define $\neg.S = \{\neg.l \mid l \in S\}$ and $S^+ = \{a \in S \mid a \text{ is an atom}\}$. We use $Lit_P = HB_P \cup \neg.HB_P$ to denote the set of all ground literals with predicate and constant symbols from P. A set $S \subseteq Lit_P$ is *consistent* iff $S \cap \neg.S = \emptyset$. A *three-valued interpretation* relative to P is any consistent $I \subseteq Lit_P$.

Definition 3.5.1 *A set $U \subseteq HB_P$ is an* unfounded set *of P relative to I, if for every $a \in U$ and every $r \in ground(P)$ with $H(r) = a$, either*

(i) *$\neg b \in I \cup \neg.U$ for some atom $b \in B^+(r)$, or*

(ii) $b \in I$ *for some atom* $b \in B^-(r)$.

There exists the greatest unfounded set of P relative to I, denoted $U_P(I)$. Intuitively, if I is compatible with P, then all atoms in $U_P(I)$ can be safely switched to false and the resulting interpretation is still compatible with P.

The operators T_P and W_P on consistent $I \subseteq Lit_P$ are then defined by:

- $T_P(I) = \{H(r) \mid r \in ground(P),\ B^+(r) \cup \neg.B^-(r) \subseteq I\}$;
- $W_P(I) = T_P(I) \cup \neg.U_P(I)$.

The operator W_P is monotonic, and thus has a least fixpoint, denoted $lfp(W_P)$, which is the *well-founded semantics* of P, denoted $WFS(P)$. An atom $a \in HB_P$ is *well-founded* (resp., *unfounded*) w.r.t. P, if a (resp., $\neg a$) is in $lfp(W_P)$. Intuitively, starting from scratch ($I = \emptyset$), rules are applied to obtain new positive and negated facts (via $T_P(I)$ and $\neg.U_P(I)$, respectively). This process is repeated until no longer possible.

Example 3.5.1 Consider the propositional program

$$P = \{p \leftarrow not\ q;\ q \leftarrow p;\ p \leftarrow not\ r\}.$$

For $I = \emptyset$, we have $T_P(I) = \emptyset$ and $U_P(\emptyset) = \{r\}$: p cannot be unfounded because of the first rule and condition (ii), and hence q cannot be unfounded because of the second rule and condition (i). Thus, $W_P(\emptyset) = \{\neg r\}$. Since $T_P(\{\neg r\}) = \{p\}$ and $U_P(\{\neg r\}) = \{r\}$, it follows $W_P(\{\neg r\}) = \{p, \neg r\}$. Since $T_P(\{p, \neg r\}) = \{p, q\}$ and $U_P(\{p, \neg r\}) = \{r\}$, it then follows $W_P(\{p, \neg r\}) = \{p, q, \neg r\}$. Thus, $lfp(W_P) = \{p, q, \neg r\}$. That is, r is unfounded relative to P, and the other atoms are well-founded. ◇

3.5.2 Generalizing Unfounded Sets

In this section, we define the well-founded semantics for dl-programs. We do this by generalizing the well-founded semantics for ordinary normal programs. More specifically, we generalize the definition based on unfounded sets given in the previous Subsection.

In the sequel, let $KB = (L, P)$ be a dl-program. We first define the notion of an unfounded set for dl-programs.

Definition 3.5.2 *Let $I \subseteq Lit_P$ be consistent. A set $U \subseteq HB_P$ is an* unfounded set *of KB relative to I iff the following holds: for every $a \in U$ and every $r \in ground(P)$ with $H(r) = a$, either*

(i) $\neg b \in I \cup \neg.U$ *for some ordinary atom* $b \in B^+(r)$, *or*

(ii) $b \in I$ *for some ordinary atom* $b \in B^-(r)$, *or*

(iii) *for some dl-atom* $b \in B^+(r)$, *it holds that* $S^+ \not\models_L b$ *for every consistent* $S \subseteq Lit_P$ *with* $I \cup \neg.U \subseteq S$, *or*

(iv) $I^+ \models_L b$ *for some dl-atom* $b \in B^-(r)$.

What is new here are the conditions (iii) and (iv). Intuitively, (iv) says that *not* b is for sure false, regardless of how I is further expanded, while (iii) says that b will never become true, if we expand I in a way such that all unfounded atoms are false. The following examples illustrate the concept of an unfounded set for dl-programs.

Example 3.5.2 Consider $KB_2 = (L_2, P_2)$, where $L_2 = \{S \sqsubseteq C\}$ and P_2 is as follows:

$$p(a) \leftarrow DL[S \uplus q; C](a); \qquad q(a) \leftarrow p(a); \qquad r(a) \leftarrow not\, q(a),\ not\, s(a).$$

Here, $S_1 = \{p(a), q(a)\}$ is an unfounded set of KB_2 relative to $I = \emptyset$, since $p(a)$ is unfounded due to (iii), while $q(a)$ is unfounded due to (i). The set $S_2 = \{s(a)\}$ is trivially an unfounded set of KB_2 relative to I, since no rule defining $s(a)$ exists.

Relative to $J = \{q(a)\}$, S_1 is not an unfounded set of KB_2 (for $p(a)$, the condition fails) but S_2 is. The set $S_3 = \{r(a)\}$ is another unfounded set of KB_2 relative to J. ◇

Example 3.5.3 Consider the dl-program $KB_3 = (L_2, P_3)$ where P_3 results by negating the dl-literal in P_2. Then $S_1 = \{p(a), q(a)\}$ is not an unfounded set of KB_3 relative to $I = \emptyset$ (condition (iv) fails for $p(a)$), but $S_2 = \{s(a)\}$ is. Relative to $J = \{q(a)\}$, however, both S_1 and S_2 as well as $S_3 = \{r(a)\}$ are unfounded sets of KB_3. ◇

Example 3.5.4 The unfounded set of $KB_1 = (L_1, P_1)$ in Example 3.2.2 w.r.t. $I_0 = \emptyset$ contains $buy\_cand(s_1, harddisk)$, $buy\_cand(s_2, case)$, and $buy\_cand(s_3, cpu)$ due to (iii), since the dl-atom in line (5) of P_1 will never evaluate to true for these pairs. It reflects the intuition that the concept *provides* narrows the choice for buying candidates. ◇

The following lemma implies that KB has a greatest unfounded set relative to I.

Lemma 3.5.1 *Let $KB = (L, P)$ be a dl-program, and let $I \subseteq Lit_P$ be consistent. Then, the set of unfounded sets of KB relative to I is closed under union.*

Proof. Let U and U' be two unfounded sets of a dl-program KB relative to I. An atom $a \in U$ must then fulfill at least one of the conditions of Definition 3.5.2. Is is clear that this is still the case if we replace U by $U \cup U'$ in (i)–(iv). □

We now generalize the operators T_P, U_P, and W_P to dl-programs as follows. We define the operators T_{KB}, U_{KB}, and W_{KB} on all consistent $I \subseteq Lit_P$ by:

- $a \in T_{KB}(I)$ iff $a \in HB_P$ and some $r \in ground(P)$ exists such that
  - (a) $H(r) = a$,
  - (b) $I^+ \models_L b$ for all $b \in B^+(r)$,
  - (c) $\neg b \in I$ for all ordinary atoms $b \in B^-(r)$, and
  - (d) $S^+ \not\models_L b$ for each consistent $S \subseteq Lit_P$ with $I \subseteq S$, for all dl-atoms $b \in B^-(r)$;
- $U_{KB}(I)$ is the greatest unfounded set of KB relative to I;
- $W_{KB}(I) = T_{KB}(I) \cup \neg.U_{KB}(I)$.

The following result shows that the three operators are all monotonic.

Lemma 3.5.2 *Let KB be a dl-program. Then, T_{KB}, U_{KB}, and W_{KB} are monotonic.*

Proof. Let $I \subseteq I' \subseteq HB_P$ be consistent interpretations. Consider any $r \in ground(P)$. Then, for every $a \in T_{KB}(I)$, conditions (a)–(d) still hold for I'. Similar to the proof of Lemma 3.5.1, if we replace I by I' in (i)–(iv) of Definition 3.5.2, the conditions still hold for any $a \in U_{KB}(I)$. Since both T_{KB} and U_{KB} are monotonic, also $W_{KB}(I)$ must be monotonic. □

Thus, in particular, W_{KB} has a least fixpoint, denoted $lfp(W_{KB})$. The well-founded semantics of dl-programs can thus be defined as follows.

Definition 3.5.3 *Let $KB = (L, P)$ be a dl-program. The* well-founded semantics *of KB, denoted $WFS(KB)$, is defined as $lfp(W_{KB})$. An atom $a \in HB_P$ is* well-founded *(resp.,* unfounded*) relative to KB iff a (resp., $\neg a$) belongs to $WFS(KB)$.*

The following examples illustrate the well-founded semantics of dl-programs.

Example 3.5.5 Consider KB_2 of Example 3.5.2. For $I_0 = \emptyset$, we have $T_{KB_2}(I_0) = \emptyset$ and $U_{KB_2}(I_0) = \{p(a), q(a), s(a)\}$. Hence, $W_{KB_2}(I_0) = \{\neg p(a), \neg q(a), \neg s(a)\}$ $(=I_1)$. In the next iteration, $T_{KB_2}(I_1) = \{r(a)\}$ and $U_{KB_2} = \{p(a), q(a), s(a)\}$. Thus, $W_{KB_2}(I_1) = \{\neg p(a), \neg q(a), r(a), \neg s(a)\}$ $(=I_2)$. Since I_2 is total and W_{KB_2} is monotonic, it follows $W_{KB_2}(I_2) = I_2$ and hence $WFS(KB_2) = \{\neg p(a), \neg q(a), r(a), \neg s(a)\}$. Accordingly, $r(a)$ is well-founded and all other atoms are unfounded relative to KB_2. Note that KB_2 has the unique answer set $I = \{r(a)\}$. ◇

Example 3.5.6 Now consider KB_3 of Example 3.5.3. For $I_0 = \emptyset$, we have $T_{KB_3}(I_0) = \emptyset$ and $U_{KB_3}(I_0) = \{s(a)\}$. Hence, $W_{KB_3}(I_0) = \{\neg s(a)\}$ $(=I_1)$. In the next iteration, we have $T_{KB_3}(I_1) = \emptyset$ and $U_{KB_3}(I_1) = \{s(a)\}$. Then, $W_{KB_3}(I_1) = I_1$ and $WFS(KB_3) = \{\neg s(a)\}$; Thus, the atom $s(a)$ is unfounded relative to KB_3. Note that KB_3 has no answer set. ◇

Example 3.5.7 Consider again $U_{KB_1}(I_0 = \emptyset)$ of Example 3.5.4. $W_{KB_1}(I_0)$ consists of $\neg.U_{KB_1}(I_0)$ and all facts of P_1. This input to the first iteration along with (iii) applied to line (8) adds those *supplied* atoms to $U_{KB_1}(I_1)$ that correspond to the (negated) *buy_cand* atoms of $U_{KB_1}(I_0)$. Then, $T_{KB_1}(I_1)$ contains *exclude*(*case*) which forces additional *buy_cand* atoms into $U_{KB_1}(I_2)$, regarding (i) and line (5). The same unfounded set thereby includes *rebate*(s_1), stemming from line (4). As a consequence, *avoid*(s_1) is in $T_{KB_1}(I_3)$. Eventually, the final $WFS(KB_1)$ is not able to make any positive assumption about choosing a new vendor (*buy_cand*), but it is clear about s_1 being definitely not able to contribute to a discount situation, since a supplier for *case* is already chosen in L_1, and s_1 offers only a single further part. ◇

3.5.3 Semantic Properties

In this section, we describe some semantic properties of the well-founded semantics for dl-programs. An immediate result is that it conservatively extends the well-founded semantics for ordinary normal programs.

Theorem 3.5.3 *Let $KB = (L, P)$ be a dl-program without dl-atoms. Then, the well-founded semantics of KB coincides with the well-founded semantics of P.*

Proof. Trivial, since the conditions (i) and (ii) of Definitions 3.5.1 and 3.5.2 coincide. □

The next result shows that the well-founded semantics of a dl-program $KB = (L, P)$ is a partial model of KB. Here, a consistent $I \subseteq Lit_P$ is a *partial model* of KB iff some consistent $J \subseteq Lit_P$ exists such that (i) $I \subseteq J$, (ii) J^+ is a model of KB, and (iii) J is *total*, i.e., $J^+ \cup (\neg.J)^+ = HB_P$. Intuitively, the three-valued I can be extended to a (two-valued) model $I' \subseteq HB_P$ of KB.

Theorem 3.5.4 *Let KB be a dl-program. Then, $WFS(KB)$ is a partial model of KB.*

Proof. For this proof we adopt the notion of *weak falsification* by van Gelder et al. [1991]. An ground rule r is *weakly falsified* in a partial or total interpretation I, if $H(r)$ is false in I but no literal in $B(r)$ is false in I. We will first show that no rule $r \in ground(P)$ is weakly falsified in $WFS(KB)$. Let r be any rule $r \in ground(P)$ with $H(r) = p$ such that

$\neg p \in W^n_{KB}$, where W^n_{KB} is an element of the sequence of $lfp(W_{KB})$. We need to show that $B(r)$ is false in W^n_{KB}. By definition, $p \in U_{KB}(W^m_{KB})$ for some $m < n$. By Lemma 3.5.2, $W^m_{KB} \subseteq W^{m+1}_{KB} \subseteq W^n_{KB}$. Either $W^m_{KB} \not\models B(r)$ or $B(r) \cap U^m_{KB} \neq \emptyset$, i.e., some atom q in $B(r)$ is unfounded, which would result in $\neg q \in W^{m+1}_{KB}$, so $B(r)$ is false in W^{m+1}_{KB}. In either case it follows that $B(r)$ is false in W^n_{KB}. Hence, r is not weakly falsified in $WFS(KB)$. Since r was chosen arbitrarily, any W^n_{KB} and therefore also $WFS(KB)$ does not weakly falsify any ground rule r in KB. To show that $WFS(KB)$ is a partial model, we add to it all atoms in HB_P that are neither true nor false in $WFS(KB)$ and denote the resulting set with M. If $WFS(KB)$ satisfies a rule $r \in ground(P)$, then also M must satisfy it. Thus, M is a total model and thus, $WFS(KB)$ is a partial model. □

Like in the case of ordinary normal programs, the well-founded semantics for positive and stratified dl-programs is total and coincides with their least model semantics and iterative least model semantics, respectively. This result can be elegantly proved using a characterization of the well-founded semantics given in the next section.

Theorem 3.5.5 *Let $KB = (L, P)$ be a dl-program. If KB is positive (resp. stratified), then (a) every ground atom $a \in HB_P$ is either well-founded or unfounded relative to KB, and (b) $WFS(KB) \cap HB_P$ is the least model (resp. the iterative least model) of KB, which coincides with the unique strong answer set of KB.*

Proof. First, we show that the well-founded model of a positive (ground) dl-program KB is a total model. T_{KB} directly corresponds to the immediate consequence operator of the least model semantics of a positive program, thus $WFS(KB) \cap HB_P$ is the least model of KB. It is easy to see that all atoms in HB_P which are not in the greatest unfounded set must be captured by the fixpoint of the operator T_{KB}. Hence, every atom in HB_P is either well-founded or unfounded.

Due to the definition of the operator T_{KB}, the evaluation of rules in P follow the stratification of P, i.e., rules r with $\lambda(H(r)) = n$ do not affect the result of strata lower than n. For stratified ground dl-programs KB we prove by induction on the number of strata in KB, showing that for any atom p in stratum k, p is in the stratified model whenever it is in $WFS(KB)$ and p is not in the stratified model whenever $\neg p$ is in $WFS(KB)$. For stratum $k = 0$ the claim follows immediately from the first part of the proof, since this stratum must be a positive program.

Let $M^\star = lfp(T_{KB})$. We can then define another monotonic operator $T'_{KB}(I) = T_{KB'}(M^\star \cap P_0 \cup I)$, where $P_0 = \{a \in HB_P \mid \lambda(a) = 0\}$, $KB' = (L, P')$, and P' is obtained from P as follows: An atom p in stratum 1 must occur in the head of a rule r with a nonempty body $B(r)$. For all such rules, we carry out the following transformation:

- If an atom $b \in B^-(r)$ is already known to be well-founded or $b \in B^+(r)$ is unfounded, we can remove this rule from the program;
- if $b \in B^+(r)$ is well-founded or $b \in B^-(r)$ is unfounded, we can remove *not* b from the body of the rule.

Thus, KB' has one stratum less than the original program.

It must hold that $lfp(T_{KB}) = lfp(T'_{KB})$, i.e., when the model of the lowest stratum is known, it can be added to the program as facts, reducing the fixpoint iterations. We can proceed with this reduction through all strata.

It has been proven in Theorem 3.4.3 that the least model of a positive (resp. stratified) dl-program coincides with its strong answer set. □

Example 3.5.8 The dl-program KB_2 in Example 3.5.2 is stratified (intuitively, recursion through negation is acyclic) while KB_3 in Example 3.5.3 is not. The result computed in Example 3.5.5 verifies the conditions of Theorem 3.5.5. ◇

The following result shows that we can limit ourselves to dl-programs in *dl-query form*, where dl-atoms equate designated predicates. Formally, a dl-program $KB = (L, P)$ is in *dl-query form*, if each $r \in P$ involving a dl-atom is of the form $a \leftarrow b$, where b is a dl-atom. Any dl-program $KB = (L, P)$ can be transformed into a dl-program $KB^{dl} = (L, P^{dl})$ in dl-query form. Here, P^{dl} is obtained from P by replacing every dl-atom $a(\mathbf{t})$ (with a of the form $DL[\cdots]$) by $p_a(\mathbf{t})$, and by adding the dl-rule $p_a(X_1, \ldots, X_k) \leftarrow a(X_1, \ldots, X_k)$ to P, where p_a is a fresh predicate whose arity is the number $k \leq 2$ of arguments of $\mathbf{t} = t_1, \ldots, t_k$. Informally, p_a is an abbreviation for a. The following result now shows that KB^{dl} and KB are equivalent under the well-founded semantics. Intuitively, the well-founded semantics tolerates abbreviations in the sense that they do not change the semantics of a dl-program.

Theorem 3.5.6 *Let $KB = (L, P)$ be a dl-program. Then, $WFS(KB) = WFS(KB^{dl}) \cap Lit_P$.*

Proof. The transformation only affects rules with dl-atoms in their bodies. For any dl-atom b in the body of such a rule r, if b is true (resp. false), the head of the replacement rule, $p_a(\mathbf{t})$, is well-founded (resp. unfounded). Thus, the truth value of the body of r is unchanged. Since all replacement heads $p_a(\mathbf{t})$ do not occur in Lit_P, $WFS(KB) = WFS(KB^{dl}) \cap Lit_P$. □

3.5.4 Relationship to Strong Answer-Set Semantics

In this section, we show that the well-founded semantics for dl-programs can be characterized in terms of the least and greatest fixpoint of a monotone operator γ^2_{KB} similar as the well-founded semantics for ordinary normal programs [Baral and Subrahmanian, 1993]. We then use this characterization to derive further properties of the well-founded semantics for dl-programs.

For a dl-program $KB = (L, P)$, define the operator γ_{KB} on interpretations $I \subseteq HB_P$ by

$$\gamma_{KB}(I) = M_{KB^I},$$

i.e., the least model of the positive dl-program $KB^I = (L, sP^I_L)$. The next result shows that γ_{KB} is anti-monotonic, like its counterpart for ordinary normal programs [Baral and Subrahmanian, 1993]. Note that this result holds only if all dl-atoms in P are monotonic.

Proposition 3.5.7 *Let $KB = (L, P)$ be a dl-program. Then, γ_{KB} is anti-monotonic.*

Hence, the operator $\gamma^2_{KB}(I) = \gamma_{KB}(\gamma_{KB}(I))$, for all $I \subseteq HB_P$, is monotonic and thus has a least and a greatest fixpoint, denoted $lfp(\gamma^2_{KB})$ and $gfp(\gamma^2_{KB})$, respectively. We can use these fixpoints to characterize the well-founded semantics of KB.

Theorem 3.5.8 *Let $KB = (L, P)$ be a dl-program. Then, an atom $a \in HB_P$ is well-founded (resp., unfounded) relative to KB iff $a \in lfp(\gamma^2_{KB})$ (resp., $a \notin gfp(\gamma^2_{KB})$).*

Proof. This theorem was proved by Baral and Subrahmanian [1993] with an equivalent definition of the γ operator. □

Example 3.5.9 Consider the dl-program KB_1 from Example 3.2.2. The set $lfp(\gamma^2_{KB_1})$ contains the atoms $avoid(s_1)$ and $supplied(s_3, case)$, while $gfp(\gamma^2_{KB_1})$ does not contain

$rebate(s_1)$. Consequently, $WFS(KB_1)$ contains the literals $avoid(s_1)$, $supplied(s_3, case)$, and $\neg rebate(s_1)$, corresponding to the result of Example 3.5.7 (and, moreover, to the intersection of all answer sets of KB_1). ◇

The next theorem shows that the well-founded semantics for dl-programs approximates their strong answer-set semantics. That is, every well-founded ground atom is true in every answer set, and every unfounded ground atom is false in every answer set.

Theorem 3.5.9 *Let $KB = (L, P)$ be a dl-program. Then, every strong answer set of KB includes all atoms $a \in HB_P$ that are well-founded relative to KB and no atom $a \in HB_P$ that is unfounded relative to KB.*

Proof. We will show that a strong answer set of $KB = (L, P)$ corresponds to a total fixpoint of W_{KB}. Let A be an answer set of KB. Then, $M = A \cup \neg.\{HB_P \setminus A\}$ is a total model of KB. We will show that $A = W_{KB}(A)$ must hold.

Since M is total, each rule in P must be satisfied w.r.t. M. Moreover, each positive atom in this model must occur in some head in P, hence it must be in $T_{KB}(A)$. From the definition of T_{KB} it is straightforward to see that the positive part of A fulfills conditions (a)-(d). The negative part of A are those atoms which are not derivable from rules, hence they must be unfounded. Thus, A is a total fixpoint of W_{KB}. The well-founded model $WFS(KB)$ is a least fixpoint of W_{KB}, therefore $WFS(KB) \subseteq W_{KB}(A)$ must hold. But then, each well-founded atom must also be in every strong answer set of KB and no answer set contains any unfounded atom. □

A ground atom a is a *cautious* (resp., *brave*) *consequence under the strong answer-set semantics* of a dl-program KB iff a is true in every (resp., some) strong answer set of KB. Hence, under the strong answer-set semantics, every well-founded and no unfounded ground atom is a cautious (resp., brave) consequence of KB.

Corollary 3.5.10 *Let $KB = (L, P)$ be a dl-program. Then, under the strong answer-set semantics, every well-founded atom $a \in HB_P$ relative to KB is a cautious (resp., brave) consequence of KB, and no unfounded atom $a \in HB_P$ relative to KB is a cautious (resp., brave) consequence of a satisfiable KB.*

If the well-founded semantics of a dl-program $KB{=}(L, P)$ is total, i.e., contains either a or $\neg a$ for every $a \in HB_P$, then it specifies the only strong answer set of KB.

The following result follows directly from Theorem 3.5.9:

Theorem 3.5.11 *Let $KB = (L, P)$ be a dl-program. If every atom $a \in HB_P$ is either well-founded or unfounded relative to KB, then the set of all well-founded atoms $a \in HB_P$ relative to KB is the only strong answer set of KB.*

Proof. Let us denote the set of atoms that occur positively in $WFS(KB)$ by $WFS^+(KB)$ and the set of atoms that occur negatively in $WFS(KB)$ by $WFS^-(KB)$. Let A be a strong answer set of KB. According to Theorem 3.5.9, $WFS^+(KB) \subseteq A$ and $WFS^-(KB) \cap A = \emptyset$. If $WFS(KB)$ is a total model, we have $WFS^+(KB) \cup WFS^-(KB) = HB_P$. Let us assume that $WFS^+(KB) \subset A$. Then, it must hold that $HB_P \setminus WFS^+(KB) \cap A \neq \emptyset$. But since $HB_P \setminus WFS^+(KB) = WFS^-(KB)$, this contradicts Theorem 3.5.9. Thus, $WFS^+(KB) = A$ and therefore A is the only strong answer set of KB. □

3.6 Computation

In this section, we give fixpoint characterizations for the strong answer set of satisfiable positive and consistent stratified dl-programs, and we show how to compute it by finite fixpoint iterations. Moreover, we give a constructive method for computing the well-founded model of a dl-program. We start with a general guess-and-check algorithm for computing weak answer sets.

3.6.1 General Algorithm for Computing Weak Answer Sets

Computing all weak answer sets of a given (general) dl-program $KB = (L, P)$ can be reduced to computing all answer sets of an ordinary disjunctive program. This is done by a guess-and-check algorithm as follows:

1. We first replace each dl-atom $a(\mathbf{t})$ in P of form
$$DL[S_1 op_1 p_1, \ldots, S_m op_m\ p_m; Q](\mathbf{t}) \tag{3.4}$$
by a globally new atom $d_a(\mathbf{t})$.

2. We then add to the result of Step 1 all ground facts of form
$$d_a(\mathbf{c}) \vee \neg d_a(\mathbf{c}) \leftarrow , \tag{3.5}$$
for each dl-atom a occurring in P and each ground term (resp., pair of ground terms) $\mathbf{c} \in HB_{DL}$. Intuitively, rules of form (3.5) "guess" the truth values of the dl-atoms of P. We denote the resulting program by P_{guess}.

3. We construct the answer sets of P_{guess} and check whether the original "guess" of the truth values of the auxiliary atoms $d_a(\mathbf{c})$ is correct with respect to the given description logic knowledge base L. That is, for each answer set I of P_{guess} and each dl-atom a of form (3.2), we check whether $d_a(\mathbf{c}) \in I$ iff $I \models_L a(\mathbf{c})$. If this condition holds, then $I|HB_P$ is a weak answer set of P.

Note that head disjunction is not indispensable here, because we can express Rule 3.5 also by an unstratified guess.

Theorem 3.6.1 *Let $KB = (L, P)$ be a dl-program, and let $I \subseteq HB_P$. Then, I is a weak answer set of KB iff I can be completed to an answer set $I^\star \subseteq HB_{P_{guess}}$ of P_{guess} such that $d_a(\mathbf{c}) \in I^\star$ iff $I^\star \models_L a(\mathbf{c})$, for all $a(\mathbf{c}) \in DL_P$.*

Proof. Let $P^\star$ be defined in the same way as P_{guess}, except that every rule $d_a(\mathbf{c}) \vee \neg d_a(\mathbf{c}) \leftarrow$ is replaced by the following two rules:

$$d_a(\mathbf{c}) \leftarrow DL[S_1 op_1 p_1, \ldots, S_m op_m\ p_m; Q](\mathbf{c});$$
$$\neg d_a(\mathbf{c}) \leftarrow not\ d_a(\mathbf{c}).$$

Then, $I \subseteq HB_P$ is a weak answer set of KB iff $I^\star$ is a weak answer set of $(L, P^\star)$, where $I^\star \subseteq HB_{P^\star} = HB_{P_{guess}}$ is obtained from I by adding (i) all $d_a(\mathbf{c})$ such that $a(\mathbf{c}) \in DL_P$ and $I \models_L a(\mathbf{c})$, and (ii) all $\neg d_a(\mathbf{c})$ such that $a(\mathbf{c}) \in DL_P$ and $I \not\models_L a(\mathbf{c})$, and conversely I is obtained from $I^\star$ by restriction to HB_P. By Theorem 3.4.7, the latter is equivalent to $I^\star$ being an answer set of $(L, P^{\star I^\star}_L)$, where $P^{\star I^\star}_L$ is defined as in Theorem 3.4.7. This is in turn equivalent to $I^\star$ being an answer set of P_{guess} such that $d_a(\mathbf{c}) \in I^\star$ iff $I^\star \models_L a(\mathbf{c})$, for all $a(\mathbf{c}) \in DL_P$. In summary, $I \subseteq HB_P$ is a weak answer set of KB iff I can be completed to an answer set $I^\star \subseteq HB_{P_{guess}}$ of P_{guess} such that $d_a(\mathbf{c}) \in I^\star$ iff $I^\star \models_L a(\mathbf{c})$, for all $a(\mathbf{c}) \in DL_P$. □

Although this basic algorithm is in general not very efficient and leaves room for improvements, it shows that the weak answer-set semantics can be realized on top of existing answer-set solvers for disjunctive programs like DLV [Eiter et al., 2000b] or GNT [Janhunen et al., 2000].

3.6.2 Fixpoint Semantics

The answer set of an ordinary positive resp. stratified normal program P has a well-known fixpoint characterization in terms of an immediate consequence operator T_P, which gracefully generalizes to analog dl-programs. This can be exploited for a bottom up computation of their strong answer set.

Positive dl-Programs

For any (not necessarily satisfiable) dl-program $KB = (L, P)$, we define the operator T_{KB} on the subsets of HB_P as follows. For every $I \subseteq HB_P$, let

$$T_{KB}(I) = \begin{cases} HB_P, & \text{if } I \text{ is not consistent,} \\ \{H(r) \mid r \in \mathit{ground}(P),\ I \models_L l \text{ for all } l \in B(r)\}, & \text{otherwise.} \end{cases}$$

The following lemma shows that for positive KB, the operator T_{KB} is monotonic, that is, $I \subseteq I' \subseteq HB_P$ implies $T_{KB}(I) \subseteq T_{KB}(I')$. This result is immediate from the fact that in $\mathit{ground}(P)$, for a positive dl-program $KB = (L, P)$, each dl-atom is monotonic relative to KB.

Lemma 3.6.2 *Let $KB = (L, P)$ be a positive dl-program. Then, T_{KB} is monotonic.*

Proof. Let $I \subseteq I' \subseteq HB_P$. Consider any $r \in \mathit{ground}(P)$. Then, for every classical literal $l \in B(r)$, it holds that $I \models_L l$ implies $I' \models_L l$. Furthermore, for every dl-atom $a \in B(r)$, it holds that $I \models_L a$ implies $I' \models_L a$, since a is monotonic relative to KB. This shows that $T_{KB}(I) \subseteq T_{KB}(I')$. □

The next result gives a characterization of the pre-fixpoints of T_{KB}. If KB is satisfiable, then every pre-fixpoint of T_{KB} is either a model of KB, or equal to HB_P. If KB is unsatisfiable, then HB_P is the only pre-fixpoint of T_{KB}. Recall that $I \subseteq HB_P$ is a *pre-fixpoint* of T_{KB} iff $T_{KB}(I) \subseteq I$.

Proposition 3.6.3 *Let $KB = (L, P)$ be a positive dl-program. Then, $I \subseteq HB_P$ is a pre-fixpoint of T_{KB} iff I is either (a) a model of KB or (b) equal to HB_P.*

Proof. ($\Rightarrow$) Assume that $T_{KB}(I) \subseteq I \subseteq HB_P$. Suppose first that I is consistent. Then, for every $r \in \mathit{ground}(P)$, $I \models_L l$ for all $l \in B(r)$ implies that $I \models_L H(r)$, and thus $I \models_L r$. Hence, I is a model of KB. Suppose next that I is not consistent. Then, $T_{KB}(I) = HB_P$, and thus $I = HB_P$.

($\Leftarrow$) Suppose first that I is a model of KB. That is, $I \models_L r$ for all $r \in \mathit{ground}(P)$. Equivalently, $I \models_L l$ for all $l \in B(r)$ implies that $I \models_L H(r)$, for all $r \in \mathit{ground}(P)$. It thus follows that $T_{KB}(I) \subseteq I$. Suppose next that $I = HB_P$. Then, $T_{KB}(I) = HB_P = I$. □

Since every monotonic operator has a least fixpoint, which coincides with its least pre-fixpoint, we immediately obtain the following corollary: The least fixpoint of T_{KB}, denoted $\mathit{lfp}(T_{KB})$, is given by the least model of KB, if KB is satisfiable, and by HB_P, if KB is unsatisfiable.

Corollary 3.6.4 *Let $KB = (L, P)$ be a positive dl-program. Then, (a) $lfp(T_{KB}) = M_{KB}$, if KB is satisfiable, and (b) $lfp(T_{KB}) = HB_P$, if KB is unsatisfiable.*

The next result shows that the least fixpoint of T_{KB} can be computed by a finite fixpoint iteration (which is based on the assumption that P and the number of constant symbols in Φ are finite). Note that for every $I \subseteq HB_P$, we define $T^i_{KB}(I) = I$, if $i = 0$, and $T^i_{KB}(I) = T_{KB}(T^{i-1}_{KB}(I))$, if $i > 0$.

Theorem 3.6.5 *Let KB be a positive dl-program. Then, $lfp(T_{KB}) = \bigcup_{i=1}^{n} T^i_{KB}(\emptyset) = T^n_{KB}(\emptyset)$ for some $n \geq 0$.*

Proof. Since T_{KB} is monotonic and HB_P is finite, it follows that $T^i_{KB}(\emptyset)$ for $i \geq 0$ is an increasing sequence of sets contained in $lfp(T_{KB})$, and $T^n_{KB}(\emptyset) = T^{n+1}_{KB}(\emptyset)$ for some $n \geq 0$. Since $T^n_{KB}(\emptyset)$ is a fixpoint of T_{KB} that is contained in $lfp(T_{KB})$, it follows that $T^n_{KB}(\emptyset) = lfp(T_{KB})$. □

Example 3.6.1 Suppose that P in $KB = (L, P)$ consists of the rules r_1: $b \leftarrow DL[S \uplus p; C](a)$ and r_2: $p(a) \leftarrow$, and L is the axiom $S \sqsubseteq C$. Then, KB is positive, and $lfp(T_{KB}) = \{p(a), b\}$, where $T^0_{KB}(\emptyset) = \emptyset$, $T^1_{KB}(\emptyset) = \{p(a)\}$, and $T^2_{KB}(\emptyset) = \{p(a), b\}$. ◇

Stratified dl-Programs

Using Theorem 3.6.5, we can characterize the answer set M_{KB} of a stratified dl-program KB by a sequence of fixpoint-iterations along a stratification as follows. Let $\widehat{T}^i_{KB}(I) = T^i_{KB}(I) \cup I$, for all $i \geq 0$.

Theorem 3.6.6 *Suppose $KB = (L, P)$ has a stratification λ of length $k \geq 0$. Define the literal sets $M_i \subseteq HB_P$, $i \in \{-1, 0, \ldots, k\}$, as follows: $M_{-1} = \emptyset$ and*

$$M_i = \widehat{T}^{n_i}_{KB_i}(M_{i-1}), \text{ where } n_i \geq 0 \text{ such that } \widehat{T}^{n_i}_{KB_i}(M_{i-1}) = \widehat{T}^{n_i+1}_{KB_i}(M_{i-1}), i \geq 1.$$

Then, KB is consistent iff $M_k \neq HB_P$, and in this case, $M_k = M_{KB}$.

Proof. Observe first that $M_0 = \widehat{T}^{n_0}_{KB_0}(\emptyset)$, where $n_0 \geq 0$ such that $\widehat{T}^{n_0}_{KB_0}(\emptyset) = \widehat{T}^{n_0+1}_{KB_0}(\emptyset)$. Since $\widehat{T}^i_{KB_0}(\emptyset) = T^j_{KB_0}(\emptyset)$ for all $j \geq 0$, it follows by Corollary 3.6.4 and Theorem 3.6.5 that (a) M_0 is the least model of KB_0 if KB_0 is satisfiable, and (b) $M_0 = HB_P$ if KB_0 is unsatisfiable. Observe then that for $i \geq 1$, it holds that $M_i = \widehat{T}^{n_i}_{KB_i}(M_{i-1})$, where $n_i \geq 0$ such that $\widehat{T}^{n_i}_{KB_i}(M_{i-1}) = \widehat{T}^{n_i+1}_{KB_i}(M_{i-1})$. Let $KB_i = (L_i, P_i)$, and let $KB_i{}' = (L_i, P_i')$, where P_i' is the strong dl-transform of P_i relative to L_i and M_{i-1}. Then, $\widehat{T}^j_{KB_i}(M_{i-1}) = T^j_{KB_i{}'}(\emptyset) \cup M_{i-1}$ for all $j \geq 0$. Hence, by Corollary 3.6.4 and Theorem 3.6.5, (a) $M_i = M_{KB_i{}'} \cup M_{i-1}$ if $KB_i{}'$ is satisfiable, and (b) $M_i = HB_P$ if $KB_i{}'$ is unsatisfiable. Equivalently, (a) M_i is the least model of KB_i with $M_i | HB^\star_{P_{i-1}} = M_{i-1} | HB^\star_{P_{i-1}}$ if such a model exists, and (b) $M_i = HB_P$ if no such model exists. In summary, $M_k \neq HB_P$ iff $M_i \neq HB_P$ for all $i \in \{0, \ldots, k\}$ iff KB is consistent. Furthermore, in this case, $M_k = M_{KB}$. □

Example 3.6.2 Assume that also rule r_3: $q(x) \leftarrow not\ \neg b, not\ DL[S](x)$ is in P of Example 3.6.1. Then, the λ assigning 1 to $q(a)$, 0 to $DL[S](a)$, and 0 to all other atoms in $HB_P \cup DL_P$ stratifies KB, and $M_0 = lfp(T_{KB_0}) = \{p(a), b\}$ and $M_1 = \{p(a), b, q(a)\} = M_{KB}$. ◇

3.6.3 Computing the Well-Founded Model

As we have shown in Subsection 3.3.1, for any positive dl-program $KB = (L, P)$, its least model M_{KB} is the least fixpoint of $T_{KB}(I)$. Thus, $\gamma_{KB}(I) = M_{KB^I}$ can be computed as $(KB^I = (L, sP_L^I))$

$$lfp(T_{KB^I}) = \bigcup_{i \geq 0} T^i_{KB^I}(\emptyset) \quad (= \bigcup_{i=0}^{|HB_P|} T^i_{KB^I}(\emptyset)).$$

The least and greatest fixpoint of γ^2_{KB} can be constructed as the limits

$$\begin{array}{rcl} U_\infty & = & \bigcup_{i \geq 0} U_i, \text{ where } U_0 = \emptyset, \text{ and } U_{i+1} = \gamma^2_{KB}(U_i), \\ O_\infty & = & \bigcap_{i \geq 0} O_i, \text{where } O_0 = HB_P \text{ and } O_{i+1} = \gamma^2_{KB}(O_i), \end{array} \quad i \geq 0,$$

respectively, which are both reached within $|HB_P|$ many steps.

3.7 Complexity

In this subsection, we draw a precise picture of the complexity of deciding strong and weak answer set existence for a dl-program, of brave and cautious reasoning from the strong and weak answer sets of a dl-program, respectively, and of well-founded reasoning. We consider the following canonical reasoning problems:

- Deciding whether a given dl-program KB has a strong (resp., weak) answer set.
- Deciding whether a given literal $l \in HB_P$ is in every strong (resp. weak) answer set of a given dl-program KB (Cautious Reasoning).
- Deciding whether a given literal $l \in HB_P$ is in some strong (resp. weak) answer set of a given dl-program KB (Brave Reasoning).
- Deciding whether a given literal $l \in HB_P$ is in the well-founded model a given dl-program.

We recall that deciding whether a given (non-ground) normal logic program has an answer set is complete for NEXP (nondeterministic exponential time) [Dantsin et al., 2001]. Furthermore, deciding satisfiability of a knowledge base L in $\mathcal{SHIF}(\mathbf{D})$ (resp. $\mathcal{SHOIN}(\mathbf{D})$) is complete for EXP (exponential time) [Tobies, 2001, Horrocks and Patel-Schneider, 2003] (resp., NEXP, assuming unary number encoding; see [Horrocks and Patel-Schneider, 2003] and the NEXP-hardness proof for $\mathcal{ALQI}$ in [Tobies, 2001], which implies the NEXP-hardness of $\mathcal{SHOIN}(\mathbf{D})$).

An easy consequence is that deciding, evaluating a given ground dl-atom a of form (3.2) in a given dl-program $KB = (L, P)$ and an interpretation I_p of its input predicates $p = p_1, \ldots, p_m$ (i.e., deciding whether $I \models_L a$ for each I which coincides on p with I_p) is EXP-complete for L from $\mathcal{SHIF}(\mathbf{D})$ resp. coNEXP-complete for L from $\mathcal{SHOIN}(\mathbf{D})$.

The proofs in the following subsections are considerably involved and therefore collected Appendix A for the sake of readability.

3.7.1 Deciding Answer Set Existence

We first consider the problem of deciding whether a given dl-program KB has a strong or weak answer set. Table 3.1 compactly summarizes our complexity results for this problem for L from $\mathcal{SHIF}(\mathbf{D})$ and $\mathcal{SHOIN}(\mathbf{D})$. The results are briefly explained as follows.

Observe first that for each dl-program KB, the number of ground dl-atoms a is polynomial, and every ground dl-atom a has in general exponentially many different concrete inputs I_p (that is, interpretations I_p of its input predicates $p = p_1, \ldots, p_m$), but each of these concrete inputs I_p has a polynomial size. Furthermore, notice that during the computation of the canonical model of a positive dl-program by fixpoint iteration, any ground dl-atom a needs to be evaluated only polynomially often, as its input can increase only that many times.

dl-program $KB = (L, P)$	L in $\mathcal{SHIF}(\mathbf{D})$	L in $\mathcal{SHOIN}(\mathbf{D})$
KB positive	EXP-complete	NEXP-complete
KB stratified	EXP-complete	$\mathsf{P}^{\mathsf{NEXP}}$-complete
KB general	NEXP-complete	$\mathsf{P}^{\mathsf{NEXP}}$-complete

Table 3.1: Complexity of deciding strong / weak answer set existence for dl-programs.

For L in $\mathcal{SHIF}(\mathbf{D})$, the evaluation of any ground dl-atom is, as mentioned above, feasible in EXP. Thus, the least fixpoint $lfp(T_{KB})$ for a positive KB is computable in exponential time; notice that any ground dl-atom a needs to be evaluated only polynomially often, as its input can increase only that many times. From $lfp(T_{KB})$ it is immediate whether KB has an answer set resp. a weak answer set, viz. iff $lfp(T_{KB}) \neq HB_P$. For other KB, we can, one by one, explore the exponentially many possible inputs of those dl-atoms which disappear in the reduction sP_L^I resp. wP_L^I. For each input, evaluating these dl-atoms and building sP_L^I resp. wP_L^I is feasible in exponential time. If we are left with a positive resp. stratified dl-program KB', we need just to compute M_{KB}, which we can do by (a sequence of) fixpoint iterations, and check compliance with the input of the dl-atoms. For unstratified KB, we need in addition an (exponential size) guess for the value of the default negated classical literals, which brings us to NEXP. The EXP- resp. NEXP-hardness lower bound for positive resp. general KB is inherited from the complexity of datalog and of deciding the existence of an answer set for a normal logic program, respectively [Dantsin et al., 2001].

The following theorem shows that deciding the existence of strong or weak answer sets of dl-programs $KB = (L, P)$ with L in $\mathcal{SHIF}(\mathbf{D})$ is complete for EXP in the positive and the stratified case, and complete for NEXP in the general case.

Theorem 3.7.1 *Given Φ and a dl-program $KB = (L, P)$ with L in $\mathcal{SHIF}(\mathbf{D})$, deciding whether KB has a strong or weak answer set is* EXP*-complete when KB is positive or stratified, and* NEXP*-complete when KB is a general dl-program.*

Proof. See Appendix A.

For L in $\mathcal{SHOIN}(\mathbf{D})$, we make use of the following observation: A positive KB has a strong resp. weak answer set, just if there exists an interpretation I and a subset $S \subseteq \{a \in DL_P \mid I \not\models_L a\}$, such that the positive logic program $P_{I,S}$ obtained from $ground(P)$ by deleting each rule which contains a dl-atom $a \in S$, and all remaining dl-atoms, has an answer set included in I. A suitable I and S, along with "proofs" $L \not\models_I a$ for all $a \in S$ (where each proof is a certificate of size bounded by an exponential), can be guessed and verified in exponential time. Hence, deciding the existence of a a strong resp. weak answer set (tantamount to consistency of KB) is in NEXP. The matching NEXP-hardness follows from coNEXP-hardness of dl-atom evaluation.

For non-positive KB, we can guess inputs I_p for all dl-atoms, and evaluate them with a NEXP oracle in polynomial time. For the (monotonic) ones remaining in sP_L^I, we can further guess a chain $\emptyset = I_p^0 \subset I_p^1 \subset \cdots \subset I_p^k = I_p$ along which their inputs are increased in a fixpoint computation for sP_L^I, and evaluate the dl-atoms on it in polynomial time with a NEXP oracle. We then ask a NEXP oracle if an interpretation I exists which is the answer set of sP_L^I resp. wP_L^I compliant with the inputs and valuations of the dl-atoms and as well as their input increase in fixpoint computation. This yields the $\mathsf{NP}^{\mathsf{NEXP}} = \mathsf{P}^{\mathsf{NEXP}}$ upper bounds. For a strong answer set of a stratified KB with only monotonic dl-atoms guesses can be avoided by increasing the monotonic dl-atoms along a stratification, and the problem is in $\mathsf{P}^{\mathsf{NEXP}}$.

We can obtain matching lower bounds by a generic reduction from Turing machines, by exploiting an NEXP-hardness proof for $\mathcal{ACLQI}$ by Tobies [2001]. The idea is to use a dl-atom to decide the result of the i-th oracle call made by a polynomial-time bounded Turing machine M with access to a NEXP oracle, where the results of the previous calls are known and input to the dl-atom. By a proper sequence of dl-atom evaluations, the result of M's computation on input w can be obtained; a nondeterministic M is modeled by further providing nondeterministically generated bits (either by unstratified rules or dl-atoms).

The next theorem shows that deciding the existence of strong or weak answer sets of dl-programs $KB = (L, P)$ with L in $\mathcal{SHOIN}(\mathbf{D})$ is complete for NEXP in the positive case, and complete for $\mathsf{P}^{\mathsf{NEXP}}$ in the stratified and the general case.

Theorem 3.7.2 *Given Φ and a dl-program $KB = (L, P)$ with L in $\mathcal{SHOIN}(\mathbf{D})$, deciding whether KB has a strong or weak answer set is* NEXP*-complete when KB is positive, and* $\mathsf{P}^{\mathsf{NEXP}}$*-complete when KB is a stratified or general dl-program.*

Proof. See Appendix A.

3.7.2 Brave and Cautious Reasoning

The canonical tasks of cautious and brave reasoning, whether a classical literal l belongs to every (resp., some) strong answer set or weak answer set of KB, are as usually easily reduced to the complement of answer set existence and answer set existence itself, respectively, by adding two rules $p \leftarrow l$ and $\neg p \leftarrow l$ in P (resp., $p \leftarrow not\, l$ and $\neg p \leftarrow not\, l$ in P), where p is a fresh propositional symbol.

Tables 3.2 and 3.3, respectively, compactly summarize our complexity results for these problems for L from $\mathcal{SHIF}(\mathbf{D})$ and $\mathcal{SHOIN}(\mathbf{D})$. Roughly, except for brave reasoning from positive dl-programs $KB = (L, P)$ with L from $\mathcal{SHOIN}(\mathbf{D})$, the complexity of cautious (resp., brave) reasoning from dl-programs coincides with the complexity of answer set non-existence (resp., existence) for dl-programs (see Table 3.1).

$KB = (L, P)$	L in $\mathcal{SHIF}(\mathbf{D})$	L in $\mathcal{SHOIN}(\mathbf{D})$
KB positive	EXP-complete	coNEXP-complete
KB stratified	EXP-complete	$\mathsf{P}^{\mathsf{NEXP}}$-complete
KB general	coNEXP-complete	$\mathsf{P}^{\mathsf{NEXP}}$-complete

Table 3.2: Complexity of cautious reasoning from the strong / weak answer sets of a dl-program.

Brave reasoning from the (unique) strong answer set of a positive dl-program is complete for $D^{exp} = \{L \times L' \mid L \in \mathsf{NEXP}, L' \in \mathsf{coNEXP}\}$, which is the "conjunction" of NEXP and

$KB = (L, P)$	L in $\mathcal{SHIF}(\mathbf{D})$	L in $\mathcal{SHOIN}(\mathbf{D})$
KB positive	EXP-complete	D^{exp}-complete / $\mathsf{P}^{\mathsf{NEXP}}$-complete
KB stratified	EXP-complete	$\mathsf{P}^{\mathsf{NEXP}}$-complete
KB general	NEXP-complete	$\mathsf{P}^{\mathsf{NEXP}}$-complete

Table 3.3: Complexity of brave reasoning from the strong / weak answer sets of a dl-program.

coNEXP. Intuitively, we have to decide consistency of KB, which is in NEXP, and a literal l might be included in the answer set on behalf of a dl-atom, which is in coNEXP.

Brave reasoning from the weak answer sets of a positive dl-program is harder, however, and in fact $\mathsf{NP}^{\mathsf{NEXP}}$-complete. Intuitively, an exponential number of candidate weak answer sets containing the query literal l might exist which are larger than the (unique) answer set of KB. This is a source of complexity which requires another guess.

Theorem 3.7.3 *Given Φ, a dl-program $KB = (L, P)$ with L in $\mathcal{SHIF}(\mathbf{D})$, and a classical literal $l \in HB_P$, deciding whether l belongs to every (resp. some) strong or weak answer set of KB is complete for* EXP *when KB is positive or stratified, and complete for* coNEXP *(resp.* NEXP*) when KB is a general dl-program.*

Proof. See Appendix A.

The next theorem shows that deciding whether a classical literal $l \in HB_P$ belongs to every (resp., some) strong/weak answer set of a given dl-program $KB = (L, P)$ with L in $\mathcal{SHOIN}(\mathbf{D})$ is complete for coNEXP (resp., $D^{exp}/\mathsf{P}^{\mathsf{NEXP}}$) in the positive case, and complete for $\mathsf{P}^{\mathsf{NEXP}}$ in the stratified and the general case.

Theorem 3.7.4 *Given Φ, a dl-program $KB = (L, P)$ with L in $\mathcal{SHOIN}(\mathbf{D})$, and a classical literal $l \in HB_P$, deciding whether l belongs to every (resp. some) strong/weak answer set of KB is complete for* coNEXP *(resp. $D^{exp}/\mathsf{P}^{\mathsf{NEXP}}$) when KB is positive, and complete for $\mathsf{P}^{\mathsf{NEXP}}$ when KB is a stratified or general dl-program.*

Proof. See Appendix A.

3.7.3 Well-Founded Reasoning

We recall that for a given ordinary normal program, computing the well-founded model needs exponential time in general (measured in the program size [Dantsin et al., 2001]), and also reasoning from the well-founded model has exponential time complexity.

The following result implies that the complexity of the well-founded semantics for dl-programs over $\mathcal{SHIF}(\mathbf{D})$ does not increase over the one of ordinary logic programs.

Theorem 3.7.5 *Given Φ and a dl-program $KB{=}(L, P)$ with L in $\mathcal{SHIF}(\mathbf{D})$, computing $WFS(KB)$ is feasible in exponential time. Furthermore, deciding whether for a given literal l it holds that $l \in WFS(KB)$ is* EXP*-complete.*

Proof. We show that, given $KB = (L, P)$ and $I \subseteq HB_P$, computing $\gamma_{KB}(I)$ is feasible in exponential time. The reduct $KB^I = (L, sP_L^I)$ is constructible in exponential time, since (i) $ground(P)$ is computable in exponential time and (ii) $I \models_L a$ for each dl-atom a in $ground(P)$ can be decided in exponential time, by the complexity of $\mathcal{SHIF}(\mathbf{D})$. Furthermore, computing the least model of KB^I is feasible in exponential time by computing $lfp(T_{KB^I}) = \bigcup_{i=0}^{|HB_P|} T_{KB^I}^i(\emptyset)$: this requires at most exponentially many applications

of $T_{KB^I}(J)$, each of which is computable in exponential time (deciding $I \models_L a$ for any dl-atom $a \in DL_P$ is feasible in exponential time).

Therefore, we can compute $lfp(\gamma^2_{KB}) = U_\infty$, by computing U_0, U_1, ... until $U_i = \gamma^{2i}_{KB}(\emptyset) = \gamma^{2i+2}_{KB}(\emptyset) = U_{i+1}$ holds for some i. Since i is bounded by $|HB_P|$ and the latter is exponential in the size of Φ and KB, the positive part of $WFS(KB)$, i.e., $lfp(\gamma^2_{KB})$, is computable in exponential time. The negative part of $WFS(KB)$ is easily obtained from $gfp(\gamma^2_{KB}) = O_\infty$, which can be similarly computed in exponential time. Therefore, computing $WFS(KB)$ is feasible in exponential time.

Consequently, deciding $l \in WFS(KB)$ is in EXP. The EXP-hardness is immediate from the EXP-hardness of deciding whether a given positive datalog program logically implies a given ground atom [Dantsin et al., 2001] (as well as from the EXP-hardness of $\mathcal{SHIF}(\mathbf{D})$ [Tobies, 2001]). □

For dl-programs over $\mathcal{SHOIN}(\mathbf{D})$, the computation of $WFS(KB)$ and reasoning from it is expected to be more complex than for $\mathcal{SHIF}(\mathbf{D})$ knowledge bases, since already evaluating a single dl-atom is coNEXP-hard. Computing WFS can be done, in a similar manner as described in the proof sketch of Theorem 3.7.5, in exponential time using an oracle for evaluating dl-atoms; to this end, an NP oracle is sufficient. As for the reasoning problem, this means that deciding $l \in WFS(KB)$ is in $\mathsf{EXP}^{\mathsf{NP}}$.

A more precise account reveals the following strict characterization of the complexity, which is believed to be lower.

Theorem 3.7.6 *Given Φ, a dl-program $KB = (L, P)$ with L in $\mathcal{SHOIN}(\mathbf{D})$, and a literal l, deciding $l \in WFS(KB)$ is $\mathsf{P}^{\mathsf{NEXP}}$-complete.*

Proof (sketch). For establishing membership in $\mathsf{P}^{\mathsf{NEXP}}$, an algorithm is not allowed to use exponential work space (only polynomial one). Thus, differently from the situation above, we cannot simply compute the powers $\gamma^j_{KB}(\emptyset)$, because $ground(P)$ is exponential. The idea is to move this problem inside an oracle call. The $\mathsf{P}^{\mathsf{NEXP}}$-hardness is easily derived from Theorem 3.5.5 and the result that deciding if a stratified KB in which strong negation $\neg$ may occur has some strong answer set is $\mathsf{P}^{\mathsf{NEXP}}$-complete (see Theorem 3.7.2). □

The results in Theorems 3.7.5 and 3.7.6 also show that like for ordinary normal programs, inference under the well-founded semantics is computationally less complex than under the answer-set semantics, since cautious reasoning from the strong answer sets of a dl-programs using a $\mathcal{SHIF}(\mathbf{D})$ (resp., $\mathcal{SHOIN}(\mathbf{D})$) description logic knowledge base is complete for coNEXP (resp., co-$\mathsf{NP}^{\mathsf{NEXP}}$).

We leave an account of the data complexity of dl-programs $KB = (L, P)$ (i.e., L and the rules of P are fixed, while facts in P may vary) for an expanded paper. However, we note that whenever the evaluation of dl-atoms is polynomial (i.e., in description logic terminology, ABox reasoning is polynomial), then also the computation of the well-founded semantics for dl-programs is polynomial. Most recent results in [Hufstadt et al., 2004] suggest that for $\mathcal{SHIF}(\mathbf{D})$, the problem is solvable in polynomial time with an NP oracle (and, presumably, complete for that complexity).

3.8 Reasoning Applications

We now want to survey three concrete scenarios that demonstrate the usefulness of dl-programs. Particularly, we will take advantage of the nonmonotonic behaviour of dl-atoms and their distinct feature of allowing to extend the Description Logics knowledge base from within the logic program.

3.8.1 Closed-World Reasoning

Reiter's well-known closed-world assumption (CWA) [Reiter, 1978][2] is acknowledged as an important reasoning principle for inferring negative information from a logical knowledge base KB: For a ground atom $p(c)$, conclude $\neg p(c)$ if $KB \not\models p(c)$. Description Logics knowledge bases lack this notion of inference, adhering to the open-world assumption. Originally aimed at relational databases, the CWA is of increasing interest also for data representations in more expressive data models like Description Logics.

The discussion whether the Semantic Web needs open- or closed-world reasoning is a very lively one in Semantic Web research. The Description Logics community leans towards the open world approach, whereas many people with a logic programming background try to bring in their ideas of nonmonotonicity. Generally, we believe that this question depends very much on the domain of reasoning. One can assume that in a Semantic Web scenario, a single knowledge base is usually considered as part of a distributed pool of information, rather than an isolated traditional database containing complete information. Thus, new knowledge may easily contradict information that was inferred earlier by the CWA, leading to inconsistency. Nevertheless, many specific scenarios in the Semantic Web might profit from some form of closed-world reasoning [Analyti et al., 2005, Heflin and Muñoz-Avila, 2002, Polleres et al., 2006].

Using dl-programs, the CWA may be easily expressed on top of an external KB which can be queried through suitable dl-atoms. We show this here for a DL knowledge base L. Intuitively, given a concept C, its negated (under CWA) version $\overline{C}$ is defined by adding to a given dl-program the rule

$$\overline{C}(X) \leftarrow not\ DL[C](X).$$

For example, given that

$$L = \{man \sqsubseteq person,\ person(lee)\}$$

for concepts man and $person$, the CWA infers $\overline{man}(lee)$.

As well known, the CWA can lead to inconsistent conclusions. If in the above example, L contains a further axiom

$$person = man \sqcup woman,\ \bot = man \sqcap woman,$$

then the CWA infers $\overline{man}(lee)$ and $\overline{woman}(lee)$, which is inconsistent with L.

We can check inconsistency of the CWA with a further rule:

$$fail \leftarrow DL[woman \uplus \overline{woman}, man \uplus \overline{man}; \bot](b),$$

where $\bot$ is the empty concept (entailment of $\bot(b)$, for any constant b, is tantamount to inconsistency).

The problem with inconsistency in the CWA as presented above evidently stems from the strategy to negate *every* literal that cannot be proven to be true. Several approaches exist in literature to avoid such inconsistencies, e.g., by explicitly selecting the predicates (or concepts, as we call them in the context of this subsection) that can safely be negated [Cadoli and Lenzerini, 1994, Gelfond et al., 1986]. The *extended closed-world assumption*, for instance, introduces a partitioning of the theory's concepts into three sets P, Q, and Z. From a model-theoretic viewpoint, the ECWA constructs models which are minimal w.r.t. P having the same fixed interpretation of Q, while Z can vary freely.

Corresponding to Gelfond et al., we include the following assumptions:

[2] Throughout this section, we refer to Łukaszewicz [1990] for references to closed-world reasoning and circumscription.

- The domain closure assumption (DCA), stating that there are no more individuals than those explicitly named in the ABox of L, i.e., the domain is finite and its cardinality is known.
- The unique name assumption (UNA), stating that distinct names also refer to distinct objects in the domain.

These assumptions are fulfilled by the semantics of dl-programs and can also be enforced in $\mathcal{SHOIN}(\mathbf{D})$.

Definition 3.8.1 *Given a knowledge base L, and a partition $\langle P, Q, Z\rangle$ of its predicates (concept and roles, respectively), a model of L is $\langle P, Q, Z\rangle$-minimal if the extension of positive concept and roles in P w.r.t. individuals in the ABox is minimal (w.r.t. subset inclusion) among those models of L that have the same extension for concept/roles in Q, regardless of the extension of predicates in Z. We say that $L \models_{\langle P,Q,Z\rangle} a$ if either,*

- *a is a concept (or role) assertion belonging to all the $\langle P, Q, Z\rangle$-minimal models of L, or*
- *a is a negative assertion $\neg l$ (where l is a concept or role assertion belonging to predicates of P) and there is no $\langle P, Q, Z\rangle$-minimal model of L containing l.*

Applying the ECWA in the general case of minimizing *all* predicates in L, means simply to consider the partition $\langle P, \emptyset, \emptyset\rangle$. Otherwise, it might be desirable to minimize only w.r.t. a specific set of predicates in L.

Intuitively, building minimal models of L corresponds to concluding as much negative facts as possible while keeping consistency. We can create the $\langle P, \emptyset, \emptyset\rangle$-minimal models of L with respect to all concepts and individuals in L elegantly with the following dl-program $KB' = (P', L)$. Given a concept C in L, we associate to it two predicates $\overline{C}$ and C^+ in P': $\overline{C}$ represents the negated counterpart of C in P', while C^+ represent the positive value of P'.

$$
\begin{aligned}
\overline{man}(X) &\leftarrow not\ man^+(X);\\
\overline{woman}(X) &\leftarrow not\ woman^+(X);\\
\overline{person}(X) &\leftarrow not\ person^+(X);\\
man^+(X) &\leftarrow DL[woman \uplus \overline{woman}, man \uplus \overline{man}, person \uplus \overline{person}; man](X);\\
woman^+(X) &\leftarrow DL[woman \uplus \overline{woman}, man \uplus \overline{man}, person \uplus \overline{person}; woman](X);\\
person^+(X) &\leftarrow DL[woman \uplus \overline{woman}, man \uplus \overline{man}, person \uplus \overline{person}; person](X).
\end{aligned}
$$

The first three rules in this program add the negations of those atoms to the answer set that are not explicitly positive in the answer set. The next three rules add the negative part of the model to L and queries under this condition the extensions of the positive concepts.

Applied to our example, we obtain two strong answer sets

$$
\begin{aligned}
M_1 &= \{person^+(lee), woman^+(lee), \overline{man}(lee)\},\\
M_2 &= \{person^+(lee), man^+(lee), \overline{woman}(lee)\}.
\end{aligned}
$$

which correspond to the $\langle P, \emptyset, \emptyset\rangle$-minimal models of L. Roles in L may be handled similarly.

Furthermore, one can easily restrict minimization to a subset of concepts and roles, and accommodate the general setting of ECWA and circumscription, dividing the predicates into minimized, fixed, and floating predicates P, Q, and Z, respectively.

We give here a proof of equivalence valid for $\langle P, \emptyset, Z\rangle$ entailment.

Definition 3.8.2 *Let L be a knowledge base, and $\langle P, \emptyset, Z\rangle$ a partition of its concepts and roles. Let $P = h_1, \ldots, h_n$. The dl-program $KB^{\text{ECWA}} = (Pr^{L,P,Z}, L)$ is built by constructing $Pr^{L,P,Z}$ as follows:*

1. *For each concept $c \in P$, add the rules*

$$\overline{c}(X) \leftarrow not\, c^+(X); \tag{3.6}$$
$$c^+(X) \leftarrow DL[h_1 \uplus \overline{h}_1, \ldots, h_n \uplus \overline{h}_n; c](X); \tag{3.7}$$

2. *For each role $r \in P$, add the rules*

$$\overline{r}(X,Y) \leftarrow not\, r^+(X,Y); \tag{3.8}$$
$$r^+(X,Y) \leftarrow DL[h_1 \uplus \overline{h}_1, \ldots, h_n \uplus \overline{h}_n; r](X,Y); \tag{3.9}$$

3. *add the rule*

$$fail \leftarrow DL[h_1 \uplus \overline{h}_1, \ldots, h_n \uplus \overline{h}_n; \bot](b), not\, fail; \tag{3.10}$$

where b is a dummy constant symbol.

Theorem 3.8.1 *Let L be a knowledge base, $\langle P, Z\rangle$ a partition of its concepts and roles, and $\neg c(a)$ (resp. $\neg r(a,b)$) a concept (resp. role) assertion. Then $L \models_{\langle P,\emptyset,Z\rangle} \neg c(a)$ (resp. $\neg r(a,b)$) iff KB^{ECWA} cautiously entails $\overline{c}(a)$ (resp. $\overline{r}(a,b)$).*

Proof. We consider a specific concept assertion $c(a)$. The proof for role assertions is analogous.

($\Leftarrow$) Assume by contradiction that $L \not\models_{\langle P,\emptyset,Z\rangle} \neg c(a)$. This means that there is at least one $\langle P, \emptyset, Z\rangle$-minimal model m of L containing $c(a)$. From m we can build a strong answer set M of $Pr^{L,P,Z}$ such that for each concept (resp. role) assertion $c(x) \in m$ (resp. $r(x,y) \in m$) we have that

- $\overline{c}(x) \in M$ if $c(x) \notin m$ (resp. $\overline{r}(x,y) \in M$ if $r(x,y) \notin m$);
- $c^+(x) \in M$ (resp. $c^+(x) \in M$) otherwise.

Note that M contains $c^+(a)$. Because of the presence of rule 3.6 and 3.7, $\overline{c}(a)$ can not belong to M, and thus it is not cautiously entailed by KB^*, contradicting the hypothesis.

($\Rightarrow$) Assume by contradiction that KB^* does not cautiously entail $\overline{c}(a)$. Then there must exist a strong answer set M of $Pr^{L,P,Z}$ which does not contain $\overline{c}(a)$. This means that $c^+(a) \in M$ instead, because of the presence of rule 3.6. Note that $fail \notin M$ because of rule 3.10. This implies that the knowledge base $L \cup \bigcup_{i=1}^{n} A_i(M)$ built according to the dl-atom occurring in 3.10 and to semantics given in Section 3.4, is consistent and entails $c(a)$.

Note that $c(a)$ is part of a $\langle P, \emptyset, Z\rangle$-minimal model m of L, and thus $L \not\models_{\langle P,\emptyset,Z\rangle} \neg c(a)$. Indeed, if this would not be the case, $\overline{c}(a)$ could be put in M, so that $c(a)$ could be asserted negated in L, still producing a consistent knowledge base. That is, a model smaller than m (without $c(a)$) with respect predicates in P, would be possible. □

3.8.2 Default Reasoning

As we already mentioned,Description Logic knowledge bases genuinely do not support nonmonotonic inheritance. However, overriding a "default" property value of a concept may be a natural way of defining a subclass. Defaults are especially suitable for implementing nonmonotonic inheritance.

For example, the rules

$$white(w) \leftarrow DL[sparklingwine](w), not\ nonwhite(w);$$
$$nonwhite(w) \leftarrow DL[whitewine \uplus white; \neg whitewine](w).$$

on top of a wine ontology L, express that sparkling wines are white by default. Given

$$L = \{sparklingWine(veuveCliquot), (sparklingWine \sqcap \neg whiteWine)(lambrusco)\},$$

we then can conclude $white(veuveCliquot)$ and $nonwhite(lambrusco)$.

Here, we are aiming at deriving as much positive information without causing inconsistencies, i.e., *maximizing* extensions instead of *minimizing* them as proposed in the previous subsection. Integrating default reasoning with Description Logics is in general a nontrivial task [Baader and Hollunder, 1995]. Our dl-programs can serve as a convenient framework to realize different notions of *default reasoning* as in the approaches by Reiter [1980] or Poole [1988].

Poole's approach is to view default reasoning as theory formation instead of the definition of a new logic, such as the one proposed by Reiter. He categorizes a theory into a set of closed formulas $\mathcal{F}$ and a set of (possibly open) formulae Δ, called *possible hypotheses*. The formulas in $\mathcal{F}$ are treated as "facts" and considered to be consistent. These statements have to be true in any case, whereas any ground instance of Δ *can* be used if it is consistent. Intuitively, we can compare a formula $g \in \Delta$ to the Reiter default $\frac{:Mg}{g}$. If we also want to take prerequisites into account, we can in a first step formulate normal defaults of the form $\frac{a:Mw}{w}$, where both a and w are concept (resp. role) names, expressing non-ground membership axioms. We abbreviate this form by $a : w$. Such a default should maximize the extension of w, whenever a is true and consistency is preserved.

We can model this behaviour in the following way, assuming to have a set of defaults $a_i : w_i$ with $0 \leq i \leq n$:

$$w_i^+(X) \leftarrow DL[a_i](X), not\ \overline{w}_i(X); \tag{3.11}$$
$$\overline{w}_i(X) \leftarrow DL[a_i](X), DL[\lambda; \neg w_i](X). \tag{3.12}$$

where $\lambda = w_1 \uplus w_1^+, \ldots, w_n \uplus w_n^+$, i.e., the update of the extension candidate. If this update causes an inconsistency in the DL knowledge base, $\overline{w}_i(X)$ will be true and hence the default rule cannot be applied.

A query for a specific concept c can then simply be expressed as follows:

$$query(X) \leftarrow DL[\lambda; c](X). \tag{3.13}$$

To explicitly build an extension at the level of ground literals for each concept c (whether or not c is a consequence of a default) instead of querying, we use this rule:

$$c^+(X) \leftarrow DL[\lambda; c](X). \tag{3.14}$$

Example 3.8.1 In a simple circuit diagnosis defaults are used to state that components are assumed to be working. Like in classical model-based diagnosis, we want to maximize the set of working components. A Reiter Diagnosis of an observed system $(SD, COMP, OBS)$ is a minimal set $\Delta \subseteq COMP$ such that $SD \cup OBS \cup \{\neg ok(c) \mid c \in \Delta\} \cup \{ok(c) \mid c \in COMP \setminus \Delta\}$ is satisfied, where SD is the system description, $COMP$ the set of components, and OBS the set of observations; the extension of ok denotes working components (cf. [Reiter, 1987]).

Assume that a Description Logics knowledge base contains the concepts *Component* and *Working*. The default $Component : Working$ can be expressed by

$$ok(X) \leftarrow DL[Component](X), not\ broken(X); \tag{3.15}$$
$$broken(X) \leftarrow DL[Component](X), DL[Working \uplus ok; \neg Working](X). \tag{3.16}$$

This encoding yields all extensions (i.e., all maximal consistent diagnoses) as answer sets. ◇

In order to enable feedback from default conclusions to prerequisites (i.e., "chaining" of defaults), we have to propagate them also to the positive queries, modifying rule 3.11 in the following way:

$$w_i^+(X) \leftarrow DL[\lambda; a_i](X), not\ \overline{w}_i(X); \tag{3.17}$$

Following this approach, we can in fact capture Reiter's default logic, which defines the extensions of a default theory $T = \langle W, D\rangle$ in terms of the the fixpoints of a closure operator $\Gamma_T(S)$ as follows:

1. $W \subseteq \Gamma_T(S)$;
2. $Cn(\Gamma_T(S)) = \Gamma_T(S)$;
3. if $\frac{\alpha : M\beta_1, \ldots, M\beta_n}{\gamma} \in D$ and $\alpha \in \Gamma_T(S)$ and $\neg\beta_1, \ldots, \neg\beta_n \notin S$ then $\gamma \in \Gamma_T(S)$.

Then, E is an extension of T iff $\Gamma_T(E) = E$.

We can model this procedure by the following guess-and-check method (the generally open defaults are closed through grounding):

Definition 3.8.3 *Let L be a DL knowledge base and D a set of defaults $a_1 : w_1, \ldots, a_n : w_n$, where each a_i and w_i are concept expressions.*[3] *The rules P^D of the dl-program $KB^{Def} = (P^D, L)$ are then constructed as follows:*

1. *For each w_i, we add the guessing rules*

$$in\_w_i(X) \leftarrow not\ out\_w_i(X), dom(X); \tag{3.18}$$
$$out\_w_i(X) \leftarrow not\ in\_w_i(X), dom(X); \tag{3.19}$$

   using a domain predicate dom that needs to contain all individuals in the ABox of L.

2. *Then, we check the compliance of the guess with L for each default:*

$$false \leftarrow DL[\lambda'; w_i](X), out\_w_i(X), not\ false; \tag{3.20}$$

   where $\lambda' = w_1 \uplus in\_w_1, \ldots, w_n \uplus in\_w_n$.

3. *The default is applied by the following rule, summarizing Rules 3.12 and 3.17, but updating w.r.t. the current guess:*

$$w_i^+(X) \leftarrow DL[\lambda; a_i](X), not\ DL[\lambda'; \neg w_i](X); \tag{3.21}$$

   where λ is defined as before.

[3] It is easy to expand this to roles: each variable X has to be replaced by the tuple X, Y, modifying also the safety guard to $dom(X), dom(Y)$.

4. Eventually, we need to verify whether the guessed extension can be reconstructed:

$$false \leftarrow not\ DL[\lambda; w_i](X), in\_w_i(X), not\ false; \tag{3.22}$$
$$false \leftarrow DL[\lambda; w_i](X), out\_w_i(X), not\ false; \tag{3.23}$$

The predicate *dom* in Rules 3.18 and 3.19 of Definition 3.8.3 ensures DL-safety, i.e., the grounding over known individuals from L. It can be shown that for each extension E of $\langle L, D\rangle$, there exists an answer set S of $KB^{Def} = (P^D, L)$ such that $S = \{in\_w_i(c) \mid w_i(c) \in E\} \cup \{w_i^+(c) \mid w_i(c) \in E\} \cup \{out\_w_i(c) \mid w_i(c) \notin E\}$ and vice versa.

Example 3.8.2 Let L be the following DL knowledge base about the prototypical penguin example. Penguins are birds that cannot fly:

$Penguin \sqsubseteq Bird \sqcap \neg Flies$;
$Penguin(tweety)$; $Bird(joe)$.

Let D contain a single default, assuming that birds normally fly: $D = \{Bird(X) : Flies(X)\}$. The rules P^D of $KB^{Def} = (P^D, L)$ are as follows:

$$\begin{aligned}
in_Flies(X) \leftarrow & not\ out_Flies(X), dom(X);\\
out_Flies(X) \leftarrow & not\ in_Flies(X), dom(X);\\
false \leftarrow & DL[Flies \uplus in_Flies; Flies](X), out_Flies(X), not\ false;\\
flies^+(X) \leftarrow & DL[Flies \uplus flies^+; Bird](X),\\
& not\ DL[Flies \uplus in_Flies; \neg Flies](X);\\
false \leftarrow & not\ DL[Flies \uplus flies^+; Flies_i](X), in_Flies(X), not\ false;\\
false \leftarrow & DL[Flies \uplus flies^+; Flies](X), out_Flies(X), not\ false;
\end{aligned}$$

We evaluate P^D w.r.t. the domain extension $\{dom(tweety), dom(joe)\}$.

The guessing rules (3.18) and (3.19) create four answer sets (modulo the facts of the predicate *dom*):

$\{in\_Flies(tweety),\ in\_Flies(joe)\}$;
$\{in\_Flies(tweety),\ out\_Flies(joe)\}$;
$\{out\_Flies(tweety),\ in\_Flies(joe)\}$;
$\{out\_Flies(tweety),\ out\_Flies(joe)\}$.

Adding Rule 3.20 checks whether we guessed a non-flier which flies according to L and removes such models. Here the second model does not survive. Note that for the first two guesses the DL KB is inconsistent due to the addition of *tweety* to *Flies*, hence the dl-atom is always satisfied.

Rule 3.21 applies the single default in D, checking whether the prerequisite is satisfied and the justification can be consistently assumed. At this point, we still have three answer sets:

$\{in\_Flies(tweety),\ in\_Flies(joe)\}$;
$\{out\_Flies(tweety),\ in\_Flies(joe),\ flies^+(joe)\}$;
$\{out\_Flies(tweety),\ out\_Flies(joe),\ flies^+(joe)\}$.

The default can be applied to *joe*, except in the inconsistent case of the first guess.

The reconstruction of the guessed extension is verified with the last two rules. Here, we feed $flies^+$ into the L and check whether the default's consequences are compliant with the guess. This results in the following single answer set:

$$\{out\_Flies(tweety),\ in\_Flies(joe),\ flies^+(joe)\}.$$

As expected, *joe* flies, because he is a bird, as opposed to *tweety*, for whom we cannot consistently assume to be flying. ◇

In general, we can straightforwardly cover more general defaults $\frac{\alpha:\beta}{\omega}$, where $\alpha = a_1 \wedge \ldots \wedge a_n$, $\omega = w_1 \wedge \ldots \wedge w_m$ and $\beta = Mb_1, \ldots, Mb_l$, with a_i, w_i, and b_i being concept or role names.

3.8.3 DL-Safe Rules

DL-safe rules [Motik et al., 2005] represent one of the first attempts to couple rules with ontologies while keeping a full first-order semantics together with decidability (see Section 3.10 for details). In order to ensure decidability, only a limited form of rules is allowed.

Intuitively, a *DL-safe* program is a description logic knowledge base L coupled with a set of Horn rules P.[4] Concept and roles from L may freely appear in P (also in rule heads). Nonetheless, any variable must appear in the body of a rule within an atom whose predicate name does not appear in L.

Definition 3.8.4 *Assume we have a $\mathcal{SHOIN}(\mathbf{D})$ knowledge base L and a set of concept names $\mathbf{A}$ and of abstract and datatype role names $\mathbf{R}_A$ and $\mathbf{R}_D$. Let $L_c \subseteq \mathbf{A}$ be the set of concept names appearing in L while $L_r \subseteq \mathbf{R}_A \cup \mathbf{R}_D$ is the set of role names appearing in L. Let P a set of Horn rules, allowing atoms with predicate names from a set $N \subseteq \mathbf{A} \cup \mathbf{R}_A \cup \mathbf{R}_D$.[5] A rule $r \in P$ is* DL-safe *if each variable in r occurs also in an atom with a predicate name $n \in N \setminus L_c \cup L_r$. A program P is* DL-safe *if all its rules are DL-safe.*

A (first order) interpretation I satisfies a DL-safe program (L, P) ($I \models L \cup P$) if $I \models L$ and $I \models P$. Given a ground atom α, we say that $(L, P) \models \alpha$ if every interpretation I satisfying (L, P) also satisfies α.

We can to simulate DL-safe programs using dl-programs in the following way:

Definition 3.8.5 *Given a DL-safe program (L, P), we build a dl-program $KB = (L, P')$ as follows: for each predicate p appearing in P, we introduce the predicates p^+ and p^- and add the following rule to P':*

$$p^+(\bar{X}) \leftarrow not\ p^-(\bar{X}); \tag{3.24}$$
$$p^-(\bar{X}) \leftarrow not\ p^+(\bar{X}). \tag{3.25}$$

For each rule $r = h(\bar{X}) \leftarrow b(\bar{X})$ in P we add the following constraint to P':

$$fail \leftarrow not\ h(\bar{X}), b(\bar{X}), not\ fail. \tag{3.26}$$

[4] Originally, Motik et al. denote a DL-safe program with (KB, P); to be consistent with the notation of dl-programs, in this subsection we replace their KB by L.

[5] Predicate names from A appear only in unary atoms whereas predicate names from $\mathbf{R}_A \cup \mathbf{R}_D$ appear only in binary atoms.

Finally, we add:

$$fail \leftarrow DL[\lambda; \bot](b), not\ fail. \tag{3.27}$$

where b is a dummy constant symbol and $\lambda = p_1 \uplus p_1^+, \ldots, p_n \uplus p_n^+, p_1 \uplus p_1^-, \ldots, p_n \uplus p_n^-$ for all predicates in P. This rule simple implements a consistency check, discarding the respective model if the guess is not compliant with L.

Intuitively, Rules 3.24 and 3.25 guess the extension of each predicate in P. For querying a ground atom $\alpha(\mathbf{x})$, we have two possibilities:

1. If α is a predicate from P, we add to P':

$$\hat{\alpha}(\mathbf{x}) \leftarrow \alpha(\mathbf{x}). \tag{3.28}$$

2. If α is a predicate from $L \setminus P$, we add the following rule to P':

$$\hat{\alpha}(\mathbf{x}) \leftarrow DL[\lambda; \alpha](\mathbf{x}). \tag{3.29}$$

Querying for $\alpha(\mathbf{x})$ then simply amounts to checking if $\hat{\alpha}(\mathbf{x})$ is cautiously entailed in all answer sets of KB.

The following lemma shows that grounding of P' over constants in $L \cup P$ is sufficient ensuring soundness and correctness of this translation.

Lemma 3.8.2 *Let (L, P) be a DL-safe program. Let $P\downarrow$ be the instantiation of P over the constant names in $P \cup L$ and α be any ground atom from $L \cup P$. Then, for each interpretation $\mathcal{I}$ such that $\mathcal{I} \models L \cup P\downarrow$ there is an interpretation $\mathcal{J}$ such that $\mathcal{J} \models L \cup P$ and $\mathcal{I} \models \alpha$ iff $\mathcal{J} \models \alpha$ Also, for each $\mathcal{J}$ such that $\mathcal{J} \models L \cup P$ there exists an $\mathcal{I}$ such that $\mathcal{I} \models L \cup P\downarrow$ and $\mathcal{J} \models \alpha$ iff $\mathcal{I} \models \alpha$. Moreover, $\mathcal{I}$ is such that any individual a belonging to $p^{\mathcal{I}}$ where p is a concept or role in P is mapped to a constant symbol appearing in $L \cup P$.*

Theorem 3.8.3 *Given a ground atom $\alpha(\mathbf{x})$ then $(L, P) \models \alpha(\mathbf{x})$ iff $KB = (L, P')$ cautiously entails $\hat{\alpha}(\mathbf{x})$ under strong answer-set semantics.*

Proof (sketch). ($\Rightarrow$) If we assume that $(L, P) \models \alpha(\mathbf{x})$ then any interpretation $\mathcal{J}$ modeling L and P is both such that $\mathcal{J} \models L \cup \alpha(\mathbf{x})$ and $\mathcal{J} \models P \cup \alpha(\mathbf{x})$. According to Lemma 3.8.2, there is a $\mathcal{I} \models L \cup P\downarrow$ that maps every individual belonging to a concept or role appearing in P to a constant symbol in $L \cup P$. From $\mathcal{I}$ it is possible to build an answer set A of KB, such that $A \models \hat{\alpha}(\mathbf{x})$. Moreover there is no answer set B such that $B \not\models \hat{\alpha}(\mathbf{x})$ because otherwise, there would be an interpretation $\mathcal{I}'$ such that either Rule (3.28) or Rule (3.29) is unsatisfied.

($\Leftarrow$) Given a strong answer set A of KB such that $A \models \hat{\alpha}(\mathbf{x})$, we can build an interpretation $\mathcal{I}$ such that $\mathcal{I} \models \alpha(\mathbf{x})$. Would there exist an interpretation $\mathcal{I}'$ such that $\mathcal{I}' \not\models \alpha(\mathbf{x})$, then either by Rule (3.28) or by Rule (3.29), there would exist an answer set A' such that $A' \not\models \hat{\alpha}(\mathbf{x})$ □

3.9 Implementing a Solver for dl-Programs

In the following, we will describe an implementation for a reasoner for dl-programs, called NLP-DL (Nonmonotonic Logic Programming with Description Logics), that can operate under different semantics. Our idea behind the implementation principle was to design a reasoning framework on top of existing reasoners for answer-set programs resp. Description

Logics instead of creating everything from scratch. The reasons for this decision were mainly constrained human resources but also the fact that these existing engines have been professionally developed and are supposedly highly efficient.

In Section 3.6 we presented a method for evaluating a general dl-program. It is evident that in practice the guessing part of this algorithm generates a great amount of preliminary models to be validated and therefore makes it very difficult to be implemented efficiently. However, when looking at the corresponding dependency graph, it often occurs in practice that answer-set programs are structured in two distinct and hierarchic layers, where a first, stratified layer at the bottom performs some preprocessing on the input data and a second, strongly connected and unstratified layer, usually is aimed at encoding some nondeterministic choice. It is therefore desirable to constrain the usage of the general algorithm only to the necessary parts of the program and evaluate the remaining part by a more efficient, fixpoint-based method. This implies that we have to split the program and evaluate each part separately — a feasible method, relying on the theorem of *splitting sets*, which was defined for the answer-set semantics by Lifschitz and Turner [1994]. For simplicity, our approach is to split the program only in two parts, having a fast routine for finding the answer set(s) of the lower layer, while the remaining subprogram will be solved by the guess-and-check method (if such a subprogram exists at all).

3.9.1 Splitting the Input Program

In answer-set mode, all stable models according to the semantics presented in Subsection 3.4.1, i.e., the strong answer sets, are generated. The method used in this prototype partly corresponds to the evaluation techniques presented in Section 3.6, where we presented a general algorithm for computing the weak answer sets following a guess-and-check approach, which we then supplemented by a fixpoint characterization for stratified normal programs. In practice it is evident, that the guessing part of the general algorithm generates a great amount of models to be checked

We have already mentioned that answer-set programs often are structured in three separate and hierarchic layers:

- a first, stratified layer at the bottom which performs some preprocessing on the input data;
- a second, unstratified layer, used for opening up some search space for a nondeterministic choice, and eventually,
- a third "checking" layer on top, where values computed through the other layers are filtered with respect to some constraint criteria.

Following this common setting, we conceived an evaluation strategy where we separate the first, stratified layer from the rest of the program and evaluate the two parts separately, feeding the result from the first into the computation of the second. The bottom layer can thus be solved by an iterative routine, while general techniques are limited to situations in which this cannot be avoided, as in non-stratified layers.

Lifschitz and Turner [1994] have shown that the computation of the answer sets of a logic program can be simplified by dividing the program into two parts. Informally, we first identify the unstratified subprograms of $KB = (L, P)$, i.e., rules of P that contain negated cycles. We then remove these rules from P as well as all rules that depend on P, leaving a stratified subprogram on the "bottom" of the dependency graph of P. The model of this part can now be solved by a fixpoint iteration, i.e., resulting in a unique least model.

Subsequently, this model is added as extensional knowledge to the remaining, unstratified part of P, which is eventually solved by means of a guess-and-check procedure.

Lifschitz and Turner define a *splitting set* as a set U of literals such that, for every rule $r \in P$, if $H(r) \cap U \neq \emptyset$ then $lit(r) \subseteq U$, where $lit(r)$ denotes $H(r) \cup B^+(r) \cup B^-(r)$. Since in dl-programs, not only the dependency between rule body and rule head, but also between dl-atoms and their input predicates needs to be taken into account here, we need to modify the definition of splitting sets. To this end, we first formalize the notion of *dependency*, which takes the occurrence of dl-atoms into account:

Definition 3.9.1 *Let $KB = (L, P)$ be a dl-program and a, b classical literals in some rule of P. Then, a* depends positively on *b ($a \rightarrow_p b$), if one of the following conditions holds:*

1. *There is some rule $r \in P$ such that $a \in H(r)$ and $b \in B^+(r)$.*
   Example: $r_1 : p(X) \leftarrow q(X), r(X)$.
   Clearly, we have the dependencies $p(X) \rightarrow q(X)$ and $p(X) \rightarrow r(X)$.

2. *There are some rules $r_1, r_2 \in P$ such that $a \in B(r_1)$ and $b \in H(r_2)$ and a and b can be unified.*
   Example: $r_1 : p(X) \leftarrow q(X), r(X)$.
   $r_2 : q(Y) \leftarrow s(Y)$.
   Here, we have $q(X) \rightarrow q(Y)$.

3. *a is a dl-literal (i.e., a possibly weakly negated dl-atom), b is an ordinary (i.e., not a dl-) literal and the predicate symbol of b occurs in the input list of a.*
   Example: $r_1 : p(X) \leftarrow DL[Student \uplus s; Person](X)$.
   $r_2 : s(X) \leftarrow enrolled(X)$.
   Here we have $DL[Student \uplus s; Person](X) \rightarrow s(X)$.

We say that a depends negatively on *b ($a \rightarrow_n b$), if one of the following conditions holds:*

1. *There is some rule $r \in P$ such that $a \in H(r)$ and $b \in B^-(r)$.*
   Example: $r_1 : flies(X) \leftarrow bird(X), not\ penguin(X)$.
   It follows that $flies(X) \rightarrow bird(X)$ and $flies(X) \rightarrow_n penguin(X)$.

2. *There is some rule $r \in P$ such that $a \in H(r)$, $b \in B(r)$ and b is a non-monotonic dl-atom.*
   Example: $r_1 : part(X) \leftarrow DL[P \uplus known; P](X)$.
   We have $part(X) \rightarrow_n DL[P \uplus known; P](X)$.

The relation $\rightarrow^+$ denotes the transitive closure of $\rightarrow$. We call a set N of literals a weakly connected component, *if for each $a \in N$ there exists a $b \in N$ such that $a \rightarrow b$. Moreover, N is a* strongly connected component, *if $a \rightarrow^+ b$ holds for all $a, b \in N$.*

It is important to note that if we speak about stratified and unstratified dl-programs, we are referring to this dependency relation, which takes not only weak negation into account to identify possible unstratified programs, but also nonmonotonic dl-atoms.

Definition 3.9.2 *A* splitting set for a dl-program *$KB = (L, P)$ is any set U of classical literals such that, for any $a \in U$, if $a \rightarrow b$, then $b \in U$. The set of rules $r \in P$ such that $H(r) \in U$ is called the* bottom of P relative to the splitting set U *and denoted by $b_U(P)$. Moreover, we denote with $ground(U)$ the set of all grounded literals in U w.r.t. the constants in U.*

To describe a method how to use this splitting for the computation of answer sets, we first need to define the notion of a *solution to KB with respect to U*, which corresponds directly to the respective notion of Lifschitz and Turner. We consider two sets of literals U, X and a dl-program $KB = (L, P)$. For each rule $r \in \mathit{ground}(P)$ such that $B^+(r) \cap \mathit{ground}(U) \subseteq X$ and $B^-(r) \cap \mathit{ground}(U)$ is disjoint from X, create a new rule r', with $H(r') = H(r)$, $B^+(r') = B^+(r) \setminus \mathit{ground}(U)$ and $B^-(r') = B^-(r) \setminus \mathit{ground}(U)$. The program consisting of all rules r' is denoted by $e_U(P, X)$.

Let U be a splitting set for a program $KB = (L, P)$. We call a pair $\langle X, Y\rangle$ of sets of literals a *solution* to KB w.r.t. U, if

- X is an answer set for $b_U(P)$,
- Y is an answer set for $e_U(P \setminus b_U(P), X)$,
- and $X \cup Y$ is consistent.

Theorem 3.9.1 *Let U be a splitting set for a dl-program $KB = (L, P)$. A set A of literals is a consistent answer set of KB iff $A = X \cup Y$ for some solution $\langle X, Y\rangle$ to KB with respect to U.*

Proof. We can reformulate this theorem as follows: Let U be a splitting set for a dl-program $KB = (L, P)$. A set A of literals is a consistent answer set of KB iff A is an answer set of $P \setminus b_U(P) \cup \mathit{res}(M)$, where M is an answer set of $b_U(P)$ and $\mathit{res}(M)$ is the program that contains exactly those facts which are in M. The proof is given for the strong answer-set semantics. The same line of reasoning holds for the weak answer-set semantics.

Let $KB = (L, P)$ be a dl-program. We assume a splitting set U (a set of non-ground literals) is given. Given an interpretation I, let I^U be the projection of atoms in I over predicate names appearing in U.

($\Rightarrow$) Assume A is a answer set for KB. First, we show that A^U is an answer set for $b_U(P)$. Indeed, let S be the least model of $\mathit{sb}_U(P)_L^{A^U}$. We prove that $S = A^U$:

A^U satisfies all the rules in $\mathit{sb}_U(P)_L^{A^U}$. Furthermore, $A^U \subset S$ cannot hold because then A^U would be the least model of $\mathit{sb}_U(P)_L^{A^U}$. Also, it cannot be that $A^U \supset S$, otherwise there would be an atom $a \in A^U$ whose predicate name appears in U not belonging to S. Since A is an answer set of P, a would be in the least model of sP_L^A. Note that, by definition of splitting set, $\mathit{sb}_U(P)_L^{A^U} \subseteq sP_L^A$. Since there are no atoms in $sP_L^A \setminus \mathit{sb}_U(P)_L^{A^U}$ having predicate name in U, then a must be necessary for modeling $\mathit{sb}_U(P)_L^{A^U}$ and thus belongs to S. It follows that $A^U = S$.

Furthermore, A is an answer set of $R = P \setminus b_U(P) \cup \mathit{res}(A^U)$. Indeed, let M be the least model of sR_L^A. We simply observe that both M and A must contain A^U, and both are models of sR_L^A. A cannot be smaller than the least model M. On the other hand if an atom $a \in (A \setminus M) \setminus A^U$ existed, then a would be in the least model of sP_L^A. Note that, except from the additional facts belonging to A^U, all the remaining rules of sR_L^A belong to sP_L^A as well. If a belongs to the least model of sP_L^A, then it must be also in M, otherwise we could build a smaller model of sP_L^A from M and A^U.

($\Leftarrow$) In this case, we assume that there are A, M, where A is an answer set for $R = P \setminus b_U(P) \cup \mathit{res}(A^U)$ and M is an answer set for $b_U(P)$.

A is then an answer set for P. Indeed, note that $A \supseteq M$, since all literals in M appear in R as a set of facts. Also, $A^U = M$. If we consider sP_L^A then we observe that

$$sP_L^A = (sR_L^A \setminus M) \cup \mathit{sb}_U(P)_L^A$$

Then A is a model of sP_L^A. There is no way for building a smaller model $A' \subset A$ for sP_L^A. An atom $a \in A^U$ cannot be eliminated from A since $A^U = M$ is the least model of $sb_U(P)_L^A$. Also, an atom $a \in A \setminus A^U$ cannot be eliminated since it would prevent A to be an answer set of R: indeed, a must appear at least in one rule head in $sR_L^A \setminus M$ otherwise it would belong to A^U. □

Our aim is to find the biggest subprogram which is stratified and does not depend an any unstratified rules:

Theorem 3.9.2 *Given $KB = (L, P)$, let V be the minimum set of classical literals such that (i) all literals a, b belong to V whenever $a \rightarrow_n b$ and $b \rightarrow^+ a$ holds in P and (ii) if $a \rightarrow b$ and $b \in V$, then $a \in V$. Then, the set $S = lit(P) \setminus V$ is a splitting set for P, where $lit(P)$ denotes the set of all literals in P. Moreover, it holds that $b_S(P)$ is stratified.*

Proof. Let V and S be as described. If we assume by contradiction that S is not a splitting set, then there exists some element $v \in V$ such that, for some $a \in S$, $a \rightarrow v$. This is impossible, since V is built in a way such that if $a \rightarrow v$ and $v \in V$ then $a \in V$.

$b_S(P)$ is clearly stratified since negative cycles are by definition put into V. □

We call such a splitting set S a *stratification splitting set* for a dl-program KB.

Corollary 3.9.3 *Each dl-program has exactly one stratification splitting set.*

Example 3.9.1 Consider the following dl-program KB:

$$p(X) \leftarrow q(X).$$
$$q(X) \leftarrow not\, r(X).$$
$$r(X) \leftarrow DL[C \uplus q](X), s(X).$$
$$s(X) \leftarrow t(X).$$

The set $\{s(X), t(X)\}$ is a stratification splitting set for P, since $r(X)$ and $q(X)$ belong to a negated cycle (over the input of the dl-atom) and $p(X)$ depends on this cycle. Only $s(X)$ and $t(X)$ are unaffected by the unstratified part of KB. ◇

Example 3.9.2 Consider the reviewer selection program from Example 3.2.1. The stratification splitting set of this program comprises all literals except those with the predicates *assign*, *a*, and *error*. Thus, it has the following stratified subprogram:

$$paper(p_1);\ kw(p_1, Semantic\_Web);$$
$$paper(p_2);\ kw(p_2, Bioinformatics);\ \ kw(p_2, Answer\_Set\_Programming);$$
$$kw(P, K_2) \leftarrow kw(P, K_1),\ DL[topicOf](S, K_1),\ DL[topicOf](S, K_2);$$
$$paperArea(P, A) \leftarrow DL[keywords \uplus kw; inArea](P, A);$$
$$cand(X, P) \leftarrow paperArea(P, A),\ DL[Referee](X),\ DL[expert](X, A);$$

It is obvious that this program can only have a single answer set. The unstratified part of the program are the remaining rules:

$$assign(X, P) \leftarrow cand(X, P), not\ \neg assign(X, P);$$
$$\neg assign(Y, P) \leftarrow cand(Y, P), assign(X, P),\ X \neq Y;$$
$$a(P) \leftarrow assign(X, P);$$
$$error(P) \leftarrow paper(P), not\ a(P).$$

◇

It follows directly from the splitting set theorem that the answer sets of a dl-program $KB = (L, P)$ can be obtained by adding the unique model of $b_U(P)$ to each answer set of the subprogram $P \setminus b_U(P)$.

In the following two sections, we explain how the computation of the models of both the stratified part and the unstratified part is carried out in practice.

Evaluation of a Stratified dl-Program

Algorithm 1 shows an implementation of a simple fixpoint computation for evaluating stratified dl-programs. Corresponding to Subsection 3.6.1, we use the replacement of a dl-atom $a(\mathbf{t})$ of form $DL[S_1 op_1 p_1, \ldots, S_m op_m p_m; Q](\mathbf{t})$ by a globally new atom $d_a(\mathbf{t})$. A ground fact of such a replacement atom is denoted by $d_a(\mathbf{c})$.

Algorithm 3.1: fixpoint(KB): Fixpoint computation of a dl-program.

Input: stratified dl-program $KB = (L, P)$
Result: single answer set
Replace all dl-atoms $a(\mathbf{t})$ in P by $d_a(\mathbf{t})$
$D' \Leftarrow \emptyset$
$R' \Leftarrow \emptyset$
repeat
  $D \Leftarrow D'$
  $R \Leftarrow R'$
  $P \Leftarrow P \cup D$; // Add the atoms in D as facts to P
  $R' \Leftarrow strat(P)$; // $strat(P)$ is the stratified answer set of P
  $D' \Leftarrow \{d_a(\mathbf{c}) \mid R' \models_L a(\mathbf{c})\}$; // i.e., all dl-atoms modeled by R' w.r.t. L
until $D = D'$
return R'

The algorithm iteratively computes the unique answer set of KB based on the previous one until a fixpoint is reached (which is determined by a non-changing set of dl-atoms in KB).

Theorem 3.9.4 *Let $KB = (L, P)$ be a stratified dl-program. If the set S is the result of* fixpoint(KB)*, then $S \cap HB_P$ is the iterative least model M_{KB} of KB.*

Proof. We can view the loop in the algorithm as an operator $F_{KB}(I)$ on the subsets of HB_P. For every $I \subseteq HB_P$, $F_{KB}(I)$ is the stratified answer set of $KB' = (L, P')$, where P' is a transformation of $ground(P)$ w.r.t. I as follows:

1. Remove each rule $r \in ground(P)$ with a dl-atom $a \in B^+(r)$ and $I \not\models_L a$;
2. remove each rule $r \in ground(P)$ with a dl-atom $a \in B^-(r)$ and $I \models_L a$;
3. from all remaining rules, remove all dl-atoms.

It is easy to see that this transformation corresponds to the addition of ground replacement atoms in D to the program.

First, we show that fixpoint(KB) is a monotone operator for positive KB with only monotonic dl-atoms. To show that $F_{KB}(I)$ is monotonic for a positive dl-program $KB_1 = (L_1, P_1)$ with only monotonic dl-atoms, i.e., $I \subseteq I' \subseteq HB_{P_1}$ implies $F_{KB_1}(I) \subseteq T_{KB_1}(I')$,

we only have to consider the first and third transformation rules. For every monotonic ground dl-atom a it holds that $I \models_{L_1} a$ implies $I' \models_{L_1} a$, and therefore $I' \not\models_{L_1} a$ implies $I \not\models_{L_1} a$. Hence, for I, the number of rules in P' must be equal or less than the number of rules in P. Thus, $F_{KB_1}(I) \subseteq F_{KB_1}(I')$. The least fixpoint of F_{KB_1}, $lfp(F_{KB_1})$ must satisfy all rules in KB_1. Hence, $lfp(F_{KB_1}) \cap HB_{P_1}$ is the least model of KB_1.

We construct this proof by induction along the stratification of KB. First, we consider stratum 0. All literals p with $\lambda(p) = 0$ must either be positive ordinary literals or positive and monotonic dl-atoms. We consider the program S^0 which contains all rules r such that $\lambda(H(r)) = 0$. Then, after a finite number of steps k, R' must contain the least model of S^0.

For any stratum $n > k$, let S^n be the set of rules $r \in ground(P)$ such that $\lambda(H(r)) = n$. Then, each classical literal $a \in B^-(r)$ and each nonmonotonic dl-atom in $B(r)$ must have been in a stratum lower than n and thus their truth value in the least model of KB is already known. Adding the least model of S^{n-1} corresponds to removing all rules r from $ground(P)$ where an atom $a \in B^-(r)$ is in the least model of S^{n-1} (resp. a nonmontonic dl-atom $a \in B^+(r)$ is not in the least model of S^{n-1}) and removing all other negative literals (resp. all other nonmontonic dl-atoms) from $B(r)$. The result is again a positive program without any nonmonotonic dl-atoms. □

Evaluation of a General dl-Program

Evidently, as soon as we have to deal with an unstratified (sub-)program, the fixpoint computation is not viable any more. Instead we use Algorithm 3.2, which follows the guess-and-check approach described in Subsection 3.6.1. Mind that this method for a general dl-program $KB = (L, P)$ as laid out there resulted in the weak answer sets of the program. To check whether a guess G, which is correct w.r.t. to L, is also a strong answer set, one has to verify whether the least model of the strong dl-transform sP_L^G is equal to G.

Top-Level Computation

The evaluation method of NLP-DL is now to find the stratification splitting set S of the input dl-program $KB = (L, P)$ and separate the stratified subprogram $P_{strat} = b_S(P)$ from the unstratified part $P_{unstrat} = P \setminus b_S(P)$. In Algorithm 3.3, we use Algorithm 1, denoted by fixpoint(KB) (returning a single model), and Algorithm 3.2, denoted by guess(KB) (returning a set of models).

It can be expected that this method of splitting the dl-program is of higher efficiency than the pure guess-and-check approach, since the "preliminary" computation of any stratified subprogram will in general narrow the search space of the guessing. It is obvious that a subsequent and more fine grained splitting into strongly and weakly connected components of the program will further optimize the computation. The efforts towards such a more sophisticated processing of the program's dependency information were eventually put into the reasoner for HEX-programs, dlvhex (see Chapter 5).

3.9.2 Well-Founded Semantics

We have shown in Subsection 3.5.2 how to generalize the original definition of the well-founded semantics (resp. the unfounded sets) to dl-programs. However, an implementation of WFS for KB by fixpoint iteration of the defining monotonic operator $W_{KB}(I)$ as in [Eiter et al., 2004b] is not attractive, since a polynomial-time algorithm for computing the greatest unfounded set of KB with respect to I, due to Condition (iii) of an unfounded set, is not evident (even if deciding $I \models_L l$ is polynomial).

Algorithm 3.2: guess(KB): Computing the strong answer sets of a general dl-program.

Input: general dl-program $KB = (L, P)$
Result: set of answer sets
Let A be the set of all dl-atoms in KB
Replace all dl-atoms $a(\mathbf{t})$ in P by $d_a(\mathbf{t})$
$D' \Leftarrow \{d_a(\mathbf{c}) \mid \emptyset \models_L a(\mathbf{c}), a(\mathbf{t}) \in A, A \text{ is monotonic}\}$
$P \Leftarrow P \cup D'$; `// Add the atoms in` D' `as facts to` P
$D \Leftarrow A \setminus D'$
forall $a \in D$ **do**
  $B' \Leftarrow B(r_a) \setminus a$, where r_a is the rule that has a in its body
  $r_{a,guess} \Leftarrow d_a(\mathbf{t}) \vee \neg d_a(\mathbf{t}) \leftarrow B', dom(\mathbf{t})$, where $dom(\mathbf{t})$ is a predicate whose extension comprises the entire HU_{KB}
  $P \Leftarrow P \cup \{r_{a,guess}\}$
end
$\mathcal{M} \Leftarrow \emptyset$
forall $R \in \mathcal{AS}(P)$ **do**
  $R_{dl} \Leftarrow \{d_a(\mathbf{c}) \mid d_a(\mathbf{c}) \in R\}$; `// Collect all dl-atoms in the result`
  if $R \setminus R_{dl} \models_L R_{dl}$ **then** `// Weak answer set?`
    if $LM(sP_L^R) = R$ **then** `// Strong answer set?`
      $\mathcal{M} \Leftarrow \mathcal{M} \cup R$
    end
  end
end
return $\mathcal{M}$

Algorithm 3.3: asp(KB): NLP-DL answer-set computation for a dl-program.

Input: general dl-program $KB = (L, P)$
Result: set of answer sets
$\mathcal{M} \Leftarrow \emptyset$
$M_{strat} \Leftarrow \emptyset$
identify stratification splitting set S of P
if $S \neq \emptyset$ **then**
  $P_{strat} \Leftarrow b_S(P)$
  $M_{strat} \Leftarrow$ fixpoint(KB'), where $KB' = (L, P_{strat})$
  $\mathcal{M} \Leftarrow \{M_{strat}\}$
end
if $b_S(P) \neq P$ **then**
  $P_{unstrat} \Leftarrow P \setminus b_S(P)$
  $\mathcal{M} \Leftarrow$ guess(KB'), where $KB' = (L, P_{unstrat} \cup M_{strat})$
end
return $\mathcal{M}$

As shown in Subsection 3.6.3, the WFS for KB, denoted $WFS(KB)$, is alternatively given by

$$WFS(KB) = lfp(\gamma^2_{KB}) \cup \{\neg a \mid a \in HB_P \setminus gfp(\gamma^2_{KB})\},$$

where the operator $\gamma_{KB}(I)$ assigns each interpretation $I \subseteq HB_P$ the least model M_{KB^I} of the strong reduct $KB^I = (L, sP^I_L)$. Since γ_{KB} is anti-monotonic, γ^2_{KB} is monotonic and thus has a least fixpoint $lfp(\gamma^2_{KB})$ and a greatest fixpoint $gfp(\gamma^2_{KB})$.

This way, $WFS(KB)$ is computable through a fixpoint iteration which computes and outputs the greatest and the least fixpoint of the γ^2_{KB} operator, starting from $\emptyset$ resp. HB_P (which may be represented by its complement). Since KB^I is a positive dl-program, machinery developed in Algorithm 1 for computing M_{KB^I} is very helpful in this respect. Caching of intermediate DL models (see next subsection) also proves to be very fruitful in this evaluation.

Algorithm 3.4 computes the result of the γ^2_{KB}-operator w.r.t. a specific interpretation.

Algorithm 3.4: $\mathsf{gamma}^2(KB, I)$: Computing $\gamma^2_{KB}(I)$.

Input: dl-program $KB = (L, P)$, interpretation I
Result: single model
Replace all dl-atoms $a(\mathbf{t})$ in P by $d_a(\mathbf{t})$
$D \Leftarrow \{d_a(\mathbf{c}) \mid I \models_L a(\mathbf{c})\}$; `// i.e., all dl-atoms modeled by` I `w.r.t.` L
$P' \Leftarrow P \cup D$
$M' \Leftarrow LM(sP'^I_L)$
$D \Leftarrow \{d_a(\mathbf{c}) \mid M' \models_L a(\mathbf{c})\}$; `// i.e., all dl-atoms modeled by` M' `w.r.t.` L
$P' \Leftarrow P \cup D$
$M \Leftarrow LM(sP'^I_L)$
return M

Having defined gamma^2, a general fixpoint computation is now trivial to specify (see Algorithm 3.5). The set of well-founded literals is $\mathsf{fp}(\emptyset, \gamma^2_{KB})$, while the set of unfounded literals is $\mathsf{fp}(HB_{KB}, \gamma^2_{KB})$.

Algorithm 3.5: $\mathsf{wfs}(KB, I)$: Computing the fixpoint of $\gamma^2_{KB}(I)$.

Input: dl-program $KB = (L, P)$, interpretation I
Result: set of facts
$I \Leftarrow S$
repeat
  $I' \Leftarrow I$
  $I \Leftarrow \mathsf{gamma}^2(I')$
until $I = I'$
return I

Enhancing answer-set generation with well-founded semantics Another interesting result from Subsection 3.5.4 allows to speed up the computation of the answer sets of a given $KB = (P, L)$ by means of a pre-evaluation of $WFS(KB)$:

Theorem 3.9.5 *Every strong answer set of a dl-program $KB = (L, P)$ includes $lfp(\gamma^2_{KB})$ and no atom $a \in HB_P \setminus gfp(\gamma^2_{KB})$.*

Proof. This result follows directly from Theorems 3.5.8 and 3.5.9. □

To consider this in the computation, we can exploit the possibility to introduce *constraints* to a DLV program (see Subsection 2.2.3). Constraints allow to filter out models which do not fulfill prescribed requirements. An intermediate ordinary program P' obtained from P can be then enriched with the constraint $\leftarrow not\, a$ for any atom a such that $a \in WFS(KB)$, and with a constraint $\leftarrow a$ for any atom a such that $\neg a \in WFS(KB)$. Notice that such constraints may also be added only for a subset of $WFS(KB)$ (e.g., the one obtained after some steps in the least resp. greatest fixpoint iteration of γ^2_{KB}). This technique proves to be useful for helping the answer-set programming solver to converge to solutions faster.

3.9.3 Efficient dl-Atom Evaluation and Caching

Since the calls to the DL-reasoner are a bottleneck in the coupling of an ASP solver with a DL-engine, special methods need to be devised in order to save on the number of calls to the DL-engine. To this end, we use complementary techniques.

DL-Function Calls

One of the features of DL-reasoners which may be fruitfully exploited for speed up are non-ground queries. RACER provides the possibility to retrieve in a function call all instances of a concept C (resp., of a role R) that are provable in the DL knowledge base. Given that the cost for accessing the DL-reasoner is high, in the case when several different ground instances $a(\mathbf{c}_1), a(\mathbf{c}_2), \ldots, a(\mathbf{c}_k)$ of the dl-atom $a(\mathbf{t})$ have be evaluated, it is a reasonable strategy to retrieve at once, using the apposite function call feature from the DL-reasoner, all instances of the concept C (resp., a role R) in $a(\mathbf{t}) = DL[S_1 op_1 p_1, \ldots; C](\mathbf{t})$. This allows to avoid issuing k separate calls for the single ground atoms $a(\mathbf{c}_1)$, $\ldots$, $a(\mathbf{c}_k)$.

If the retrieval set has presumably many more than k elements, we can filter it with respect to $\mathbf{c}_1, \ldots, \mathbf{c}_k$, by pushing these instances to a DL-engine as follows. For the query concept C, we add in L axioms to the effect that $C'' = C \sqcap C'$, where C' and C'' are fresh concept names, and axioms $C'(\mathbf{c}_1)$, $\ldots$, $C'(\mathbf{c}_k)$; then we ask for all instances of C''. For roles, a similar yet more involved approximation method is introduced, given that $\mathcal{SHIF}(\mathbf{D})$ and $\mathcal{SHOIN}(\mathbf{D})$ do not offer role intersection.

With the above techniques, the number of calls to the DL-reasoner can be greatly reduced. Another very useful technique to achieve this goal is caching.

DL-Caching

Whatever semantics is considered, a number of calls will be made to the DL-engine. Therefore, it is is very important to avoid an unnecessary flow of data between the two engines, and to save time when a redundant DL-query has to be made. In order to achieve these objectives, it is important to introduce some special caching data structures tailored for fast access to previous query calls. Such a caching system needs to deal with the case of Boolean as well as non-Boolean DL-calls.

For any dl-atom $DL[\lambda; Q](\mathbf{t})$, where λ is a list $S_1 op_1 p_1, \ldots, S_n op_n p_n$, and interpretation I, let us denote by I^λ the projection of I on $p_1, \ldots, p_n$.

Boolean DL-calls. In this case, an external call must be issued in order to verify whether a given ground dl-atom b fulfills $I \models_L b$, where I is the current interpretation and L is the DL-knowledge base hosted by the DL-engine. In this setting, the caching system exploits properties of monotonic dl-atoms $a = DL[\lambda; Q](\mathbf{c})$.

Given two interpretations I_1 and I_2 such that $I_1 \subseteq I_2$, monotonicity of a implies that (i) if $I_1 \models_L a$ then $I_2 \models_L a$, and (ii) if $I_2 \not\models_L a$ then $I_1 \not\models_L a$. This property allows to set up a caching machinery where only the outcome for ground dl-atoms with minimal/maximal input is stored.

Roughly speaking, for each monotonic ground dl-atom a we store a set $\mathit{cache}(a)$ of pairs $\langle I^\lambda, o\rangle$, where $o \in \{\mathit{true}, \mathit{undefined}\}$. If $\langle I^\lambda, \mathit{true}\rangle \in \mathit{cache}(a)$, then we can conclude that $J \models_L a$ for each J such that $I^\lambda \subseteq J^\lambda$. Dually, if $\langle I^\lambda, \mathit{undefined}\rangle \in \mathit{cache}(a)$, we can conclude that $J \not\models_L a$ for each J such that $I^\lambda \supseteq J^\lambda$.

We sketch the maintenance strategy for $\mathit{cache}(a)$ in the following. The rationale is to cache minimal (resp., maximal) input sets I^λ for which a is evaluated to *true* (resp., *undefined*) in past external calls.

Suppose a ground dl-atom $a = \mathit{DL}[\lambda; Q](\mathbf{c})$, an interpretation I, and a cache set $\mathit{cache}(a)$ are given. With a small abuse of notation, let $I(a)$ be a function whose value is *true* iff $I \models_L a$ and *undefined* otherwise. In order to check whether $I \models_L a$, $\mathit{cache}(a)$ is consulted and updated as follows:

1. Check whether $\mathit{cache}(a)$ contains some $\langle J, o\rangle$ such that $J \subseteq I^\lambda$ if $o = \mathit{true}$, or $J \supseteq I^\lambda$ if $o = \mathit{undefined}$. If such J exists, conclude that $I(a) = o$.

2. If no such J exists, then decide $I \models_L a$ through the external DL-engine. If $I \models_L a$, then add $\langle I^\lambda, \mathit{true}\rangle$ to $\mathit{cache}(a)$, and remove from it each pair $\langle J, \mathit{true}\rangle$ such that $I^\lambda \subset J$. Otherwise (i.e., if $I \not\models_L a$) add $\langle I^\lambda, \mathit{undefined}\rangle$ to $\mathit{cache}(a)$ and remove from it each pair $\langle J, \mathit{undefined}\rangle$ such that $I^\lambda \supset J$.

Some other implementational issues are worth mentioning. First of all, since the subsumption test between sets of atoms is a critical task, some optimization is made in order to improve cache look-up. For instance, an element count is stored for each atom set, in order to prove early that $I \not\subseteq J$ whenever $|I| > |J|$. More intelligent strategies could be envisaged in this respect. Furthermore, a standard *least recently used* (LRU) algorithm has been introduced in order to keep a fixed cache size.

Non-Boolean DL-calls. In most cases, a single non-ground query for retrieving all instances of a concept or role might be employed. Caching of such queries is also possible, but cache look-up cannot take advantage of monotonicity as in the Boolean case. For each non-ground dl-atom $a = \mathit{DL}[\lambda; Q](\mathbf{c})$, a set $\mathit{cache}(a)$ of pairs $\langle I^\lambda, a{\downarrow}(I^\lambda)\rangle$ is maintained, where $a{\downarrow}(I)$ is the set of all ground instances a' of a such that $I \models_L a'$. Whenever for some interpretation I, $a{\downarrow}(I)$ is needed, then $\mathit{cache}(a)$ is looked up for some pair $\langle J, a{\downarrow}(J)\rangle$ such that $I^\lambda = J$.

3.9.4 Prototype

The architecture of our system prototype NLP-DL is depicted in Figure 3.1. The system comprises different modules, each of which is coded in the PHP scripting language; the overhead is insignificant, provided that most of the computing power is devoted to the execution of the two external reasoners. Moreover, the choice of this language enabled us to make the prototype easily accessible by a Web-interface, thus serving its main purposes as a testing and demonstration tool. The Web-interface allows the user to enter a dl-program KB in form of an OWL-ontology L and an answer-set program P. It can then be used either to compute the model(s) or to perform reasoning, both according to the selected semantics, which can be chosen between the strong answer-set semantics and the well-founded semantics. The second mode requires the specification of one or more query

atoms as input from the user; here, another choice between brave and cautious reasoning is available. Furthermore, the result can be filtered by specific predicate names.

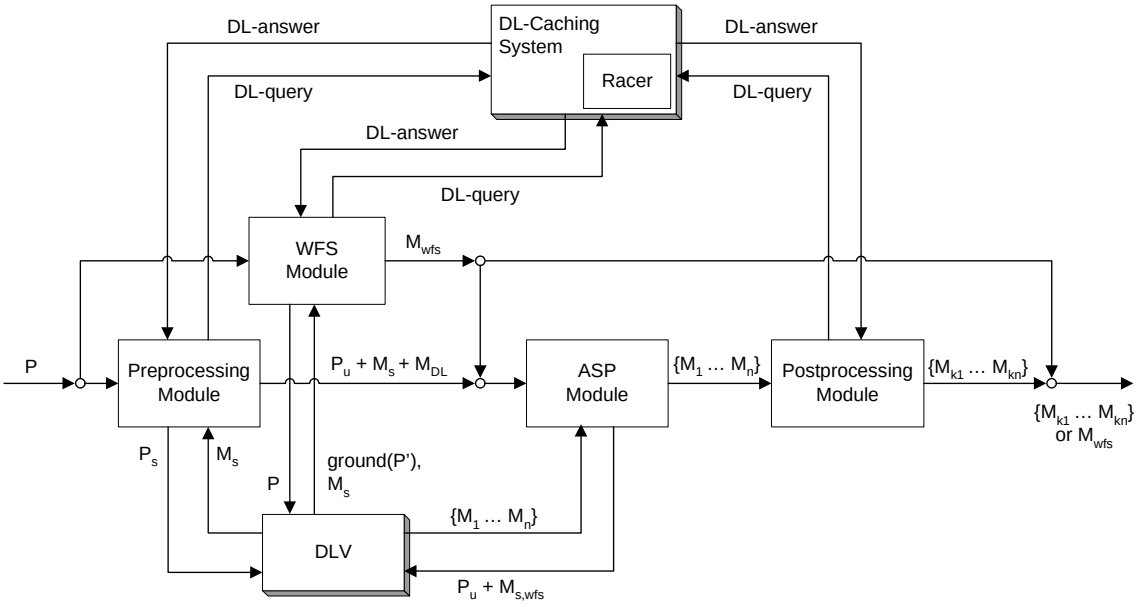


Figure 3.1: System architecture of the dl-program evaluation prototype.

The shadowed boxes represent the external reasoning engines: `DLV` [Leone et al., 2006] was used as answer-set solver and RACER [Haarslev and Möller, 2001] as description Logics Reasoner, which is embedded in a caching module.

Our prototypical implementation is capable of evaluating a dl-program in three different modes: (1) under answer-set semantics, (2) under WFS, and (3) under answer-set semantics with preliminary computation of the WFS.

The preprocessing module evaluates all dl-atoms without any input (M_{DL}), applies the splitting method (separating the unstratified subprogram P_u) and computes the single answer set of the stratified subprogram (M_s). The ASP module implements the guessing part of the evaluation, using `DLV` for the answer-set generation ($\{M_1 \dots M_n\}$). This result is streamed to a post-processing module, which carries out the verification of each incoming answer set according to the weak- resp. strong answer-set semantics, return the final result $\{M_{k1} \dots M_{kn}\}$. The WFS module is used for computing the well-founded model M_{wfs}. It uses `DLV` not only as a solver, but also as a grounder to be able to transform the program according to the GL-reduct. As we have shown in the previous subsection, the well-founded model approximates the intersection of all strong answer sets and thus can optionally be added to the input of the ASP module.

3.10 Related Work

As we have already pointed out in the introduction of this chapter, related works on combining rules and ontologies can essentially be grouped into the following three lines of research:

(a) interaction of rules and ontologies with strict semantic separation (loose coupling),

(b) interaction of rules and ontologies with strict semantic integration (tight coupling), and

(c) reductions from description logics to answer-set programming (ASP) and/or other formalisms.

Interaction of Rules and Ontologies with Strict Semantic Separation

Here, the (usually nonmonotonic) rules component is kept strictly separate from the ontology layer, where OWL/RDF flavors keep their purpose of description languages, not aimed at intensive reasoning jobs. The two layers only communicate via a "safe interface", and no syntactic restrictions on either the rules or the ontology part are imposed.

From the rules layer point of view, ontologies are dealt with as an external source of information whose semantics is treated separately. Non-monotonic reasoning and rules are allowed in a decidable setting, as well as arbitrary mixing of closed and open world reasoning. This approach typically involves special predicates in rule bodies which allow queries to a DL knowledge base, and exchange factual knowledge. Examples for this type of interaction are dl-programs themselves, their extension to HEX-programs (see Chapter 4), to probabilistic dl-programs [Lukasiewicz, 2005a,b], and to fuzzy dl-programs [Lukasiewicz, 2006]. HEX-programs extend the framework of dl-programs so that multiple sources of external knowledge, with possibly different semantics, can be combined in a single logic program, while probabilistic dl-programs and fuzzy dl-programs extend dl-programs by probabilistic uncertainty and fuzzy vagueness/imprecision, respectively.

Further work in this direction is due to Antoniou [2002], which deals with a combination of defeasible reasoning with Description Logics. Like in other work mentioned above, the considered description logic serves here only as an input for the default reasoning mechanism running on top of it. Wang et al. [2005] extend dl-programs by a framework conceived for alignment of ontologies. Also, early work on dealing with default information in the context of description logic is the approach by Baader and Hollunder [1995], where Reiter's default logic is adapted to terminological knowledge bases, differing significantly from our approach. Less closely related work includes also the investigations by Baumgartner et al. [2002] and Provetti et al. [2003]. Similar in spirit is also the notion of call to external description logic reasoners in the TRIPLE [Sintek and Decker, 2002] rules engine.

Interaction of Rules and Ontologies with Strict Semantic Integration

This category comprises formalisms that introduce rules by adapting existing semantics for rule languages directly in the ontology layer. Recently, several proposals have been made to extend expressiveness while still retaining decidability, remarkably several attempts in the ASP field. Common to these approaches are syntactic restrictions of the combined language in a way that guarantees "safe interaction" of the rules and the ontology parts of the language.

Grosof et al. [2003] show how inference in a subset of the description logic $\mathcal{SHOIQ}$ can be reduced to inference in a subset of Horn programs (in which no function symbols, negations, and disjunctions are permitted), and vice versa. This work resulted in the conception of the Web Rule Language (WRL) proposal [Angele et al., 2005].

The works by Donini et al. [1998], Levy and Rousset [1998], and Rosati [1999] are representatives of hybrid approaches using Description logics knowledge as input. In detail, Donini et al. introduce a combination of (disjunction-, negation-, and function-free)

datalog with the description logic $\mathcal{ALC}$. An integrated knowledge base consists of a structural component in $\mathcal{ALC}$ and a relational component in datalog, where the integration of both components lies in using concepts from the structural component as constraints in rule bodies of the relational component. Donini et al. also present a technique for answering conjunctive queries (existentially quantified conjunctions of atoms) with such constraints, where SLD-resolution as an inference method for datalog is integrated with a method for inference in $\mathcal{ALC}$. The closely related work by Levy and Rousset [1998] presents a combination of Horn rules with the description logic $\mathcal{ALCNR}$. In contrast to Donini et al., Levy and Rousset also allow for roles as constraints in rule bodies, and do not require the safety condition that variables in constraints in the body of a rule r must also appear in ordinary atoms in the body of r. Levy and Rousset also present a technique for answering queries, which are of the very general form of disjunctions of conjunctive queries, conditioned on conjunctive queries. Rosati [1999] presents a combination of disjunctive datalog (with classical and default negation, but without function symbols) with the description logic $\mathcal{ALC}$, which is based on a generalized answer-set semantics. Similarly to Levy and Rousset, here Rosati also allows for roles as constraints in rule bodies, and does not require the above-mentioned safety condition. He presents a technique for answering queries of the form of ground atoms, which is based on a combination of ordinary answer-set programming with inference in $\mathcal{ALC}$.

The decidability result for so-called *DL-safe rules* (presented in Subsection 3.8.3) is extended to a more expressive description logic $\mathcal{SHIQ}$ in [Motik et al., 2005] bringing us closer to OWL, whereas in SWRL [Horrocks et al., 2004] the safety restriction, which retains decidability, is not enforced. Another approach in this direction by Heymans et al. [2005a] shows decidability for query answering in $\mathcal{ALCHO}$-$\mathcal{Q}(\sqcup, \sqcap)$ with DL-safe rules by an embedding in extended conceptual logic programming, a decidable extension of the answer-set semantics by open domains. The most recent work in this direction by Rosati [2005, 2006a,b] further weakens the safety restriction, by allowing non-rule atoms also in rule heads, and also gives a nonmonotonic semantics for non-Horn rules in the spirit of answer-set programming.

Rosati's $\mathcal{DL}$+*log* formalism [2006a, 2006b], which extends his previous work is the closest in spirit to dl-programs. In this approach, predicates are split into *ontology predicates* and into *logic program* (*datalog*) *predicates*. A notion of model of a combined rule and ontology knowledge base is defined using a two-step reduct in which, in the first step, the ontology predicates are eliminated under the open-world assumption (OWA) and, in the second step, the negated logic programming predicates under the closed-world assumption (CWA). As shown by Rosati, the emerging formalism (which focuses on first-order models under the standard-names assumption over infinite universes), is decidable provided that containment of conjunctive queries and union of conjunctive queries over the underlying ontology is decidable. The main differences between $\mathcal{DL}$+*log* and dl-programs are:

- $\mathcal{DL}$+*log* is a tight coupling of rules and ontologies, while dl-programs provide a loose coupling of rules and ontologies.
- While extensions of dl-programs to integrate ontologies even in different formats are straightforward, there is no corresponding counterpart in $\mathcal{DL}$+*log*. Indeed, the approach of dl-atoms is more flexible for mixing different reasoning modalities, such as consistency checking and logical consequence. In the realm of HEX-programs, almost arbitrary combinations can be conceived.
- The coupling as realized in dl-programs aims at facilitating interoperability of existing reasoning systems and software (such as DLV and RACER). On the other hand, the

loose coupling requires a bridging between the two worlds of ontologies and rules, which has to be provided by the user. In particular, this applies to the individuals at the instance level.

Reductions from Description Logics to ASP and/or Other Formalisms

Some representatives of approaches reducing Description Logic to logic programming are the works by van Belleghem et al. [1997], Alsaç and Baral [2001], Swift [2004], Grosof et al. [2003], and Heymans and Vermeir [2003a,b]. In detail, van Belleghem et al. analyze the close relationship between Description Logics and open logic programs, and present a mapping of description logic knowledge bases in $\mathcal{ALCN}$ to open logic programs. They also show how other description logics correspond to sublanguages of open logic programs, and they explore the computational correspondences between a typical algorithm for description logic inference and the resolution procedure for open logic programs. The works by Alsaç and Baral and Swift reduce inference in the description logic $\mathcal{ALCQI}$ to query answering from normal logic programs (with default negation, but without disjunctions and classical negations) under the answer-set semantics.

The remarkable work of Motik et al. [Hustadt et al., 2004] considers $\mathcal{SHIQ}$ ontologies. Query answering is reduced to the evaluation of a positive disjunctive datalog program. Such a program is generated after an ordinary translation to first-order logic, followed by the application of superposition techniques. The latter aims at eliminating function symbols from the first-order theory. The method has been practically adopted in the KAON2 system, whose experimental results are accounted in [Motik and Sattler, 2006].

Finally, Heymans and Vermeir [2003a,b] present an extension of disjunctive logic programming under the answer-set semantics by inverses and an infinite universe. In particular, they prove that this extension is still decidable under the assumption that the rules form a tree structure, and they show how inference in the description logic $\mathcal{SHIF}$ extended by transitive closures of roles can be simulated in it.

Chapter 4

HEX-Programs

The development of dl-programs as presented in Chapter 3 lead to a novel framework that couples logic programming and description logic reasoning in a fully declarative formalism. It enables the user to solve sophisticated reasoning tasks that go beyond the power of ontology languages such as OWL. However, this interface is restricted to a specific description logic, covering only a fraction of real-world Semantic Web use cases. In this chapter, we present a more general logic programming framework, which still based on answer-set semantics, but facilitates a much more versatile interface mechanism and additionally introduces higher-order reasoning. We call this novel kind of programs HEX-*programs*, that is, *higher-order* logic programs with *external atoms*. Both features will be introduced below.

4.1 Introduction

From a conceptual viewpoint, the characteristic feature of the interface between a logic program and an external reasoning formalism that was introduced by dl-programs, is to leave the semantics of each side untouched (what we called *strict semantic separation*). The DL knowledge base presents itself as a "black box" to the logic program. The user does not need to know its entire signature, it is sufficient that she is aware of a subset of its concepts and roles, enabling her to extend and query them. This strict separation saved us from any decidability issues that come along a tighter integration of such diverse formalisms.

In order to extend the answer-set semantics to the framework of dl-programs, basically only the satisfiability relation for dl-atoms had to be introduced; a dl-program without any such atoms behaves exactly like an ordinary answer-set program. On the other hand, the satisfiability of a (ground) dl-atom depends on the entailment of its query w.r.t. the respective DL knowledge base. The possibility of augmenting the assertional knowledge of the DL KB was provided by supplying an interpretation along with the ground dl-atom. Let us recall that a dl-atom not only specifies a dl-query, but also the specific input to the ABox. Consider the following example: We want to query the concept *IsBusy* from an ontology, after we add the extension of the logic-program predicate *worksWith* the role *collaborates*, and the extension of *holiday* to the negation of the concept *InOffice*. This task can be accomplished by the following dl-atom:

$$DL[collaborates \uplus worksWith, InOffice \uplus holiday; IsBusy](X)$$

If we only consider the information flow between the DL-reasoner and the logic program,

we can abstract the parts of this dl-atom the following way:

$$\begin{aligned} \text{input-specification:} \quad & \mathit{collaborates} \uplus \mathit{worksWith}, \mathit{InOffice} \uplus \mathit{holiday}; \mathit{IsBusy} \\ \text{input:} \quad & \text{input-specification} + I \\ \text{output:} \quad & X \end{aligned}$$

In fact, we can further generalize this interfacing method and speak about an arbitrary *external source* of knowledge, which returns a set of ground output tuples based on such an input-specification together with an interpretation. In this respect, we do not need to speak of dl-atoms anymore, but can call them *external atoms*. The truth value of a ground external atom is determined by some external evaluation function and the atom's input as defined above. This input is in principle only a string, but in practice we can categorize it into constants or names of predicates. For instance, the rule

$$\mathit{reached}(X) \leftarrow \&\mathit{reach}[\mathit{edge}, a](X)$$

computes the predicate *reached* taking values from the predicate $\&reach$, which computes via $\&reach[edge, a]$ all the reachable nodes in the graph *edge* from node a, delegating this task to an external computational source (e.g., an external deduction system, an execution library, etc.).

Evidently, with this concept we still stick to the idea of using a loose, semantically "safe interface". The logic program that uses such external atoms still obeys the answer-set semantics, provided that we properly define the satisfiability of such special atoms. On the other hand, any such external function does not need to know anything about logic-programming semantics, since it is regarded as a black box with a clearly specified input and output. Of course, different types of external atoms may be used within a single program, providing a framework to interoperate with a variety of external software at the same time, combining heterogeneous knowledge under the multiple-model-generating semantics of ASP (See Figure 4.1). In Chapter 5, where we present the implementation of a HEX-reasoner, we outline the method how the prototype interfaces the external reasoners in practice and present a number of such interfaces (called *plugins*) that are already available.

Apart from the concept of external atoms, HEX-programs offer another extension with respect to traditional answer-set programs, namely the possibility to use higher order syntax. For important issues such as *meta-reasoning* in the context of the Semantic Web, no adequate support is available in ASP to date. Intuitively, a *higher-order atom* allows to quantify values over predicate names, and to freely exchange predicate symbols with constant symbols, like in the rule

$$C(X) \leftarrow \mathit{subClassOf}(D, C), D(X).$$

Especially when dealing with ontological knowledge on the Semantic Web, this enables us to intuitively define specific semantics by means of logic programming rules, as for the subsumption relationship in the rule above.

In this chapter, we will present the following contributions:

- We define the syntax and answer-set semantics of HEX programs, extending ASP with higher-order features and powerful interfacing of external computation sources. While answer-set semantics for higher-order logic programs has been proposed earlier by Ross [1994], further extension of that proposal to accommodate external atoms is technically difficult since the approach of Ross is based on the notion of unfounded

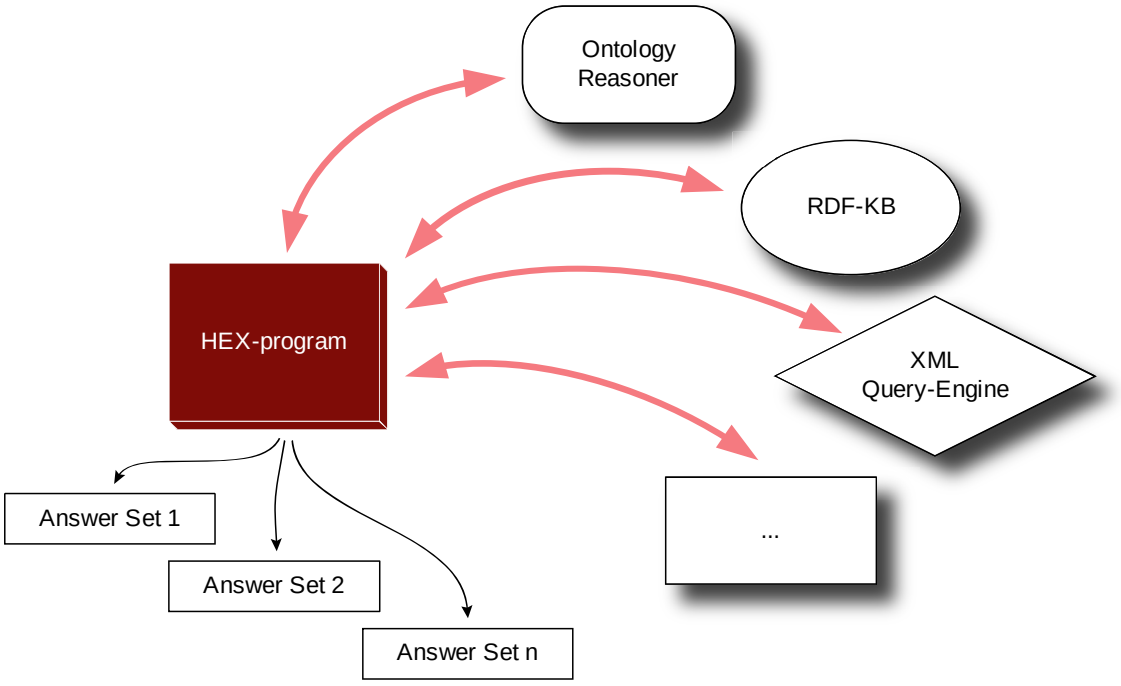


Figure 4.1: HEX-program Concept.

set, which cannot be easily generalized to this setting. Our approach, instead, is based on a recent notion of program reduct due to Faber et al. [2004], which admits a natural definition of answer-set semantics.

- We will discuss external atoms as a useful abstraction of several extensions to ASP including, among others, aggregates, description logic atoms, or agent programs. External atoms thus facilitate investigating common properties of such extensions, and can serve as a uniform framework for defining semantics of further similar extensions of ASP. Moreover, HEX-programs are a basis for the efficient design of generic evaluation algorithms for such extensions in this framework.

- By means of HEX-programs, powerful meta-reasoning becomes available in a decidable context, e.g., for Semantic Web applications, for meta-interpretation in ASP itself, or for defining policy languages. For example, advanced closed world reasoning or the definition of constructs for an extended ontology language (e.g., of RDF Schema) is well-supported. Due to the higher-order features, the representation is succinct.

- Eventually, we will give a detailed account of methods how to compute the answer sets of a HEX-program under the condition of using an existing solver for traditional ASP. To this end, we will define structural properties of a HEX-program that will enable us to evaluate it by splitting it into components. Additionally, the complexity of solving a HEX-program is surveyed.

Note that other logic-based formalisms, like TRIPLE [Sintek and Decker, 2002] or F-Logic [Kifer et al., 1995], feature also higher-order predicates for meta-reasoning in Semantic Web applications. However, TRIPLE is low-level oriented and lacks precise semantics, while F-Logic in its implementations (Flora, Florid, Ontoweb) restricts its expressiveness to well-founded semantics for negation, in order to gain efficiency. Our formalism, instead,

is fully declarative and offers the possibility of nondeterministic predicate definition with higher complexity. This proved already useful and reasonably efficient for a range of applications with inherent nondeterminism, such as diagnosis, planning, or configuration, and thus provides a rich basis for integrating these areas with meta-reasoning.

In the course of Section 5.3, where we introduce already implemented external atoms, a number of illustrative examples will demonstrate the convenience of HEX-programs.

4.2 HEX-Program Syntax

Let $\mathcal{C}$, $\mathcal{X}$, and $\mathcal{G}$ be mutually disjoint sets whose elements are called *constant names*, *variable names*, and *external predicate names*, respectively. Unless explicitly specified, elements from $\mathcal{X}$ (resp., $\mathcal{C}$) are denoted with first letter in upper case (resp., lower case), while elements from $\mathcal{G}$ are prefixed with "&". We note that constant names serve both as individual and predicate names.

Elements from $\mathcal{C} \cup \mathcal{X}$ are called *terms*. A *higher-order atom* (or *atom*) is a tuple $(Y_0, Y_1, \ldots, Y_n)$, where $Y_0, \ldots, Y_n$ are terms; $n \geq 0$ is the *arity* of the atom. Intuitively, Y_0 is the predicate name, and we thus also use the more familiar notation $Y_0(Y_1, \ldots, Y_n)$. The atom is *ordinary*, if Y_0 is a constant.

For example, $(x, rdf{:}type, c)$, $node(X)$, and $D(a, b)$, are atoms; the first two are ordinary atoms.

An *external atom* is of the form

$$\&g[Y_1, \ldots, Y_n](X_1, \ldots, X_m), \tag{4.1}$$

where $Y_1, \ldots, Y_n$ and $X_1, \ldots, X_m$ are two lists of terms (called *input* and *output* lists, respectively), and $\&g \in \mathcal{G}$ is an external predicate name. We assume that $\&g$ has fixed lengths $in(\&g) = n$ and $out(\&g) = m$ for input and output lists, respectively. Intuitively, an external atom provides a way for deciding the truth value of an output tuple depending on the extension of a set of input predicates.

Example 4.2.1 The external atom $\&reach[edge, a](X)$ may be devised for computing the nodes which are reachable in the graph *edge* from the node a. Here, we have that $in(\&reach) = 2$ and $out(\&reach) = 1$. ◇

A *rule* r is of the form

$$\alpha_1 \vee \cdots \vee \alpha_k \leftarrow \beta_1, \ldots, \beta_n, not\ \beta_{n+1}, \ldots, not\ \beta_m, \tag{4.2}$$

where $m, k \geq 0$, $\alpha_1, \ldots, \alpha_k$ are atoms, and $\beta_1, \ldots, \beta_m$ are either atoms or external atoms. We define $H(r) = \{\alpha_1, \ldots, \alpha_k\}$ and $B(r) = B^+(r) \cup B^-(r)$, where $B^+(r) = \{\beta_1, \ldots, \beta_n\}$ and $B^-(r) = \{\beta_{n+1}, \ldots, \beta_m\}$. If $H(r) = \emptyset$ and $B(r) \neq \emptyset$, then r is a *constraint*, and if $B(r) = \emptyset$ and $H(r) \neq \emptyset$, then r is a *fact*; r is *ordinary*, if it contains only ordinary atoms. Note that in contrast to dl-programs, HEX-programs allow for disjunctive heads and constraints.

A HEX-*program* is a finite set P of rules. It is *ordinary*, if all rules are ordinary.

4.3 Semantics of HEX-Programs

We define the semantics of HEX-programs by generalizing the answer-set semantics by Gelfond and Lifschitz [1991]. To this end, we use the recent notion of a reduct as defined

by Faber et al. [2004] (referred to as *FLP-reduct* henceforth) instead of to the traditional reduct by Gelfond and Lifschitz [1991]. The FLP-reduct admits an elegant and natural definition of answer sets for programs with aggregate atoms, since it ensures answer-set minimality, while the definition based on the traditional reduct lacks this important feature.

In the sequel, let P be a HEX-program. The *Herbrand base* of P, denoted HB_P, is the set of all possible ground versions of atoms and external atoms occurring in P obtained by replacing variables with constants from $\mathcal{C}$. The grounding of a rule r, $grnd(r)$, is defined accordingly, and the grounding of program P is given by $grnd(P) = \bigcup_{r \in P} grnd(r)$. Unless specified otherwise, $\mathcal{C}$, $\mathcal{X}$, and $\mathcal{G}$ are implicitly given by P.

Example 4.3.1 Given $\mathcal{C} = \{edge, arc, a, b\}$, ground instances of $E(X, b)$ are for instance $edge(a, b)$, $arc(a, b)$, $a(edge, b)$, and $arc(arc, b)$; ground instances of $\&reach[edge, N](X)$ are $\&reach[edge, edge](a)$, $\&reach[edge, arc](b)$, and $\&reach[edge, edge](edge)$, etc. ◇

An *interpretation relative to* P is any subset $I \subseteq HB_P$ containing only atoms. We say that I is a *model* of atom $a \in HB_P$, denoted $I \models a$, if $a \in I$.

With every external predicate name $\&g \in \mathcal{G}$, we associate an $(n+m+1)$-ary Boolean function $f_{\&g}$ assigning each tuple $(I, y_1 \ldots, y_n, x_1, \ldots, x_m)$ either 0 or 1, where $n = in(\&g)$, $m = out(\&g)$, $I \subseteq HB_P$, and $x_i, y_j \in \mathcal{C}$. We say that $I \subseteq HB_P$ is a *model* of a ground external atom $a = \&g[y_1, \ldots, y_n](x_1, \ldots, x_m)$, denoted $I \models a$, if and only if $f_{\&g}(I, y_1 \ldots, y_n, x_1, \ldots, x_m) = 1$.

Note that in contrast to the semantics of higher-order atoms, which in essence reduces to first-order logic as customary (cf. [Ross, 1994]), the semantics of external atoms is in spirit of second order logic since it involves predicate extensions.

Example 4.3.2 Let us associate with the external atom $\&reach$ a function $f_{\&reach}$ such that $f_{\&reach}(I, E, A, B) = 1$ iff B is reachable in the graph E from A. Let $I = \{e(b, c), e(c, d)\}$. Then, I is a model of $\&reach[e, b](d)$ since $f_{\&reach}(I, e, b, d) = 1$. ◇

Let r be a ground rule. We define (i) $I \models H(r)$ iff there is some $a \in H(r)$ such that $I \models a$, (ii) $I \models B(r)$ iff $I \models a$ for all $a \in B^+(r)$ and $I \not\models a$ for all $a \in B^-(r)$, and (iii) $I \models r$ iff $I \models H(r)$ whenever $I \models B(r)$. We say that I is a *model* of a HEX-program P, denoted $I \models P$, iff $I \models r$ for all $r \in grnd(P)$. We call P *satisfiable*, if it has some model.

Given a HEX-program P, the *FLP-reduct* of P with respect to $I \subseteq HB_P$, denoted fP^I, is the set of all $r \in grnd(P)$ such that $I \models B(r)$. $I \subseteq HB_P$ is an *answer set of* P iff I is a minimal model of fP^I.

We next give an illustrative example.

Example 4.3.3 Consider the following HEX-program P:

(1) $subRelation(brotherOf, relativeOf)$.
(2) $brotherOf(john, al)$.
(3) $relativeOf(john, joe)$.
(4) $brotherOf(al, mick)$.
(5) $invites(john, X) \vee skip(X) \leftarrow X \neq john, reach[relativeOf, john](X)$.
(6) $R(X, Y) \leftarrow subRelation(P, R), P(X, Y)$.
(7) $\leftarrow \&degs[invites](Min, Max), Min < 1$.
(8) $\leftarrow \&degs[invites](Min, Max), Max > 2$.

Informally, this program randomly selects a certain number of John's relatives for invitation. The first line states that *brotherOf* is a subrelation of *relativeOf*, and the next three lines give concrete facts. The disjunctive rule (5) chooses relatives, employing the external predicate &*reach* from Example 4.3.2. Rule (6) declares a generic subrelation inclusion exploiting higher-order atoms.

The constraints (7) and (8) ensure that the number of invitees is between 1 and 2, using (for illustration) an external predicate &*degs* from a graph library, where $f_{\&degs}(I, E, Min, Max)$ is 1 iff Min and Max is the minimum and maximum vertex degree of the graph induced by the edges E, respectively. As John's relatives are determined to be Al, Joe, and Mick, P has six answer sets, each of which contains one or two of the facts $invites(john, al)$, $invites(john, joe)$, and $invites(john, mick)$. ◇

In principle, the truth value of an external atom depends on its input and output lists and the entire model of the program. Practically however, we can identify certain types of input terms that allow to restrict the input interpretation to specific relations. The atom $\&reach[edge, a](X)$ for instance will only consider the extension of the predicate *edge* and the constant value a for computing its result and simply ignore the remaining interpretation. In Chapter 5 we will formalize these two types of input terms and restrict the practical usage of external atoms to them, since such type information will support an efficient evaluation to a great extent.

We now state some basic properties of the semantics.

Theorem 4.3.1 *The answer-set semantics of* HEX*-programs extends the answer-set semantics of ordinary programs as defined by Gelfond and Lifschitz [1991], as well as the answer-set semantics of HiLog programs as defined by Ross [1994].*

Proof. Let P be a HEX-program without any external atoms. The semantics of P directly correspond to the classical answer-set semantics. □

The next property, which is easily proved, expresses that answer sets adhere to the principle of minimality.

Theorem 4.3.2 *Every answer set of a* HEX*-program P is a minimal model of P.*

Proof. First, we show that an answer set A of P is also a model of P. This follows from the fact that each answer set A is a least model of the FLP-reduct of P. Hence, A must satisfy each rule r in fP^A. If a rule r was removed by the reduct, it is trivially satisified by A. Thus, A is a model of P.

We prove minimality by contradiction. Assume that I is an answer set of a HEX-program P, J a model of P and that $J \subset I$. Since I is an answer set of P, it must be a minimal model of fP^I. But then, J cannot be a model of fP^I, hence there must be a rule $r \in ground(fP^I)$ such that r is unsatisfied w.r.t. J. Due to the nature of the FLP-reduct, $fP^I \subseteq fP^J$, thus r must also be in $r \in ground(fP^J)$ and therefore J cannot be a model of P. □

A ground external atom a is called *monotonic relative to* P iff $I \subseteq I' \subseteq HB_P$ and $I \models a$ imply $I' \models a$. For instance, the ground versions of $\&reach[edge, a](X)$ are all monotonic.

Theorem 4.3.3 *Let P be a* HEX*-program without "not" and constraints. If all external atoms in $grnd(P)$ are monotonic relative to P, then P has some answer set. Moreover, if P is disjunction-free, it has a single answer set.*

Proof. A positive program with only montonic external atoms must have a model and therefore also a minimal model. But then it also has an answer set, since each minimal model is an answer set.

If P is disjunction-free, it is Horn and thus must have a unique least model. □

Notice that this property fails if external atoms can be non-monotonic. Indeed, we can easily model default negation $not\, p(a)$ by an external atom $\&not[p](a)$; the HEX-program $p(a) \leftarrow \&not[p](a)$ amounts then to the ordinary program $p(a) \leftarrow not\, p(a)$, which has no answer set.

4.4 Modeling ASP Extensions by External Atoms

The formalism of answer-set programming has been enriched by various important extensions over the past years. By means of external atoms, these features can be generalized and expressed in terms of HEX-programs, reconstructing their semantics at an abstract level. In the following, we will pick out four such extensions and show how to model them with external atoms.

4.4.1 Programs with Aggregates

Extending ASP with special *aggregate atoms*, through which the sum, maximum, etc. of a set of numbers can be referenced, is an important issue which has been considered in several recent works (cf., e.g., [Faber et al., 2004]). A non-trivial and challenging problem in this context is giving a natural semantics for aggregates involving recursion. The recent proposal of a semantics by Faber *et al.* is an elegant solution of this problem. We show here how it can be easily captured by HEX-programs.

Let us recall the definition of an aggregate, which we already introduced in Subsection 2.2.3. An aggregate atom $a(Y,T)$ has the form $f\{S\} \prec T$, where f is an aggregate function (*sum*, *count*, *max*, etc.), $\prec \in \{=, <, \leq, >, \geq\}$, T is a term, and S is an expression $X{:}\vec{E}(\vec{X},\vec{Y},\vec{Z})$, where $\vec{X}$ and $\vec{Y}$ are lists of *local variables*, $\vec{Z}$ is a list of *global variables*, and $\vec{E}$ is a list of atoms whose variables are among $\vec{X},\vec{Y},\vec{Z}$.

For example, $\&count\{X : r(X,Z), s(Z,Y)\} \geq T$ is an aggregate atom which is intuitively true if, for given Y and T, at least T different values for X are such that the conjunction $r(X,Z), s(Z,Y)$ holds.

Given $a(Y,T) = f\{S\} \prec T$ as above, an interpretation I, and values y for Y and t for T, f is applied to the set $S(I,y)$ of all values x for X such that $I \models E(x,y,z)$ for some value z for Z. We then have $I \models a(y,t)$ (i.e., $I \models f\{X{:}E(X,y,Z)\} \prec t$) iff $f(S(I,y)) \prec t$.

Using the above notion of truth for $a(y,t)$, Faber et al. [2004] define answer sets of an ordinary program plus aggregates using the reduct fP^I.

We can model an aggregate atom $a(Y,T)$ by an external atom $\&a[Y](T)$ such that for any interpretation I and ground version $\&a[y](t)$ of it, $f_{\&a}(I,y,t) = 1$ iff $I \models a(y,t)$. Note that writing code for evaluating $f_{\&a}(I,y,t)$ is easy.

For any ordinary program P with aggregates, let $\&agg(P)$ be the HEX-program which results from P by replacing each aggregate atom $a(Y,T)$ with the respective external atom $\&a[Y](T)$. The following result can then be shown:

Theorem 4.4.1 *For any ordinary program P with aggregates, the answer sets of P and $\&agg(P)$ coincide.*

Proof. Let P be a program with aggregates. We need to show that I is an answer set of P iff it is an answer set of $\&agg(P)$.

($\Rightarrow$) If I is an answer set of P, it must be a minimal model of fP^I, i.e., each r in fP^I is satisfied by I. For each such rule r in fP^I, there must be a corresponding rule in $f(\&agg(P))^I$, which then must also satisfied by I, because by definition, for each aggregate atom a, it holds that $I \models a(y,t)$ iff $I \models \&a[y](t)$. Thus, I is also a model of $f(\&agg(P))^I$. Let J be a model of $f(\&agg(P))^I$ and $J \subset I$. Then, each rule r in $f(\&agg(P))^I$ must be satisfied by J and conversely also each corresponding rule in fP^I. But this contradicts our assumption of I being a minimal model of fP^I. Hence, I is also a minimal model of $f(\&agg(P))^I$ and therefore an answer set of $\&agg(P)$.

It is easy to see that the proof for the other direction is similar. □

4.4.2 dl-Programs

In Chapter 3, we defined answer sets of an ordinary non-disjunctive program P relative to a DL knowledge base L through a reduct sP_L^I, which extends the traditional reduct of Gelfond and Lifschitz [1991]. Assuming that each ground dl-atom $dl(c)$ is monotonic (i.e., $I \models dl(c)$ implies $I' \models dl(c)$, for $I \subseteq I'$; this is the predominant setting), sP_L^I treats negated dl-atoms like negated ordinary atoms. The resulting ground program sP_L^I has a least model, $LM(sP_L^I)$. Then, I is a *strong answer set* of (L,P) iff $I = LM(sP_L^I)$ holds.

We can simulate dl-atoms by external atoms in several ways. A simple one is to use external atoms $\&dl[\,](X)$ where $f_{\&dl}(I,c)=1$ iff $I \models_L dl(c)$. Let $\&dl_L(P)$ be the HEX-program obtained from a dl-program (L,P) by replacing each dl-atom $dl(X)$ with $\&dl[\,](X)$. We can then show:

Theorem 4.4.2 *Let $KB = (L,P)$ be any dl-program for which all ground dl-atoms are monotonic. Then, the strong answer sets of $KB = (L,P)$ and $\&dl_L(P)$ coincide.*

Proof. We follow the line of the proof of Theorem 3 by Faber et al. [2004]. First, we show that an interpretation I that is an answer set of $\&dl_L(P)$ must also be a strong answer set of $KB = (L,P)$. For readability reasons, we will denote $\&dl_L(P)$ by Π and assume both P and Π to be already grounded. We recall that $I \models_L dl(c)$ iff $I \models \&dl[\,](c)$

If I is an answer set of Π, it must be a minimal model of the reduct $f\Pi^I$. For each rule ρ in $f\Pi^I$, there must be a corresponding rule r in the strong reduct sP_L^I, which is obtained from ρ by removing all negative literals from ρ. Since $\rho \in f\Pi^I$, it must hold that all negative literals in I are true w.r.t. ρ and also w.r.t. J for all $J \subseteq I$. A rule r' that is in sP_L^I but has no corresponding rule in $f\Pi^I$ must have some literal p in $B^+(r')$ which is false w.r.t. I (otherwise the rule would not have been removed by the FLP-reduct) and also false w.r.t. J for all $J \subseteq I$. Thus, I is a model of sP_L^I and moreover a minimal model, because if a $J \subset I$ was a model of sP_L^I, it must also be a model of $f\Pi^I$, and hence I would not be a minimal model of $f\Pi^I$.

Next, we show that an interpretation I that is a strong answer set of $KB = (L,P)$ must also be an answer set of Π. Such an answer set is a minimal model of sP_L^I. For each rule $\rho \in fP^I$, there must be a corresponding rule $r \in sP_L^I$ such that $I \models B(r)$, with the negated literals removed from the body of ρ. Since $I \models H(r)$, we have $I \models H(\rho)$ and therefore $I \models fP^I$. To show that I is also a minimal model of fP^I, we assume $J \models fP^I$ with $J \subset I$. Again, for each rule $\rho \in fP^I$, the corresponding rule $r \in sP_L^I$ must also be satisfied by J. If a rule $r \in sP_L^I$ has no corresponding rule $\rho \in fP^I$, it must hold that $I \not\models B^+(r)$, because the negative part of the body is satisfied by I (otherwise the rule would have been in the strong reduct, too). But then, also $J \not\models B^+(r)$ and hence, $J \models r$. It follows that $J \models sP_L^I$, which contradicts that I is a strong answer set of KB. Thus, I is also a minimal model of $f\Pi^I$. □

Note that we can extend the strong answer-set semantics to disjunctive dl-programs by simply extending the embedding $\&dl_L(P)$ to disjunctive programs. This illustrates the use of HEX-programs as a framework for defining semantics.

The Description Logics plugin, that will be introduced in Subsection 5.3.1, supplies external atoms that entirely model the functionality of dl-atoms.

4.4.3 Programs with Monotone Cardinality Atoms

Marek et al. [2004] present an extension of ASP by *monotone cardinality atoms* (*mc-atoms*) $k\,X$, where X is a finite set of ground atoms and $k \geq 0$. Such an atom is true in an interpretation I, if $k \geq |X \cap I|$ holds. Note that an ordinary atom A amounts to $1\{A\}$. An *mca-program* is a set of rules

$$H \leftarrow B_1, \ldots, B_m, not\, B_{m+1}, \ldots, not\, B_n \tag{4.3}$$

where H and the B_i's are mc-atoms. Answer sets (stable models) for an mca-program P are interpretations I which are *derivable* models of an extended reduct P^I (in the sense of Gelfond and Lifschitz [1991]), which treats negated mc-atoms like negated ordinary atoms. Informally, a model of P^I is derivable, if it can be created from the empty set by iterative rule applications in which the heads of firing rules are nondeterministically satisfied.

We can embed any mca-program P into a HEX-program $\&mc(P)$ as follows. Each mc-atom $k\,X$ is modeled by an external atom $e(k\,X) = \&k\_X[\,]()$, where $f_{\&k\_X}(I) = 1$ iff $k \geq |X \cap I|$. In each rule of form (4.3), we replace H with a new atom t_H and all B_i with $e(B_i)$, and add the following rules (for $H = k\,\{A_1, \ldots, A_m\}$):

$$\begin{aligned} A_i \vee n\_A_i &\leftarrow t_H, && 1 \leq i \leq m, \\ &\leftarrow not\, e(H), t_H, \end{aligned}$$

where, globally, n_A is a new atom for each atom A. Informally, these rules simulate the occurrence of the mc-atom in the head. We can then show that for any finite mca-program P over atoms At, the answer sets of P and $\&mc(P)$ projected to At coincide.

As shown by Marek et al. [2004], ASP extensions similar to mca-programs can be modeled as mca-programs. Hence, these extensions can be similarly embedded into HEX-programs.

4.4.4 Agent Programs

Eiter et al. [1999b] describe logic-based *agent programs*, consisting of rules of the form

$$Op_0\alpha_0 \leftarrow \chi, [\neg]\, Op_1\alpha_1, \ldots, [\neg]\, Op_m\alpha_m,$$

governing an agent's behavior. The Op_i are *deontic modalities*, the α_i are *action atoms*, and χ is a *code-call condition*. The latter is a conjunction of (i) *code-call atoms* of the form $in(X, f(Y))$ resp. $notin(X, f(Y))$, which access the data structures of the internal agent state through API functions $f(Y)$ and test whether X is in the result, and (ii) *constraint atoms*. For example, the rule

$$Do\_dial(N) \leftarrow in(N, phone(P)), O\,call(P)$$

intuitively says that the agent should dial phone number N if she is obliged to call P.

A semantics of agent programs in terms of "reasonable status sets", which are certain sets of ground formulas $Op\,\alpha$, is defined by Eiter et al. [1999b]. They show that the answer

sets of a disjunction-free logic program P correspond naturally to the reasonable status sets of a straightforward agent program $AG(P)$. Conversely, code-call atoms as above can be modeled by external atoms $\&in_f[Y](X)$ resp. $\&notin_f[Y](X)$, and deontic modalities by different propositions and suitable rules. In this way, a class of agent programs can be embedded into HEX-programs as a host for evaluation.

4.5 Application Examples

In Section 3.8, we showed how to employ the semantics of dl-programs to problems related to the coupling of rules and ontologies. In this section, we present some ideas for the usage of HEX-programs for a more general class of different purposes, in which the joint availability of higher-order and external atoms is beneficial.

4.5.1 Semantic Web Applications

HEX-programs are well-suited as a convenient tool for a variety of tasks related to ontology languages and for Semantic-Web applications in general, since, in contrast to other approaches, they keep decidability but do not lack the possibility of exploiting nondeterminism, performing meta-reasoning, or encoding aggregates and sophisticated constructs through external atoms.

An interesting application scenario where several features of HEX-programs come into play is *ontology alignment*. Merging knowledge from different sources in the context of the Semantic Web is a very important task [Calvanese et al., 2001]. To avoid inconsistencies which arise in merging, it is important to diagnose the source of such inconsistencies and to propose a "repaired" version of the merged ontology. In general, given an entailment operator $\models$ and two theories T_1 and T_2, we want to find some theory $rep(T_1 \cup T_2)$ which, if possible, is consistent (with respect to $\models$). Usually, *rep* is defined according to some customized criterion, so that to save as much knowledge as possible from T_1 and T_2. Also, *rep* can be nondeterministic and admit more than one possible solution.

HEX-programs allow to define $\models$ according to a range of possibilities; in the same way, HEX-programs are a useful tool for modeling and customizing the *rep* operator. In order to perform ontology alignment, HEX-programs must be able to express tasks such as the following ones:

Importing external theories. This can be achieved, e.g., in the following way:

$$triple(X, Y, Z) \leftarrow \&rdf[uri](X, Y, Z);$$
$$triple(X, Y, Z) \leftarrow \&rdf[uri2](X, Y, Z);$$
$$proposition(P) \leftarrow triple(P, rdf{:}type, rdf{:}Statement).$$

We assume here to deal with RDF resp. RDF Schema theories [Brickley and Guha, 2004]. We take advantage of an external predicate $\&rdf$ intended to extract knowledge from a given URI (Uniform Resource Identifier), in form of a set of "reified" ternary assertions. Here, we clearly see the more general approach compared to our previously defined dl-programs: in a HEX-program, we are not bound to one specific ontology, but can refer to several sources in the same logic program. Moreover, provided that the respective external atoms are available, knowledge bases specified in various different formalisms, such as RDF and OWL can be merged.

Searching in the space of assertions. This task is required in order to choose non-deterministically which propositions have to be included in the merged theory and which not, with statements like

$$pick(P) \vee drop(P) \leftarrow proposition(P).$$

Thus, we specifically make use of the model-generation feature of the answer-set programming paradigm to be able to create a search space on top of terminological knowledge.

Translating and manipulating reified assertions. E.g., for choosing how to put RDF triples (possibly including OWL assertions) in an easier manipulatable and readable format, and for making selected propositions true, the following rules can be employed:

$$\begin{aligned}(X, Y, Z) \leftarrow\ & pick(P), triple(P, rdf{:}subject, X),\\ & triple(P, rdf{:}predicate, Y),\\ & triple(P, rdf{:}object, Z);\\ C(X) \leftarrow\ & (X, rdf{:}type, C).\\ owl{:}maxCardinality(C, R, N) \leftarrow\ & (X, rdf{:}type, owl{:}Restriction),\\ & (X, owl{:}onProperty, R),\\ & (X, owl{:}maxCardinality, N).\end{aligned}$$

Filtering propositions. The search space created by disjunctive rules can be narrowed by customizing criteria for selecting which propositions can be dropped and which cannot. For instance, a proposition cannot be dropped if it is an RDF Schema axiomatic triple:[1]

$$pick(P) \leftarrow axiomatic(P).$$

Defining ontology semantics. The operator $\models$ can be defined in terms of entailment rules and constraints expressed in the language itself, like in:

$$\begin{aligned}D(X) \leftarrow\ & (C, rdf{:}subClassOf, D), C(X);\\ \leftarrow\ & owl{:}maxCardinality(C, R, N), C(X),\\ & \&countr[R, X](M), M > N,\end{aligned}$$

where the external atom $\&countr[R, X](M)$ models the aggregate atom $\&count\{Y : R(X, Y)\} = M$, i.e., counting the role fillers Y for X in R. Moreover, semantics can be defined by means of external reasoners, using constraints like

$$\leftarrow \&inconsistent[pick],$$

where the external predicate $\&inconsistent$ establishes through an external reasoner whether the underlying theory is inconsistent w.r.t. a set of assertions.

[1] In a language enriched with *weak constraints*, we can also maximize the set of selected propositions using a constraint of form $:\sim drop(P)$.

4.5.2 Closed-World and Default Reasoning

In Sections 3.8.1 and 3.8.2, we showed how to exploit dl-programs to implement the non-monotonic inference principles of default reasoning and the closed-world assumption. Since suitable HEX-atoms can model the semantics of dl-atoms, these methods can be applied to HEX-programs accordingly. Furthermore, with HEX-programs we can take advantage of the fact that multiple external knowledge bases can be imported into a single program and hence apply the completion of a predicate (positively or negatively) only to specific parts of the imported information. Such a strategy corresponds to the widely claimed assertion that knowledge on the Semantic Web is not entirely open or closed, but rather contains "islands" of complete information, floating in a generally open world of reasoning.

4.6 Computation of HEX-programs

Our approach to designing and implementing a reasoner for HEX-programs was to use existing solvers as efficiently as possible by integrating them into a reasoning framework, instead of creating a model generator from scratch. We realized that existing implementations of ASP reasoners employ very sophisticated and effective methods, which can be reused for this novel semantics to a great extent. In this chapter we present principles and algorithms for solving HEX-programs.

The challenge of implementing a reasoner for HEX-programs lies in the interaction between external atoms and the ordinary part of a program. Due to the bidirectional flow of information represented by its input list, an external atom cannot be evaluated prior to the rest of the program. However, the existence of established and efficient reasoners for answer-set programs led us to the idea of splitting and rewriting the program such that an existing answer-set solver can be employed in turn with the external atoms' evaluation functions.

The reasoner for dl-programs, that was outlined in Chapter 3.9 already adopted a naive version of this method, trying to separate the program in a stratified and an unstratified part and thus speed up the computation. Here, we want to pursue a more sophisticated concept of processing the program. The basic idea is to identify as large as possible subprograms that can be solved "at once", i.e., by a single call to an external ASP reasoner. These calls are carried out alternatingly with the evaluation routines of the external atoms, which determine the locations where to split the program. In case of a stratified HEX-program, this strategy can be applied in a straightforward way, but as soon as external atoms occur in recursive definitions, other methods for evaluating such a "cycle" have to be applied.

In the following subsection, we will define suitable notions of dependency that enable us to view a HEX-program as a graph in order to identify subgraphs with certain properties that can be evaluated separately. This dependency information will be similar to the one that was developed for dl-programs, but also more general, since we have to account for disjunctive heads as well as higher-order syntax. Thus, we will repeat the notion of dependency in a logic program and enhance it where needed. Moreover, we will outline syntactic criteria for safety constraints of HEX-programs, guaranteeing a finite reasoning domain. In [Eiter et al., 2006f] we have presented these methods first and refined them later in [Eiter et al., 2006c].

Contrary to the treatment of external evaluations, the second feature of HEX-programs, the higher-order syntax, does not involve such sophisticated mechanisms. Our notion of higher-order can basically be regarded as syntactic sugar and translated to a first-order

logic program by moving the predicate inside the tuple of arguments. Thus, it is sufficient to carry out the following replacement prior to any program evaluation: Each ordinary atom of the form $p(\bar{X})$, where the predicate symbol p can also be a variable, is replaced by a first-order atom $a_n(p, \bar{X})$, where n is the arity of $\bar{X}$.

4.6.1 Dependency Information

Taking the dependency between heads and bodies into account is a common tool for devising an operational semantics for ordinary logic programs, e.g., by means of the notions of *stratification* or *local stratification* [Przymusinski, 1988], or through *modular stratification* [Ross, 1994] or *splitting sets* [Lifschitz and Turner, 1994]. Contrary to the traditional definition of dependency, like in [Apt et al., 1988], we have to consider that in HEX-programs, dependency between heads and bodies is not the only possible source of interaction between predicates. Moreover, allowing higher order atoms to have non-ground predicates, we use a modified notion of dependency between atoms, taking the entire atom and not only its predicate symbol into account. In particular we can have:[2]

Dependency between higher order atoms. For instance, $p(A)$ and $C(a)$ are strictly related. Intuitively, since C can unify with the constant symbol p, rules that define $C(a)$ may implicitly define the predicate p. This is not always the case: for instance, rules defining the atom $p(X)$ do not interact with rules defining $a(X)$, as well as $H(a, Y)$ does not interact with $H(b, Y)$.

Dependency through external atoms. External atoms can take predicate extensions as input: as such, external atoms may depend on their input predicates. This is the only setting where predicate names play a special role.

Disjunctive dependency. Atoms appearing in the same disjunctive head have a tight interaction, since they intuitively are a means for defining a common nondeterministic search space.

In the following we recall the traditional notion of stratification, supplementing the definition already given in Subsection 2.2.2. A program P is called *stratified*, if there is a partition

$$P = P_1 \dot{\cup} \ldots \dot{\cup} P_n$$

such that the following conditions hold for $i = 1, \ldots, n$:

1. if a relation symbol r occurs positively (i.e., is contained in a positive literal) in a rule in P_i, then its definition (i.e., the subset of P consisting of all rules where r occurs in the head) within $\bigcup_{j \leq i} P_j$.

2. if a relation symbol occurs negatively (i.e., is contained in a negative literal) in a rule in P_i, then its definition is contained within $\bigcup_{j < i} P_j$.

According to this definition, P is *stratified by* $P_1 \dot{\cup} \ldots \dot{\cup} P_n$ and each P_i is called a *stratum* of P.

Naturally, this definition is insufficient for HEX-programs, considering that not only external atoms depend from other atoms without occurring in any head, but also external atoms can have non-monotonic behavior and thus must be treated like weakly negated

[2] Of course, any such considerations must be carried out on the original program, prior to the translation from higher-order to first-order syntax as described before.

literals regarding stratification. In Section 4.3 we already noted that, while theoretically an external atom depends on the entire model(s) of the program, in practice we can restrict the input interpretation to specific relations.

Definition 4.6.1 *Let $\&g$ be an external predicate, $f_{\&g}$ its evaluation function, I an interpretation, and $X_1, \ldots, X_n$ its input list. Then $\&g$ is associated with a* type signature *$(t_1, \ldots, t_n)$, where each t_i is the* type *associated with X_i and can either be* `c` *or* `p`*. If t_i is* `c`*, then we assume that X_i is a constant, otherwise we assume that X_i is a predicate symbol. $f_{\&g}$ depends only on those atoms in I that have a predicate symbol p equal to some $X_i \in X_1, \ldots, X_n$ with $t_i = \mathtt{p}$.*

In order to be able to identify a reasonable dependency structure, In practice we do not allow to specify variables for input terms of type `p`. Otherwise the calculation of the part of the program that such an external atom depends on would quickly become very complex.

Definition 4.6.2 *Let P be a program and a, b atoms occurring in some rule of P. Then,* a depends positively *on b ($a \rightarrow_p b$), if one of the following conditions holds:*

1. *There is some rule $r \in P$ such that $a \in H(r)$ and $b \in B^+(r)$.*
   Example: $r_1 : p(X) \leftarrow q(X), r(X)$.
   Clearly, we have $p(X) \rightarrow_p q(X)$ and $p(X) \rightarrow_p r(X)$.

2. *There are some rules $r_1, r_2 \in P$ such that $a \in B(r_1)$ and $b \in H(r_2)$ and there exists a partial substitution θ of variables in a such that either $a\theta = b$ or $a = b\theta$. E.g., $H(a, Y)$ unifies with $p(a, X)$.*
   Example: $r_1 : p(X) \leftarrow q(X), r(X)$.
   $r_2 : q(Y) \leftarrow s(Y)$.
   Since $q(X)$ unifies with $q(Y)$, we have $q(X) \rightarrow_p q(Y)$.

3. *There is some rule $r \in P$ such that $a, b \in H(r)$. Note that this relation is symmetric.*
   Example: $r_1 : p(X) \vee q(X) \leftarrow r(X)$.
   From this we get $p(X) \rightarrow_p q(X)$ and $q(X) \rightarrow_p p(X)$.

Furthermore, a depends externally *on b ($a \rightarrow_e b$), if one of the following conditions holds:*

1. *a is an external predicate of form $\&g[\bar{X}](\bar{Y})$ with a type signature $(t_1, \ldots, t_n)$, where $\bar{X} = X_1, \ldots, X_n$, b is of form $p(\bar{Z})$, and, for some i, $X_i = p$ and $t_i = \mathtt{p}$.*
   Example: $r_1 : num(N) \leftarrow \&count[item](N)$.
   $r_2 : item(X) \leftarrow part(X)$.
   Here we have $\&count[item](N) \rightarrow_e item(X)$, if the input term *item* is of type `p` instead of merely denoting a constant string.

2. *there is some rule $r \in P$ with $a, b \in B(r)$ such that a is an external predicate of form $\&g[\bar{X}](\bar{Y})$ where $\bar{X} = X_1, \ldots, X_n$, and b is of form $p(\bar{Z})$, and $\bar{X} \cap \bar{Z} \neq \emptyset$.*
   Example: $r_1 : reached(X) \leftarrow \&reach[N, edge](X), startnode(N)$.
   This causes $\&reach[N, edge](X) \rightarrow_e startnode(N)$.

Moreover, a depends negatively *on b ($a \rightarrow_n b$), if there is some rule $r \in P$ such that either $a \in H(r)$ and $b \in B^-(r)$ or b is a non-monotonic external atom.*

We say that a depends on b, if $a \rightarrow b$, where $\rightarrow \; = \; \rightarrow_p \cup \rightarrow_e \cup \rightarrow_n$. The relation $\rightarrow^+$ denotes the transitive closure of $\rightarrow$. We say that a strictly depends *on b, or $a \mapsto b$, if $a \rightarrow^+ b$, but not $b \not\rightarrow^+ a$.*

These dependency relations let us construct a graph G_P, which we call the *dependency graph* of the corresponding program P.

Definition 4.6.3 *Let P be a* HEX*-program. A dependency graph G_P of P consists of the set V_P that contains all atoms in P (i.e., the vertices of G_P) and the set E_P of dependency relations contained in P according to Definition 4.6.2 (i.e., the edges of G_P).*

Note that this definition is based on a non-ground HEX-program P.

Example 4.6.1 Consider the following program, modeling the search for personal contacts that stem from a FOAF-ontology,[3] which is accessible by a URL.

$$
\begin{aligned}
&(1)\quad url(\text{“}http://www.kr.tuwien.ac.at/staff/roman/foaf.rdf\text{”}) \leftarrow;\\
&(2)\quad url(\text{“}http://www.mat.unical.it/\tilde{}ianni/foaf.rdf\text{”}) \leftarrow;\\
&(3)\quad \neg input(X) \vee \neg input(Y) \leftarrow url(X), url(Y), X \neq Y;\\
&(4)\quad input(X) \leftarrow not\ \neg input(X), url(X);\\
&(5)\quad triple(X,Y,Z) \leftarrow \&rdf[A](X,Y,Z), input(A);\\
&(6)\quad name(X,Y) \leftarrow triple(X, \text{“}http://xmlns.com/foaf/0.1/name\text{”}, Y);\\
&(7)\quad knows(X,Y) \leftarrow name(A,X), name(B,Y),\\
&\qquad\qquad\qquad\quad triple(A, \text{“}http://xmlns.com/foaf/0.1/knows\text{”}, B).
\end{aligned}
$$

The first two facts specify the URLs of the FOAF ontologies we want to query. Rules 3 and 4 ensure that each answer set will be based on a single URL only. Rule 5 extracts all triples from an RDF file specified by the extension of *input*. Rule 6 converts triples that assign names to individuals into the predicate *name*. Finally, the last rule traverses the RDF graph to construct the relation *knows*. Figure 4.2 shows the dependency graph of P.[4] ◇

We can now define several structural properties of HEX-programs.

Definition 4.6.4 *Let P be a* HEX*-program and $\rightarrow$ the relation defined above. We say that P is*

(i) nonrecursive, *if $\rightarrow$ is acyclic;*

(ii) stratified, *if there is no cycle in $\rightarrow$ containing some atom a and b such that $a \rightarrow_n b$;*

(iii) e-stratified, *if there is no cycle in $\rightarrow$ containing some atom a and b such that $a \rightarrow_e b$; and*

(iv) totally stratified, *if it is both stratified and e-stratified.*

For instance, the program in Example 4.6.1 is totally stratified because the only cycle is caused by the disjunction in Rule (3) and does not include negation.

Before we show how to process this graph in order to compute the answer sets of a HEX-program, we first need to answer the question how to tackle the potentially infinite domain of a HEX-program.

[3] “FOAF” stands for “Friend Of A Friend”, and is an RDF vocabulary to describe people and their relationships.

[4] Long constant names have been abbreviated for the sake of compactness.

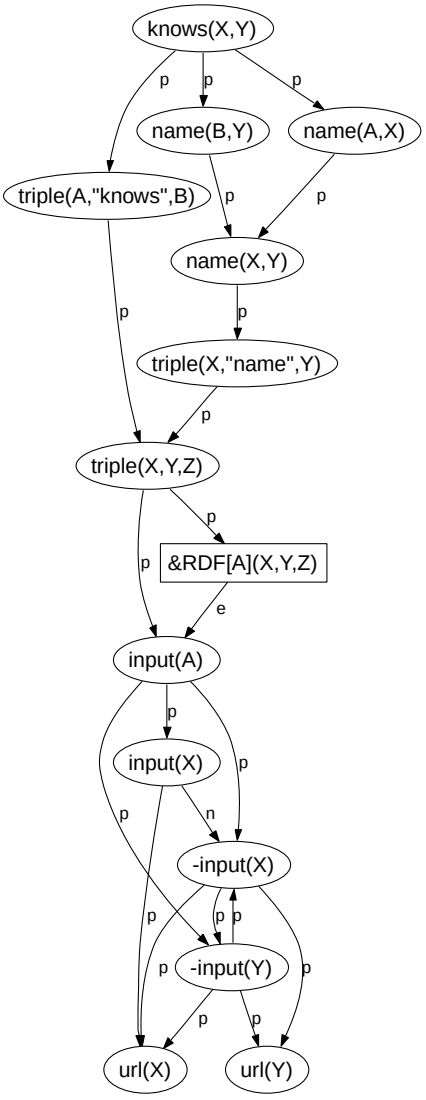


Figure 4.2: FOAF program graph.

4.6.2 Infinite Domains

Given a HEX-program P, its grounding $grnd(P)$ is infinite in general, and cannot be reduced straightforwardly to a finite portion since, given an external predicate $\&g$, the co-domain of $F_{\&g}$ is unknown and possibly infinite. It is thus important to impose two restrictions:

1. We assume that for a single specific ground input tuple, an external atom only returns a finite set of output tuples. Otherwise, finiteness could never be guaranteed, independently of the program's structure.

2. We restrict the usage of external predicates within a HEX-program in terms of stratification in order to bound the number of symbols to be taken into account to a finite number, whilst external knowledge in terms of new symbols can still be brought into a program.

In the following, we will describe the second condition in detail, beginning with the definition of *rule safety*.

Definition 4.6.5 *Given a rule r, the set of* safe variables *in r is the smallest set X of variables such that*

(i) *X appears in a positive ordinary atom in the body of r, or*

(ii) *X appears in the output list of an external atom $\&g[Y_1,\ldots,Y_n](X_1,\ldots,X_m)$ in the body of r and $Y_1,\ldots,Y_n$ are safe.*

A rule r is safe, *if each variable appearing in a negated atom and in any input list is safe, and variables appearing in $H(r)$ are safe.*

For instance, the rule $r : C(X) \leftarrow url(U), \&rdf[U](X, \text{“}rdfs{:}subClassOf\text{”}, C)$ is safe. Intuitively, this notion captures those rules for which input to external atoms can be determined by means of other atoms in the same rule. Given the extension of the predicate *url*, the number of relevant ground instances of r intuitively is finite and can be determined by repeated calls to $F_{\&rdf}$.

In some cases, safety is not enough for determining finiteness of the set of relevant symbols to be taken in account. This motivates the following stronger notion:

Definition 4.6.6 *Let r be a rule in P with external atoms $\&f_1[\bar{Y_1}](\bar{X_1}),\ldots\&f_n[\bar{Y_n}](\bar{X_n})$ in $B(r)$ and E be the set of all variables in $\bigcup_{i\leq n}\bar{Y_i}$. Moreover, let S be the set of atoms $b \in B^+(r)$ such that each atom $a \in H(r)$ strictly depends on b. Let V be the set of all variables that occur in the ordinary atoms in S and all variables in the output list of the external atoms in S. Let G be the set of all predicate symbols in $\bigcup_{i\leq n}\bar{Y_i}$. Then, r is strongly safe, iff (i) $E \subseteq V$ and (ii) each atom $a \in H(r)$ strictly depends on all $p \in G$.*

Informally, a rule is strongly safe, if its external atoms receive their input from a lower stratum of the program. This way, even if they occur in a cycle, their output cannot grow infinitely, since the size of their input is fixed “before entering" the cycle.

The rule r above is not strongly safe. Indeed, if some external URL invoked by means of *&rdf* contains some triple of form $(X, \text{“}rdfs{:}subClassOf\text{”}, url)$, the extension of the *url* predicate is potentially infinite. The rule

$$r' : instanceOf(C, X) \leftarrow url(U), \&rdf[U](X, \text{“}rdfs{:}subClassOf\text{”}, C)$$

is strongly safe, if $url(U)$ does not depend transitively on $instanceOf(C, X)$.

The strong safety condition is, anyway, only needed for rules which are involved in cycles of $\rightarrow$. In other settings, the ordinary safety restriction is enough. This leads to the following notion of a *domain-expansion safe* program. Let $grnd_U(P)$ be the ground program generated from P using only the set U of constants.

Definition 4.6.7 *A* HEX*-program P is* domain-expansion safe *iff each rule $r \in P$ is safe and each rule $r \in P$ containing some external atom $b \in B(r)$ is strongly safe.*

The following theorem states that we can effectively reduce the grounding of domain-expansion safe programs to a finite portion.

Theorem 4.6.1 *For any domain-expansion safe* HEX*-program P, there exists a finite set $D \subseteq \mathcal{C}$ such that $grnd_D(P)$ is equivalent to $grnd_{\mathcal{C}}(P)$ (i.e., has the same answer sets).*

Proof (sketch).

The proof proceeds by considering that, although the Herbrand universe of P is in principle infinite, only a finite set D of constants can be taken into account. From D, a

finite ground program, $grnd_D(P)$, can be used for computing answer sets. Provided that P is domain-expansion safe, it can be shown that $grnd_D(P)$ has the same answer sets as $grnd_{\mathcal{C}}(P)$.

A program that incrementally builds D and $grnd_D(P)$ can be sketched as follows: We update a set of *active* ordinary atoms A and a set R of ground rules (both of them initially empty) by means of a function $ins(r, A)$, which is repeatedly invoked over all rules $r \in P$ until A and R reach a fixed point. The function $ins(r, A)$ is such that, given a safe rule r and a set A of atoms, it returns the set of all ground versions of r such that each of its body atom a is either (i) such that $a \in A$ or (ii) if a is external, f_a is true. D is the final value of A, and $R = grnd_A(P)$. It can be shown that the above algorithm converges and $grnd_D(P) \subseteq grnd_{\mathcal{C}}(P)$. The program $grnd_{\mathcal{C}}(P)$ can be split into two modules: $N_1 = grnd_D(P)$ and $N_2 = grnd_{\mathcal{C}}(P) \setminus grnd_D(P)$. It holds that each answer set S of $grnd_{\mathcal{C}}(P)$ is such that $S = S_1 \cup S_2$, where $S_1 \in AS(N_1')$ and $S_2 \in AS(N_2)$. N_1' is a version of N_1 enriched with all the ground facts in $AS(N_2)$. Also, we can show that the only answer set of N_2 is the empty set. From this the proof follows. □

4.6.3 Splitting Algorithm

Similar to the methods described in Subsection 3.9.1, the principle of evaluation of a HEX-program relies on the theory of *splitting sets*. Intuitively, given a program P, a splitting set S is a set of ground atoms that induce a sub-program $grnd(P') \subset grnd(P)$ whose models $\mathcal{M} = \{M_1, \ldots, M_n\}$ can be evaluated separately. Then, an adequate *splitting theorem* shows how to plug the models M_i from $\mathcal{M}$ into a modified version of $P \setminus P'$ so that the overall models can be computed. Here, we use a modified notion of splitting set, accommodating non-ground programs and suited to our definition of dependency graph.

Definition 4.6.8 *A* global splitting set *for a* HEX*-program P is a set of atoms A appearing in P, such that whenever $a \in A$ and $a \rightarrow b$ for some atom b appearing in P, then also $b \in A$.*

Additionally, we define another type of splitting set:

Definition 4.6.9 *A* local splitting set *for a* HEX*-program P is a set of atoms $A \subseteq V_A$, such that for each atom $a \in A$ there is no atom $b \notin A$ such that $a \rightarrow b$ and $b \rightarrow^+ a$.*

Thus, contrary to a global splitting set, a local splitting set does not necessarily include the lowest layer of the program, but it never "breaks" a cycle.

Definition 4.6.10 *The* bottom *of P w.r.t. a set of atoms A is the set of rules $b_A(P) = \{r \in P \mid H(r) \cap A \neq \emptyset\}$.*

In other words, the bottom of P w.r.t. a set of atoms A includes all those rules that "define" A, i.e., whose head atoms occur in A.

Theorem 4.6.2 *Let P be a* HEX*-program and let A be a global splitting set for P. Then M is an answer set of P iff M is an answer set of P', where P' is the program obtained by removing $b_A(P)$ from P and adding the literals in N as facts to P and N is an answer set of $b_A(P)$.*

Proof. The proof is *mutatis mutandis* as the one of Theorem 3.9.1, replacing the strong reduct by the FLP-reduct. □

Apart from these definitions, we will next describe some preparations of the original HEX-program in order to be able to implement the idea of splitting sets by processing the program's dependency graph.

Preparing the Program

From the viewpoint of program evaluation, it turns out to be impractical to define the semantics of an external predicate by means of a Boolean function. Rather, we need a functional definition, that delivers a set of output tuples for a specific input tuple. To this end, we define $F_{\&g} : 2^{HB_P} \times D_1 \times \cdots \times D_n \to 2^{R_{\mathcal{C}}^m}$ with $F_{\&g}(I, y_1, \ldots, y_n) = \langle x_1, \ldots, x_m \rangle$ iff $f_{\&g}(I, y_1, \ldots, y_n, x_1, \ldots, x_m) = 1$, where $R_{\mathcal{C}}^m$ is the set of all tuples of arity m that can be built with symbols from $\mathcal{C}$ and $D_i = \mathcal{C}$ for $1 \leq i \leq n$. With this notion, we can compute the entire output of an external atom with a ground input list.

If the input list $y_1, \ldots, y_n$ is not ground in the original program, the restriction of domain-expansion safety for HEX-programs ensures that its values can be determined from the remaining rule body. However, since the basic idea of our algorithm is to split up the program along external atoms and evaluate the resulting parts in turn, we might lose exactly this context of an external atom. To illustrate this, let us first consider the following rule from the introduction:

$$reached(X) \leftarrow \&reach[edge, a](X).$$

where the first input term is of type `p`, i.e., *edge* is interpreted as a predicate name and a is of type `c`, i.e., a constant. Here, the input is completely determined w.r.t. to an interpretation and the external evaluation function can $F_{\&reach}$ can be evaluated with the input I, *edge*, a (assuming that I contains facts about *edge* and a that have been computed before). Now things are not so straightforward if variables are used in the input list, as in Rule (5) of Example 4.6.1:

$$triple(X, Y, Z) \leftarrow \&rdf[A](X, Y, Z), input(A).$$

Naturally, before we can evaluate this atom its input must be grounded. But since these ground values depend on a predicate which occurs in the same rule body, *input*, we need to add an auxiliary rule to the program:

$$\begin{aligned} triple(X, Y, Z) &\leftarrow \&rdf[A](X, Y, Z), input(A). \\ rdf_{inp}(A) &\leftarrow input(A). \end{aligned}$$

Of course, the head predicates of such rules must be uniquely associated with the respective external atom. Adding such rules will also affect G_P accordingly.

We can generalize these auxiliary rules by the following definition:

Definition 4.6.11 *Let P be a HEX-program and $\&g[\bar{Y}](\bar{X})$ be some external atom with input list $\bar{Y}$ occurring in a rule $r \in P$. Then, for each such atom, a rule $r_{inp}^{\&g}$ is composed as follows:*

- *The head $H(r_{inp}^{\&g})$ contains an atom $g_{inp}(\bar{Y})$ with a fresh predicate symbol g_{inp}.*
- *The body $B(r_{inp}^{\&g})$ of the auxiliary rule contains all body literals of r other than $\&g[\bar{Y}](\bar{X})$ that have at least one variable in its arguments (resp. in its output list if b is another external atom) that occurs also in $\bar{Y}$.*

For each external atom in P we can create such a rule. We denote the set of all such rules with P_{inp}.

The evaluation algorithm will ensure that the extension of g_{inp} is known before an external atom $\&g$ — i.e., the function $F_{\&g}(I, y_1, \ldots, y_n)$ — has to be evaluated. For atoms with an input list that was already ground in the original program, this extension coincides with the single such input tuple. For input lists with variables, the extension may contain zero or more ground tuples, each of which will be input to the external evaluation function. Note that there is no need to modify the original rule. The purpose of these auxiliary rules is basically to introduce an additional edge in the dependency graph, giving us the possibility to split the graph there and compute the input of the external atom before proceeding with the original rule itself.

We know now how to prepare the information flow from the program to the external atom. The only part of our interfacing machinery we are still missing before we can process the dependency graph is a way of importing the external evaluation result into the program. To this end we need another definition:

Definition 4.6.12 *Let P be a* HEX*-program We denote with P_{hex} the ordinary logic program having each external atom $\&g[\bar{Y}](\bar{X})$ in P replaced by $d_{\&g}(\bar{Y}, \bar{X})$ (we call this kind of atoms* replacement atoms*), where $d_{\&g}$ is a fresh predicate symbol.*

This replacement turns a HEX-program into an ordinary answer-set program that can be evaluated by an existing answer-set solver. Using the definitions above, the calls to such a solver and the external computations can be seamlessly interleaved. Next, we will lay out how to use the technique of splitting sets in a suitable algorithm.

Program Components

We define the concept of *external component*, which represents a part of the dependency graph including at least one external atom. Intuitively, an external component is the minimal local splitting set that contains one or more external atoms. We distinguish between different types of external components, each with a specific procedure of evaluation, i.e., computing its model(s) w.r.t. to a set of ground atoms I.

Before these are described, we introduce the notion of monotonicity, which helps us categorizing external components w.r.t. their algorithmic behaviour and selecting a proper evaluation method:

Definition 4.6.13 *A ground external atom $\&g$ is monotonic iff $I \models \&g$ implies $I' \models \&g$, for every $I \subseteq I' \subseteq HB_P$.*

The categories of external components in a HEX-program P we consider are:

- A single external atom $\&g$ that does not occur in any cycle in P. Its evaluation method returns for each tuple $\langle x_1, \ldots, x_m \rangle$ in $F_{\&g}(I, y_1, \ldots, y_n)$ a ground replacement atom $d_{\&g}(y_1, \ldots, y_n, x_1, \ldots, x_m)$ as result. The external atom in Figure 4.2, surrounded by a rectangular box, represents such a component.

- A strongly connected component C without any weakly negated atoms and only monotonic external atoms. A simple method for computing the (unique) model of such a component is given by the fixpoint operation of the operator $\Lambda : 2^{HB_P} \to 2^{HB_P}$, defined by $\Lambda(I) = M(P'_{hex} \cup D'(I)) \cap HB_P$, where:

  - P'_{hex} is an ordinary logic program as defined above, with $P' = b_C(P)$.
  - $D'(I)$ is the set of all facts $d_{\&g}(\bar{y}, \bar{c}) \leftarrow$ such that $I \models \&g[\bar{y}](\bar{c})$ for all external atoms $\&g$ in C; and

  - $M(P'_{hex} \cup D'(I))$ is the single answer set of $P'_{hex} \cup D'(I)$; since P'_{hex} is stratified, this answer set is guaranteed to exist and to be unique.

- A strongly connected component C with negative dependencies or nonmonotonic external atoms. In this case, we cannot rely on an iterative approach, but are forced to guess the value of each external atom beforehand and validate each guess w.r.t. the remaining atoms:

  - Construct P'_{hex} from $P' = b_C(P)$ as before and add for each replacement atom $d_{\&g}(\bar{y}, \bar{x})$ all rules
    $$d_{\&g}(\bar{y}, \bar{c}) \vee \neg d_{\&g}(\bar{y}, \bar{c}) \leftarrow \tag{4.4}$$
    such that $\&g[\bar{y}](\bar{c})$ is a ground instance of $\&g[\bar{y}](\bar{x})$. Intuitively, the rules (4.4) "guess" the truth values of the external atoms of C. Denote the resulting program by P'_{guess}.
  - Compute the answer sets $Ans = \{M_1, \ldots, M_n\}$ of P'_{guess}.
  - For each answer set $M \in Ans$ of P'_{guess}, test whether the original "guess" of the value of $d_{\&g}(\bar{y}, \bar{c})$ is compliant with $f_{\&g}$. That is, for each external atom a, check whether $M \models \&g[\bar{y}](\bar{c})$. If this condition does not hold, remove M from Ans.
  - Each remaining $M \in Ans$ is an answer set of P' iff M is a minimal model of fP'^{M}_{hex}.

Note that a cyclic subprogram must be domain-expansion safe in order to bound the number of symbols to be taken into account to a finite extent and avoid a potentially infinite ground program, while still allowing external atoms to bring in additional symbols to the program.

4.6.4 Evaluation Algorithm

The evaluation algorithm in (Figure 6) uses the following subroutines:

eval$(comp, I)$ Computes the models of an external component *comp* (which is of one of the types described above) for the interpretation I; I is added as a set of facts to each result model.

solve(P, I) Returns the answer sets of $P \cup A$, where P does not contain any external atom and A is the set of facts that corresponds to I.

Intuitively, the algorithm traverses the dependency graph from bottom to top, gradually pruning it while computing the respective models. Before entering the loop, the auxiliary rules from Definition 4.6.11 are added to the program in Step (a), followed by the identification of the dependency graph and the external components. $\mathcal{M}$ represents the set of the "current" models after each iteration, starting with the single set of facts F in the program. Step (b) singles out all external components that do not depend on any further atom or component, i.e., that are on the "bottom" of the dependency graph. Then, the algorithm loops through all current models $M \in \mathcal{M}$. For each such model those components are evaluated in Step (c) and can be removed from the list of external components that are left to be solved. In this innermost loop, the results of all components are accumulated in $\mathcal{M}'$ and then added to $\mathcal{M}''$ Moreover, Step (d) ensures that all rules of these components are removed from the program. $\mathcal{M}$ holds the result of all components over all current models. From the remaining part of the graph, Step (e) extracts the largest

Algorithm 4.1: Dependency graph evaluation.

Input: a HEX-program P
Result: a set of models $\mathcal{M}$

(a) $P \Leftarrow P \cup P_{inp}$
Determine the dependency graph G_P for P
Find all external components C_i of P and build $Comp = \{C_1, \ldots, C_n\}$
$T \Leftarrow Comp$
$\mathcal{M} \Leftarrow \{F\}$, where F is the set of all facts originally contained in P
while $P \neq \emptyset$ **do**
(b) $\quad \overline{T} \Leftarrow \{C \in T \mid \forall a \in C : \text{if } a \rightarrow b \in V_G \text{ then } b \in C\}$
$\quad \mathcal{M}'' \Leftarrow \emptyset$
$\quad$ **forall** $M \in \mathcal{M}$ **do**
$\quad\quad \mathcal{M}' \Leftarrow \{M\}$
$\quad\quad \mathcal{C} \Leftarrow \emptyset$
$\quad\quad$ **forall** $C \in \overline{T}$ **do**
(c) $\quad\quad\quad \mathcal{C} \Leftarrow \bigcup_{M' \in \mathcal{M}'} \mathsf{eval}(C, M')$
$\quad\quad\quad \mathcal{M}' \Leftarrow \mathcal{C}$
$\quad\quad$ **end**
$\quad\quad \mathcal{M}'' \Leftarrow \mathcal{M}'' \cup \mathcal{M}'$
$\quad$ **end**
$\quad Comp \Leftarrow Comp \setminus \overline{T}$
(d) $\quad P \Leftarrow P \setminus b_c(P)$ with $c \in \overline{T}$
$\quad \mathcal{M} \Leftarrow \mathcal{M}''$
$\quad \overline{C} \Leftarrow \{u \in V_G \mid u \in C \text{ or } u \rightarrow^+ c, c \in C \text{ for any } C \in Comp\}$
(e) $\quad P' \Leftarrow P_{hex} \setminus b_{\overline{C}}$
$\quad \mathcal{M} \Leftarrow \bigcup_{M \in \mathcal{M}} \mathsf{solve}(P', M)$
$\quad P \Leftarrow P \setminus P'$
$\quad$ remove all atoms from the graph that are not in $\overline{C}$
end
return $\mathcal{M}$

possible subprogram that does not depend on any remaining external component i.e., that is again on the "bottom" of the graph. After computing the models of this subprogram with respect to the current result, it is removed from the program resp. the dependency graph.

Basically, the iteration traverses the program graph by applying two different evaluation functions each turn. While *eval* solves minimal subprograms containing external atoms, *comp* solves maximal non-external subprograms.

Example 4.6.2 Let us exemplarily step through the algorithm with Example 4.6.1 as input program P. First, the graph G is constructed corresponding to Figure 4.2, but additionally including the dependencies that stem from the auxiliary rule for the external atom. Since P contains only a single external atom, the set *Comp* contains just one external component C, the *&rdf*-atom itself. Step (b) extracts those components of *Comp* that form a global splitting set, i.e., that do not depend on any atom not in the component. Clearly, this is not the case for C and hence, $\overline{T}$ is empty. This means that $\mathcal{M}''$ and therefore also $\mathcal{M}$ will be set to $\{F\}$. Step (e) constructs an auxiliary program P' by removing the bottom of $\overline{C}$, which contains each component that is still in *Comp* and every atom "above" it in

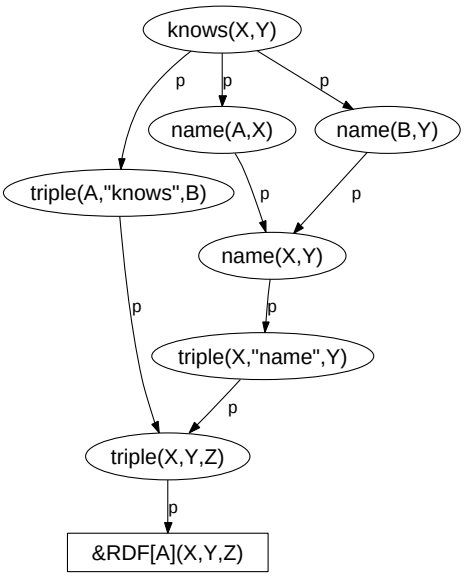


Figure 4.3: Pruned dependency graph.

the dependency graph. The following rules remain:

$$\neg input(X) \vee \neg input(Y) \leftarrow url(X), url(Y), X \neq Y;$$
$$input(X) \leftarrow not\ \neg input(X), url(X);$$
$$\&rdf_{inp}(A) \leftarrow input(A);$$

$solve(P', M)$ yields the answer sets of P', where the set of the original facts from P (the two URIs) is the single element of $\mathcal{M}$. P' is removed from P and $\bar{C}$ from the dependency graph (the resulting subgraph is shown in Figure 4.3). Continuing with (b), now the external component C is contained in $\bar{T}$, and therefore in Step (c) evaluated for each set in $\mathcal{M}$. After removing C from $Comp$ and propagating the result $\mathcal{M}'$ of the component to be the current result $\mathcal{M}$, $\overline{C}$ is empty and by Step (e) $P' = P_{hex}$, i.e., the remaining ordinary, stratified program, which is evaluated against each set in $\mathcal{M}$. Note that these sets now also contain the result of the external atom, represented as ground replacement atoms. At this point, P is empty and the algorithm terminates, having $\mathcal{M}$ as result. ◇

Example 4.6.3 This example combines the usage of the RDF-atom and another external atom that concatenates two strings. It takes advantage of the RSS-interface of the social bookmarking system del.icio.us, extracting the links that were associated with a specific keywords by the users of this service:

$$\#namespace(rdf, \text{“http://www.w3.org/1999/02/22-rdf-syntax-ns\#”})$$
$$\#namespace(rss, \text{“http://purl.org/rss/1.0/”})$$

$$tag(\text{“}turing\text{”}).$$
$$url(X) \leftarrow \&concat[\text{“}http://del.icio.us/rss/tag/\text{”}, W](X), tag(W).$$
$$Y(X, Z) \leftarrow \&rdf[A](X, Y, Z), url(A).$$
$$link(X) \leftarrow \text{“}rdf{:}type\text{”}(X, \text{“}rss{:}item\text{”}).$$

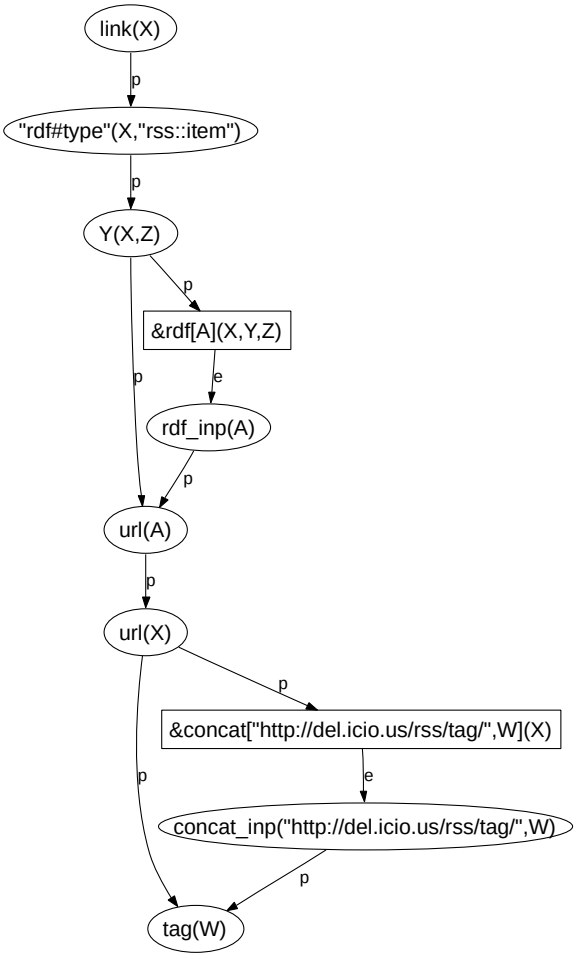


Figure 4.4: del.icio.us program graph.

After the specification of two namespaces (they simply act as macros and will be introduced in Subsection 5.2.3), the tag *turing* is defined. By means of the &*concat* atom, the appropriate URL is built, in order to retrieve the RSS-item of del.icio.us regarding that tag. The triples of this RSS-item are imported by means of the &*rdf* atom, using higher-order syntax. Eventually, the actual links are filtered out from the entire RDF-graph of the query result.

The dependency graph of this program is depicted in Figure 4.4. Here we also included the auxiliary rules that compute the input of both external atoms atom, $concat_{inp}$ and rdf_{inp}.

In the first iteration of our algorithm, no external atom can be evaluated and the program resulting from the pruned subgraph only contains the rule computing the extension of $concat_{inp}$, which amounts to $\{concat_{inp}(\text{“}http://del.icio.us/rss/tag/\text{”}, \text{“}turing\text{”})\}$. In the second iteration, the &*concat* atom can be evaluated with this input, concatenating both strings. The program that is created in Step (e) of the second iteration is as follows:

$$url(X) \leftarrow \&concat[\text{“}http://del.icio.us/rss/tag/\text{”}, W](X), tag(W);$$
$$rdf_{inp}(A) \leftarrow url(A).$$

The single model in $\mathcal{M}$ and hence the input to $\mathsf{solve}(P', M)$ already contains the result of the $\&concat$ atom, i.e., :

$$\&concat[\text{“}http://del.icio.us/rss/tag/\text{”}, \text{“}turing\text{”}](\text{“}http://del.icio.us/rss/tag/turing\text{”})$$

Thus, at the end of the second iteration, the extension of *url* and hence also of rdf_{inp} is fully determined.

In the third iteration, the $\&rdf$ atom can now be evaluated on this URI. The remaining program does not contain any external atoms any more and traverses the RDF graph and singles out the links contained within.

◇

We obtain the following property:

Theorem 4.6.3 *Let P be a* HEX*-program and $\mathcal{M}$ the output of the evaluation algorithm from Figure 6. Then, M is an answer set of P iff $M \in \mathcal{M}$.*

Proof (sketch). The given algorithm is actually a repeated application of the Splitting Set Theorem 4.6.2 for HEX-programs. Basically, the theorem says that if U is a splitting set for a program P, then, a set A is an answer set of program P iff A is an answer set of $P' = (P \setminus b_U) \cup B$ where B contains the facts corresponding to some answer set of b_U.

Given the current value of P, Step (b) of the algorithm finds splitting sets corresponding to external components of P. The splitting set theorem is applied by computing the answer sets of the bottoms of each of these components. If one of the components is found to be inconsistent, then the entire program must be inconsistent and no answer set exists. Step (e) again applies the Splitting Set Theorem on the remaining program. In this case, the splitting set which is searched for does not contain external atoms. After each iteration of the algorithm, the set of final answer sets is updated, while P is reduced. Finally, all answer sets of P are left. □

4.7 Complexity

It appears that higher-order atoms do not add complexity compared to ordinary atoms. Indeed, for finite $\mathcal{C}$, the grounding of an arbitrary HEX-program P is, like for an ordinary program, at most exponential in the size of P and $\mathcal{C}$. Since HEX-programs with higher-order atoms subsume ordinary programs, we obtain by well-known complexity results for ordinary programs [Dantsin et al., 2001] the following result. Recall that NEXP denotes nondeterministic exponential time, and that for complexity classes C and D, C^D denotes complexity in C with an oracle for a problem in D.

Theorem 4.7.1 *Deciding whether a given* HEX*-program P without external atoms has some answer set is* $\mathsf{NEXP}^{\mathsf{NP}}$*-complete in general, and* NEXP*-complete if P is disjunction-free.*

The proof is trivial, since such a program coincides with an ordinary answer-set program.

Classes of programs with lower complexity can be identified under syntactic restrictions, e.g., on predicate arities. Furthermore, if from the customary ASP perspective, P is fixed except for ground facts representing ad-hoc input, the complexity exponentially drops to NP^{NP} resp. NP.

On the other hand, external atoms clearly may be a source of complexity, and without further assumptions even incur undecidability. Viewing the function $f_{\&g}$ associated with

an external predicate $\&g \in \mathcal{G}$ as an oracle with complexity in C, we have the following result:

Theorem 4.7.2 *Let P be a* HEX*-program, and suppose that for every $\&g \in \mathcal{G}$ the function $f_{\&g}$ has complexity in C. Then, deciding whether P has some answer set is in $\mathsf{NEXP}^{\mathsf{NP}^C}$, and is in NEXP^C if P is disjunction-free.*

Again, this result is trivially obtained by using oracles to compute external atoms in a single step, this extending the well-known results of $\mathsf{NEXP}^{\mathsf{NP}}$ resp. NEXP for answer-set existence by a C-oracle.

However, there is no complexity increase by external atoms under the following condition on the cardinality of $\mathcal{C}$:

Theorem 4.7.3 *Let P be a* HEX*-program. Suppose that for every $\&g \in \mathcal{G}$, the function $f_{\&g}$ is decidable in exponential time in $|\mathcal{C}|$. Then, deciding whether P has some answer set is $\mathsf{NEXP}^{\mathsf{NP}}$-complete, and* NEXP*-complete if P is disjunction-free.*

Proof. The NEXP complexity of deciding answer-set existence stems from the grounding of an answer-set program. If the program is already grounded, the function $f_{\&g}$ is decidable in polynomial time and hence is not needed as an oracle any more. □

Informally, the reason is that a possibly exponential-size grounding compensates the exponentiality of external atoms, whose evaluation then becomes polynomial in the size of $grnd(P)$. The hypothesis of Theorem 4.7.3 applies to external atoms modeling aggregate atoms and, under small adjustments, to dl-atoms, if $\models$ is decidable in exponential time. Some complexity results by Faber et al. [2004] on ASP with aggregates and by Eiter et al. [2004a] on interfacing logic programs with the description logic $\mathcal{SHIF}(\mathbf{D})$ therefore follow easily from Theorems 4.4.1, 4.4.2, and 4.7.3.

4.8 An Extension: Weak Constraints

In Section 2.2.3 we have described the concept of weak constraints implemented by the answer-set solver DLV. Weak constraints are a powerful tool for specifying optimization problems by means of answer-set programs. Therefore we saw it as a crucial feature of a solver for HEX-programs to support this language extension. It was already described that the computation procedure for HEX-programs relies on an existing answer set solver as a "black box", which is subsequently called to solve parts of the input program. As a consequence, we cannot use the optimization features of the external solver, but had to find a procedure to apply the semantics of weak constraints to the entire HEX-program. For simplicity reasons we adopted a naive method of applying after the computation of the answer sets instead of considering them already in the stepwise model generation.

For each weak constraint W in a HEX-program P

$$:\sim Conj\ [W : L]$$

we add an auxiliary rule to P prior to the evaluation routine:

$$W^{aux}(W, L) \leftarrow Conj$$

Evidently, for each satisfied body $Conj$ of such a rule, a fact $W_i^{aux}(W, L)$ is added to the respective model with instantiated values for W and L. After completion of the model

generation procedure, the costs of each model M_j of P are added up for each specified level L:

$$\mathsf{cost}_{j,L} = \sum_{W^{aux}(W,L) \in M_j} W$$

Now we can order the models according to the semantics of weak constraint described in Section 2.2.3.

Example 4.8.1 Recall Example 3.2.3, where we used a dl-program for computing the possibilities of connecting new nodes to an existing network. In this example, we avoided to connect the new nodes to those, which belong to the concept *HighTrafficNode*. Since membership to this concept depends on the number of connections, we demonstrated the effect of a feedback from P to L.

Instead of ruling out nodes in *HighTrafficNode* from future connections completely, we can use weak constraints to minimize the number of such overloaded nodes. The following program P_N^{HEX} is given in dl-program syntax — we anticipate here the development of a DL-interface for our HEX-framework, that is capable of parsing dl-programs and will be described in Subsection 5.3.1.

(1) $newnode(add_1)$;
(2) $newnode(add_2)$;
(3) $overloaded(X) \leftarrow DL[wired \uplus connect; HighTrafficNode](X), DL[Node](X)$;
(4) $connect(X,Y) \leftarrow newnode(X), DL[Node](Y), not\ DL[HighTrafficNode](X),$
$not\ excl(X,Y)$;
(5) $excl(X,Y) \leftarrow connect(X,Z), DL[Node](Y), Y \neq Z$;
(6) $excl(X,Y) \leftarrow connect(Z,Y), newnode(Z), newnode(X), Z \neq X$;
(7) $excl(add_1, n_4)$;
(8) $:\sim overloaded(X).\ [1:1]$.

The difference to Example 3.2.3 is that in Rule (4) we do not consider the predicate *overloaded* anymore when building the connections. Instead, we only avoid nodes that are originally in *HighTrafficNode* and create all other possible connections, though applying a penalty to the model for each overloaded node by the weak constraint (8). This results in the following answer sets:

$\{overloaded(n_2);\ connect(add_1, n_5);\ connect(add_2, n_4);\ \ldots\}$ Cost: 1:1

$\{overloaded(n_2);\ connect(add_1, n_1);\ connect(add_2, n_4);\ \ldots\}$ Cost: 1:1

$\{overloaded(n_2);\ connect(add_1, n_1);\ connect(add_2, n_5);\ \ldots\}$ Cost: 1:1

$\{overloaded(n_2);\ connect(add_1, n_5);\ connect(add_2, n_1);\ \ldots\}$ Cost: 1:1

$\{overloaded(n_2);\ overloaded(n_3);$
$connect(add_1, n_3);\ connect(add_2, n_4);\ \ldots\}$ Cost: 2:1

$\{overloaded(n_2);\ overloaded(n_3);$
$connect(add_1, n_5);\ connect(add_2, n_3);\ \ldots\}$ Cost: 2:1

$\{overloaded(n_2);\ overloaded(n_3);$
$connect(add_1, n_1);\ connect(add_2, n_3);\ \ldots\}$ Cost: 2:1

$\{overloaded(n_2);\ overloaded(n_3);$
$connect(add_1, n_3);\ connect(add_2, n_5);\ \ldots\}$ Cost: 2:1

$\{overloaded(n_2);\ overloaded(n_3);$
$connect(add_1, n_3);\ connect(add_2, n_1);\ \ldots\}$ Cost: 2:1

The first four models with the lowest costs correspond to the result we obtained in Example 3.4.2. Moreover, we also computed "worse" models with more overloaded nodes. ◇

In Section 6.1 we describe another example combining external evaluations with an optimization problem.

Chapter 5

Implementation of a HEX-Reasoner

In this chapter, we present the prototype implementation of a solver for HEX-programs, called dlvhex, which is publicly available for download. dlvhex implements fully the HEX-program syntax and semantics including hard and weak constraints. It conservatively extends `DLV`, such that for any ordinary answer-set program dlvhex behaves equally to `DLV`.

5.1 Architecture

The principal design goals of dlvhex were:

- Reusing and integrating existing reasoning applications instead of writing them from scratch.
- Following a modular approach regarding the integration of external atom evaluation.
- Using an efficient programming language.
- Exploiting object-oriented principles for maximum maintainability and extensibility.

The decision of using an existing answer-set solver was already the foundation of the evaluation algorithm presented in Subsection 4.6.4. Thus, dlvhex can be seen as a reasoner framework rather than as a stand-alone inference engine.

5.1.1 Code Layout

Figure 5.1 depicts the overall architecture of dlvhex. We split the input program P into its *intensional database* (IDB), which is constituted by the rules of P, and its *extensional database* (EDB), containing the facts of P. This distinction corresponds to the view of an answer-set program as the union of a *problem specification* (IDB) and the *instance data* (EDB). Initially, the entire program is parsed and transformed into proper data-structures, which are then passed to an object of the class `DependencyGraph`. This class maintains an object of the class `GraphBuilder`, which builds the dependency graph according to the definitions of Subsection 4.6.1. The class `ComponentFinder` identifies the components of P following the component types which were presented in Subsection 4.6.3. The `SafetyChecker` class then verifies whether the safety conditions (Subsection 4.6.2) hold. Next, the dependency graph is passed to the `GraphProcessor`, which implements the main algorithm as shown in Subsection 4.6.4. At this point, the IDB is introduced in the evaluation. The result is fed into an object of the class `ResultContainer`, which takes care of filtering and applying weak constraints to the answer sets of P.

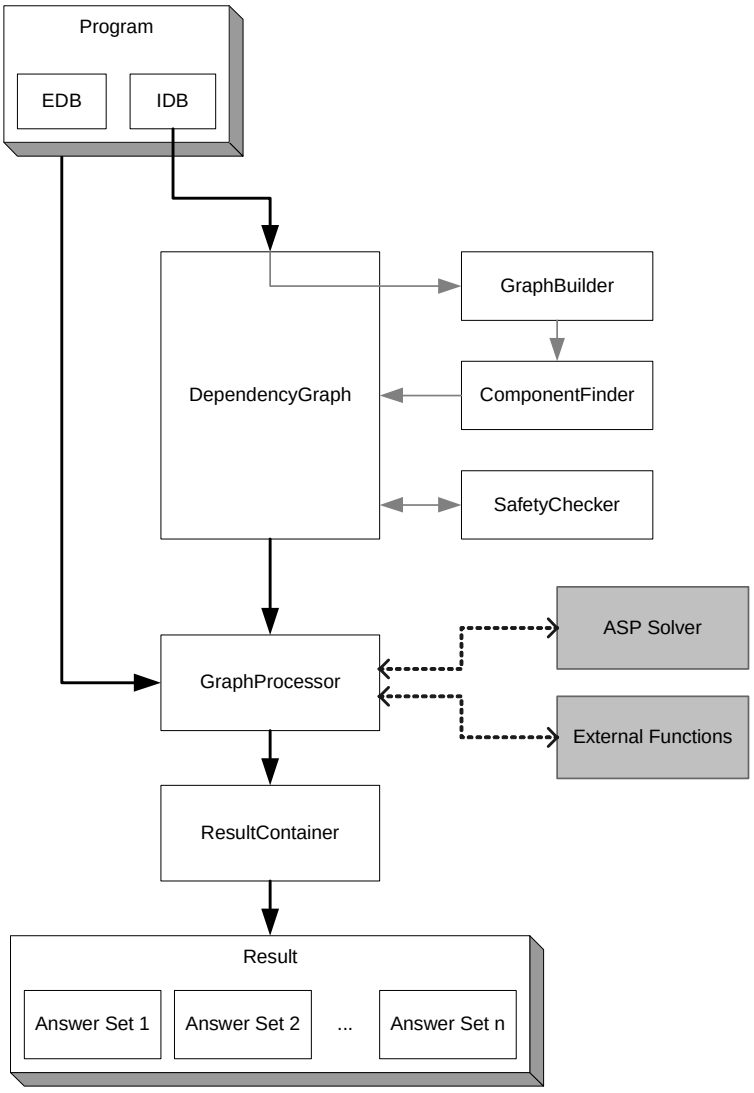


Figure 5.1: dlvhex Architecture.

In the following, we will outline the function and implementation of some of the major modules of dlvhex.

DependencyGraph

The class `DependencyGraph` is responsible for constructing the dependency graph G_P from the rules of the input program P and maintaining it. This class uses modules that are implemented according to the Strategy Design Pattern (cf. [Gamma et al., 1995]), which basically consists of decoupling an algorithm from its host, and encapsulating the algorithm into a separate class. One of these algorithms is the actual construction of the dependency relations from the rules, which is encoded in the strategy class `GraphBuilder`. Given the graph, finding the program components is another task that is handed to a strategy class, namely `ComponentFinder`. Its main purpose is to find strongly connected components, which — in the current version of dlvhex— is carried out by the Boost Graph Library [Siek et al., 2002]. Having identified the SCCs, the `DependencyGraph` eventually creates the

corresponding objects for each program component.

The class `SafetyChecker`, which again is implemented as a strategy pattern, examines the dependency graph for violation of weak safety and domain-expansion safety.

GraphProcessor

The principal algorithm for computing the answer sets of a HEX-program is implemented in the class `GraphProcessor`. It is initialized with the EDB of the input program P and the dependency graph G_P and traverses the graph until all nodes are visited or an inconsistency is found. As laid out in the evaluation algorithm, the `GraphProcessor` works with the previously identified program components and calls their evaluation routines. Subgraphs without any external atoms are also assigned to a component, whose evaluation procedure simply calls the external answer-set solver. The `GraphProcessor` has to take care of combining the results of various subgraphs while preserving minimality of the intermediate models.

ResultContainer

The class `ResultContainer` serves as a repository for answer sets and provides methods for filtering out specific literals, such as auxiliary atoms or user-specified filters. It also takes care or ordering the result sets if weak constraints were used. The `ResultContainer` implements different methods of displaying the final result, using the Builder Design Pattern. At its current state, dlvhex can display the result in textual representation (closely related to the output of `DLV`) or in RuleML syntax [Boley et al., 2001].

5.1.2 Plugin Integration

The integration of external sources of computation is the most distinguished feature of dlvhex. We pursued an approach that guaranteed maximum flexibility and extensibility while trying to keep the interface lean and making external atoms as easy to implement as possible by the user. To this end, we defined a so-called *Plugin*, which is represented by a shared library that is linked to dlvhex at runtime and which provides one or more external atoms and their evaluation functions. Upon invocation, dlvhex looks for such plugin-libraries in specific places in the local filesystem and registers the found atoms.

The plugin author is responsible for the following specifications:

- the evaluation function of each external atom,
- the type of each input term of each external atom, (cf. Def. 4.6.1)
- the output arity, and
- the predicate name of each atom.

Rewriters

A powerful feature of the plugin mechanism is the possibility for plugin authors to integrate a custom rewriter for HEX-programs. The main routine of dlvhex calls each plugin's rewriter prior to its own parsing routine and passes the entire input program to it. The rewriter is responsible for returning a valid HEX-program. This way, abbreviations for external atoms can be introduced, that don't comply with the original HEX-Syntax, but are more convenient in the respective case. Of course it is also possible for a plugin to provide only

a rewriter, but no external atoms. An example for such a pure rewriter is presented in Section 6.2, where dlvhex is used as a reasoner for the RDF query language SPARQL.

Another peculiar property of the plugin architecture is the possibility to pass dlvhex command line parameters to plugins. Any parameters that are unknown to dlvhex itself are passed on to the registered plugins. Thus, a plugin can for instance display its own help message.

5.2 Using dlvhex

The initial target platform for dlvhex was Unix/Linux. Since the Apple operating system MacOS X is Unix-based, our reasoner complies and runs also on these systems flawlessly. This Section describes how to install and use dlvhex and how to write well-formed HEX-programs.

5.2.1 Installation

For the user's convenience, we provide the dlvhex source code as well as its binary, compiled for Linux i386 platforms as well as MacOS X. Unfortunately, porting the application to Windows proved to be too complicated regarding our development manpower.

All downloads are available at the following location:

http://www.kr.tuwien.ac.at/research/dlvhex/

Compiled Binary

In principle, the binary is ready to use. However, if one wants to use any plugins (they are also available as binaries at the specified URL), one has to take care of either specifying the plugins' path to dlvhex on the command-line via the `-p` switch (see also Subsection 5.2.2) or put the plugins in the folder `$HOME/.dlvhex/plugins/`.

Source Package

dlvhex is maintained using the GNU autotool framework, which does not only ease the development process, but also disburdens the user from dealing with the entire compile and link procedure. If the user wishes to develop her own plugin(s), downloading and installing the sources is absolutely necessary, since they provide necessary header and package configuration files.

After unpacking the source package, it usually is sufficient to subsequently call

```
./configure
make
make install
```

The configure script accepts the usual switches, e.g., if the user is not root and wants to install the package in a specific directory, she would specify

```
./configure --prefix=/path/to/dir
```

The configure script examines whether the system has all necessary conditions for building and running dlvhex. It has two requirements that are nonstandard:

- the Boost library, available as packages for all common Linux distributions, and
- the answer-set solver `DLV`, which can be downloaded at http://www.dbai.tuwien.ac.at/proj/dlv/.

5.2.2 Calling dlvhex on the Command Line

If the installation process was successful, dlvhex should be executable from the command line. The usual way to invoke it is as follows:

`dlvhex` *filename* [*filename2* ...]

Here, *filename* is the HEX-program. Optionally, more than one files can be specified, whose contents will be merged to a single program.

dlvhex Options

The following options are supported:

'`--`'
: Parse from `stdin`. This way, a program can be piped to dlvhex, such as `cat test.hex | dlvhex --`.

'`-s`'
'`--silent`'
: Suppresses all informational output and displays only the actual result.

'`-v`'
'`--verbose`'
: Dumps various intermediate processing information and writes the program dependency graph to filename.dot.

'`-p` *dir*'
'`--plugindir=` *dir*'
: Specify an additional directory where to look for plugin-libraries (additionally to the installation plugin-dir and `$HOME/.dlvhex/plugins`).

'`-f` *foo*, *bar*, ...'
'`--filter=`*foo*, *bar*, ...'
: Specify predicate names that are to be kept in the resulting answer sets. All other atoms are removed from the output.

'`--firstorder`'
: Do not allow higher order syntax.

'`--ruleml`'
: Display result in RuleML format (v0.9).

'`--noeval`'
: Just parse the program and build the dependency graph, but do not evaluate it (only useful in combination with `--verbose`).

'`--allmodels`'
: Display all models in case of using weak constraints and not only the optimal one(s).

5.2.3 Writing HEX-Programs

The general syntax of HEX-programs has already been presented in Section 4.2. Here, we will go into details concerning the actual textual representation of a HEX-program that can be solved by dlvhex, which largely corresponds to the syntax of disjunctive datalog for DLV. Thus, we will only briefly recall the basics of DLV-programs here and point out the differences to dlvhex input programs.

Terms, Atoms, and Rules

The smallest entity in a HEX-program is a *Term*. A term can be one of the following:

Constant: A constant start with a lowercase letter and may contain letters, digits and underscores.
Examples: `p`, `person`, `42`, `an_Animal`
A constant can also be a quoted arbitrary string.
Examples: `"ABC"`, `"http://example.org"`, `"any(!)thing..."`

Variable: A variable must begin with an uppercase letter and may again contain letters, underscores and digits.
Examples: `X`, `Animal`, `C_a1`

Anonymous variable: An anonymous variable is a special variable that can be seen as unique variable in its context (i.e., rule or constraint). Thus, the value of such a term will simply be ignored. An anonymous variable is denoted by a single underscore: `_`

An Atom is written in the expected way, specifying a predicate symbol followed by a list of terms in parentheses. Naturally, also propositional atoms are admitted:

```
pair(X,Y), light_on, -p(Q,_,r), X(y,Z)
```

dlvhex, like `DLV`, supports true negation, written as a `-` and used for the third atom above, and negation-as-failure. The last atom in the list of examples above represents a higher-order atom, having a variable predicate. Such a predicate is treated like any other variable term.

Another feature not available in `DLV` is the possibility to state atoms in tuple syntax:

```
(X,a,Y), ("rdf:type",X,"rss:item")
```

This is simply an alternative way of writing `X(a,Y)`, `"rdf:type"(X,"rss:item")`. Both versions are semantically equal.

Negation-as-failure, or weak negation, is specified by the symbol `not` preceding the (possibly classically negated) atom, forming a literal:

```
not (X,a,Y), not -flies(tweety)
```

Facts are ground atoms that correspond to a rule without body, i.e., explicitly true assertions. The set of facts in a program is often denoted as the *extensional database* (EDB) of the program:

```
-visible(ground).
edge(1,2).
```

As opposed to the EDB, the rules of a program are called the *intensional database*, encoding implicit knowledge. In the answer-set programming paradigm, the EDB is seen as a concrete problem instance, while the rules of a program represent the general specification of the problem's solution(s). The semantics of a rule in answer-set programming was made sufficiently clear in Subsection 4.3. A rule's head can contain a disjunction of atoms, its body a conjunction of literals:

```
connect(X,Y) v -connect(X,Y) :- node(X), node(Y), not unreachable(X,Y).
```

A rule without head, a constraint, is specified accordingly:

```
:- visit(X), visit(Y), X != Y.
:- arc(X,Y), color(X,C), color(Y,C).
```

A special case of facts are disjunctive rules without body, i.e., disjunctive facts:

```
color(green) v color(blue) v color(red).
```

Weak constraints are stated with a ":∼" and the (optional) specification of level and weight:

```
:∼ member(X,P), member(Y,P), X != Y, married(X,Y). [1:2]
```

The default value for both weight and level is 1; thus, the following weight constraints are equivalent:

```
:∼ salesman(X), travels(X,Y,Z).
:∼ salesman(X), travels(X,Y,Z). [1:]
:∼ salesman(X), travels(X,Y,Z). [:1]
```

Builtin Predicates dlvhex supports all comparative predicates that are also available in DLV, <, >, <=, >=, =, !=. They can be used in normal prefix or infix notation:

```
in_range(X,A,B) :- X >= A, <(X,B).
```

Currently *not* supported by dlvhex are the arithmetic predicates #int, #succ, +, * and facts over a fixed integer range, such as weekday(1..7).

External Atoms An external atom can occur (possibly weakly negated) in the body of a rule:

```
name(X,Y) :- mem(X), &dlDR[U,a,b,c,d,"name"](X,Y), url(U).
```

It is also allowed to use external atoms in constraints.

Aggregate Predicates Aggregate predicates have been introduced in Subsection 2.2.3. They allow to express properties over a set of elements and can be used in rule bodies and constraints:

```
   q :- 0 <= #count{X,Y : a(X,Z,k),b(1,Z,Y)} <= 3.
q(Z) :- 2 < #sum{V : d(V,Z)}, c(Z).
```

An aggregate predicate consists of one or more guards and the aggregate function. dlvhex supports all aggregate functions that are currently available in DLV: #count, #sum, #times, #min, and #max. For more information about the specification of aggregates, we refer to the manual of DLV.[1]

[1] http://www.dbai.tuwien.ac.at/proj/dlv/man/

Rule Safety Intuitively, safety ensures that the grounding of the program does not "explode". A (possibly disjunctive) fact is safe if does not contain any variables. A rule is safe, if each variable occurring

- in its head,
- in a negation-as-failure literal,
- in a built-in comparative predicate,
- as a guard or in the symbolic set of an aggregate

also occurs in a non-comparative positive literal in the body of the same rule. This also applies to constraints. In the case of external atoms, an extended notion of safety applies, which has been explained in Subsection 4.6.2.

Namespaces

Aiming at various reasoning tasks in the context of the Semantic Web, we introduced a syntactic feature, that can be used as a string macro. A *namespace specification* relates a shortcut with a string, such that the shortcut can be used as an abbreviation. A namespace definition starts with a `#` and has to occur on a separate line, *not* ending in a period (to distinguish it as a preprocessor-directive from a rule):

```
#namespace("foaf","http://xmlns.com/foaf/0.1/")
```

The string `foaf` can now be used as a placeholder for the namespace, suffixed with a colon:

```
hasMail(X,Mbox) :- triple(X,"foaf:mbox",Mbox).
```

This rule will internally be expanded to:

```
hasMail(X,Mbox) :- triple(X,"http://xmlns.com/foaf/0.1/mbox",Mbox).
```

5.3 Available Plugins

A number of plugins have been implemented so far and are available at the website of dlvhex (see Subsection 5.2.1). Equally to dlvhex itself, each plugin is available as precompiled binary as well as source package. The binaries are statically linked to any used nonstandard libraries, such that they should run on a typical system configuration. If the user wishes to compile a plugin herself, she might have to install such additional software. The READMEs on the website of dlvhex contain further information.

5.3.1 The Description Logic Plugin

The Description Logic Plugin simulates the behaviour of dl-programs within the more general framework of HEX-programs. To model dl-programs in terms of HEX-programs, we developed the *description-logics plugin* (or *DL-plugin*), which includes several external atoms, which — in accord to the semantics of dl-programs — also allow for extending a description logic knowledge base, before submitting a query, by means of the atoms' input parameters. Additionally, the DL-plugin provides a rewriter that processes the syntax of dl-atoms as presented in Section 3.2, allowing to use dlvhex directly as a reasoner for dl-programs. Moreover, caching strategies based on the considerations of Subsection 3.9.3 optimize the interaction between dlvhex and the description-logics reasoner.

In the implementation of this plugin, we use RACER as a DL reasoning engine [Haarslev and Möller, 2001], being able to process OWL DL ontologies, i.e., description logic knowledge bases in the language $\mathcal{SHOIN}(\mathbf{D})$. However, the interface to the external reasoner is general enough to be adapted easily to any other description-logics engine. Future work will involve the integration of Pellet [Parsia and Sirin, 2004] as an alternative reasoning engine.

DL-Plugin Atoms

Concept Queries A query for the extension of a concept is carried out by the *&dlC* atom in the following way:

```
&dlC[uri,a,b,c,d,q](X)
```

with the following input parameters:

- `uri` ... constant string denoting the uri of the DL-KB, e.g., `"/home/roman/dl/wine.owl"` or `"http://www.example.org/food.owl"`
- `a` ... predicate symbol denoting the positive extensions of concepts. For instance, specifying 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` ... predicate symbol denoting the negative extensions of concepts. This works like the term above, but here not the concept itself is extended, but its complement.
- `c` ... predicate symbol denoting the positive extension of properties. For example, using a predicate `foo` while having an interpretation including the fact *foo*("*knows*", "*Bob*", "*Alice*") will put the pair ⟨"*Bob*", "*Alice*"⟩ into the property *knows*.
- `d` ... predicate symbol denoting the negative extension of properties. Again, this corresponds to the previous input term, except that it adds the negation of a role membership axiom.
- `q` ... constant string that denotes the concept to be queried.

The four input parameters that allow for extending the DL-KB are common to all DL-atoms and will not be explained for them in the following.

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

```
&dlR[uri,a,b,c,d,q](X,Y)
```

Naturally, the result here is binary.

Datatype Role Queries A query for pairs of a datatype role requires another external atom, represented by *&dlDR* as follows:

```
&dlDR[uri,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.

Example 5.3.1 The following example program demonstrates the usage of these three external atoms. Its purpose is to search a database about researchers and create one model per each research project together with those people involved that have Semantic-Web related research interests.

$$
\begin{aligned}
&url(\text{“http://www.personal-reader.de/rdf/ResearcherOntology.owl”}).\\
p(X) &\leftarrow \&dlC[U,a,b,c,d,\text{“}ResearchProject\text{”}](X), url(U).\\
\neg proj(X) \vee \neg proj(Y) &\leftarrow p(X), p(Y), X \neq Y.\\
proj(X) &\leftarrow not\ \neg proj(X), p(X).\\
mem(X) &\leftarrow \&dlR[U,a,b,c,d,\text{“}involvedIn\text{”}](X,Y), url(U), proj(Y).\\
name(X,Y) &\leftarrow mem(X), \&dlDR[U,a,b,c,d,\text{”}name\text{”}](X,Y), url(U).\\
interest(X,Y) &\leftarrow \&dlDR[U,a,b,c,d,\text{“}currentProfessionalInterest\text{”}](X,Y),\\
&\qquad mem(X), url(U).\\
swperson(X,Y) &\leftarrow name(Z,X), interest(Z,Y), \&strstr[Y,\text{“}Semantic\ Web\text{”}].
\end{aligned}
$$

The fact in the first line determines the URL of the ontology, which is used for all queries in this program. The second rule queries the concept *ResearchProject*, whose extension is used for creating a search space via disjunction by the next two rules. Subsequently, the members of a project are queried and — via two datatype role queries — their real names and their research keywords extracted. Eventually, those members of a project are identified that have “Semantic Web” among their research interests.[2] Since no input to the DL-KB is used in this example, a, b, c, and d serve as dummy predicates, which do not occur outside any external atom.

The answer sets (at the time of writing this chapter) are the following (listing only the relevant predicates *swperson* and *project*):

$\{project(\text{“}elena\text{”}),$
$swperson(\text{“}Tina\ Martens\text{”}, \text{“}Semantic\ Web\text{”})\}$

$\{project(\text{“}prolearn\text{”}),$
$swperson(\text{“}Sebastian\ Mayer\text{”}, \text{“}Semantic\ Web\text{”})\}$

$\{project(\text{“}elan\text{”}),$
$swperson(\text{“}Johanna\ Schulz\text{”}, \text{“}Semantic\ Web\text{”}),$
$swperson(\text{“}Sebastian\ Mayer\text{”}, \text{“}Semantic\ Web\text{”})\}$

$\{project(\text{“}knowledgeweb\text{”}),$
$swperson(\text{“}Peter\ Schulze\text{”}, \text{“}Semantic\ Web\text{”}),$
$swperson(\text{“}Susanne\ Meier\text{”}, \text{“}Semantic\ Web\ Architektur\text{”})\}$

$\{project(\text{“}REWERSE\text{”}),$
$swperson(\text{“}Paula\ Lavinia\ Patranjan\text{”}, \text{“}Evolution\ of\ data\ on\ the\ Semantic\ Web\text{”}),$
$swperson(\text{“}Sebastian\ Schaffert\text{”}, \text{“}Semantic\ Web\text{”}),$
$swperson(\text{“}Roman\ Schindlauer\text{”}, \text{“}Semantic\ Web\text{”}),$

[2] This last rule of the example program makes use of the substring-finding external atom that will be introduced in the following subsection.

swperson(*"Grigoris Antoniou"*, *"Web-based systems: Semantic Web, Web services"*),
swperson(*"Tim Furche"*, *"Semantic Web"*),
...}

◇

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

```
&dlCQ[uri,a,b,c,d,q](X̄)
```

Here, `X̄` represents a tuple of arbitrary arity, reflecting the free variables in the conjunction `q`. Conjunctive queries provide a very versatile way of interfacing the DL-reasoner. Multiple queries can be joined into a single one, reducing the number of interactions between dlvhex and the DL-reasoner as well as the size of the transferred extensions between them. However, the conjunction of two queries might not necessarily yield the same result as their join outside the description logic KB, as the following example shows.

Example 5.3.2 Let $KB = (L, P)$ be a dl-program with L:

$$\begin{aligned} Zebra &\sqsubseteq Animal \\ Lion &\sqsubseteq Animal \\ Lion &\sqsubseteq \exists eats.Zebra \\ &Lion(bob) \end{aligned}$$

and P:

$$\begin{aligned} &add(\text{“}Lion\text{”}, \text{“}Bob\text{”}). \\ &carnivore(X) \leftarrow \&dlCQ[\text{“}L\text{”}, add, b, c, d, \text{“}eats(X,Y), Animal(Y)\text{”}](X). \end{aligned}$$

Since *Bob* is added to the concept *Lion*, he must also occur in the relation *eats* together with a *Zebra*. In other words, 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. Now let us replace the second rule of the program by the following one:

$$\begin{aligned} carnivore(X) \leftarrow &\&dlR[\text{“}L\text{”}, add, b, c, d, eats(X,Y)](X,Y), \\ &\&dlC[\text{“}L\text{”}, add, b, c, d, Animal(Y)](Y). \end{aligned}$$

Here, we use conventional role and concept queries and join their result only in the logic program. Given that no explicit tuple occurs in *eats*, the result of the first query is empty and thus also the extension of *carnivore*. In general, the user has to be aware of such subtleties, resulting from the semantic gap between logic programs and description logics.[3]

◇

Consistency Check The atom *&dlConsistent* checks whether the DL-KB is consistent with the specified extensions:

```
&dlConsistent[uri,a,b,c,d]
```

Since this atom is purely Boolean, it does not have any output term.

[3] However, in the concrete implementation of this plugin, we are disburdened from this concern, since RACER lets every variable in a conjunctive query expression range only over the domain of known individuals, i.e., adopting a notion of *safety* comparable to the finite domain of logic programming semantics.

DL-Plugin Rewriter

As noted above, the description logic plugin can model the semantics of dl-programs. It is obvious that the generality of an external atom in the HEX-syntax results in this rather inconvenient method of specifying the extension of the DL-KB by four predicates. In contrast, dl-atoms, having the sole purpose of interacting with an ontology, are more intuitive by allowing for a list of mappings of arbitrary length. Here, we have a perfect situation for deploying a rewriter, which allows to use dl-atoms in a HEX-program and ensures the correct transformation to HEX-syntax before the actual evaluation of dlvhex commences.

Example 5.3.3 To show the principle of rewriting in the case of converting dl-atoms to external atoms, we assume the usage of an atom

$$DL[knows \uplus worksWith, Available \uplus vacation; Attendants](X)$$

in a dl-program $DL = (L, P)$ with $L =$ "personell.owl", querying the members of class *Attendants*, while extending the role *knows* by the extension of the predicate *worksWith* and adding the extension of predicate *vacation* to the complement of the class *Available*. Using dl-syntax — recall that dl-programs work on a single DL-KB — the user has to specify the URI of the ontology by a parameter on the command line. Together with this information, the rewriter replaces the dl-atom by the following external atom (ensuring that the four predicates in its input list did not occur in P prior to the rewriting):

$$\&dlC[\text{"personell.owl"}, plusC, minusC, plusR, minusR, \text{"}Attendants\text{"}](X)$$

and adds the following rules to the program:

$$plusR(\text{"}knows\text{"}, X, Y) \leftarrow worksWith(X, Y).$$
$$minusC(\text{"}Available\text{"}, X) \leftarrow vacation(X).$$

◇

Another interesting feature of this plugin's rewriter is *query pushing*, i.e., combining distinct description logic queries into one conjunctive query — either within the same body of a rule, or even, by rule unfolding, across multiple rules (see Footnote 3 on Page 117 explaining why we can follow this strategy without changing the program's semantics).

The following example shows how to use conjunctive queries in order to decrease the necessary interactions between dlvhex and the DL-engine and hence speed up the computation.

Example 5.3.4 Imagine we want to travel through regions where wines with delicate flavors are available:

$$\begin{aligned} visitRegion(L) \vee \neg visitRegion(L) \leftarrow\ & DL[WhiteWine](W_1), \\ & DL[RedWine](W_2), \\ & DL[locatedIn](W_1, L), \\ & DL[locatedIn](W_2, L) \\ & not\ DLCQ[locatedIn(L, L')](L). \\ \leftarrow\ & visitRegion(X), visitRegion(Y), X \neq Y. \\ haveRegion \leftarrow\ & visitRegion(X). \end{aligned}$$

$$
\begin{aligned}
&\leftarrow \mathit{not}\ \mathit{haveRegion}.\\
\mathit{delicate}(W) &\leftarrow DL[\mathit{hasFlavor}](W, \mathit{wine}{:}\mathit{Delicate}).\\
\mathit{delicateInRegion}(W) &\leftarrow \mathit{visitRegion}(L),\\
&\qquad \mathit{delicate}(W), DL[\mathit{locatedIn}](W, L)
\end{aligned}
$$

The first rule uses disjunction to create a search space over regions to be visited. We consider regions of red and white wines and only "top-level" regions, i.e., those which are not contained in other regions. The following three rules ensure that each answer set contains exactly one region to be visited. In the chosen regions, we single out delicate wines.

The result of query pushing by rewriting the program is displayed below:[4]

$$
\begin{aligned}
\mathit{visitregion}(L) \vee \neg \mathit{visitregion}(L) &\leftarrow DLCQ[WhiteWine(W_1),\\
&\qquad RedWine(W_2),\\
&\qquad locatedIn(W_1, L),\\
&\qquad locatedIn(W_2, L)](L),\\
&\quad \mathit{not}\ DLCQ[\mathit{locatedIn}(L, L')](L).\\
&\leftarrow \mathit{visitregion}(X), \mathit{visitregion}(Y), X \neq Y.\\
\mathit{haveRegion} &\leftarrow \mathit{visitRegion}(X).\\
&\leftarrow \mathit{not}\ \mathit{haveRegion}.\\
\mathit{delicate}(W) &\leftarrow DL[\mathit{hasFlavor}](W, \mathit{wine}{:}\mathit{Delicate}).\\
\mathit{delicateInRegion}(W) &\leftarrow \mathit{visitRegion}(L),\\
&\quad DLCQ[\mathit{hasFlavor}(W, \mathit{wine}{:}\mathit{Delicate}),\\
&\qquad \mathit{locatedIn}(W, L)](W, L).
\end{aligned}
$$

The body of the first rule now contains two external atoms instead of five, thus drastically reducing the effort to interact with the description-logics reasoner, both in terms of time and size of exchanged data (instead of receiving five times the extension of a concept or role, we only need to wait for two such sets of tuples). The join operation is simply transferred to the external engine, keeping only the "relevant" variables in the output list. Execution time of this HEX-program was reduced from 2.68 seconds to 0.59 seconds as a result of query pushing. ◇

The removal of the atom $\mathit{delicate}(W)$ and the introduction of a conjunctive query in the body of the last rule may seem odd, since it doesn't cut down the number of external atoms any further. However, the rewriter is aware of the possibility that the user can specify a filter, i.e., a list of predicates, which causes dlvhex to reduce its visible output to their extensions (see Subsection 5.2.2 about the command-line parameters of dlvhex). If query pushing causes a rule to be "superfluous" w.r.t. considering a filter, i.e., its head neither contained in the filter predicates nor occurring in any other rule, it can be discarded. The penultimate rule in the previous example is a candidate for such a removal. Dropping it from the program, execution time was further reduced to 0.21 seconds.

Further information about the description logics plugin in general and its optimization strategies regarding caching and particularly query pushing can be found in [Krennwallner, 2007].

[4] Strictly speaking, this is not a valid syntax, since we use the simplified syntax of dl-atoms also for the conjunctive query atoms, though they are not accepted by the rewriter.

5.3.2 The String Plugin

The string plugin supplies a number of string manipulation functions that come in handy when we deal with strings in an answer-set program. It currently consists of five atoms.

String concatenation The `&concat` atom is used as follows:

```
&concat[A,B](X)
```

where `A` and `B` are two constant strings and `X` is the concatenation result.

Example 5.3.5 The result of the program

$$folder(\text{“/home/staff/roman/examples/”}).$$
$$file(\text{“test.owl”}).$$
$$path(P) \leftarrow \&concat[Fo, Fi](P), folder(Fo), file(Fi).$$

includes the fact $path(\text{“/home/staff/roman/examples/test.owl”})$. ◇

Substring inclusion Testing whether one string is contained by another can be carried out with the `&strstr` atom:

```
&strstr[A,B]
```

This boolean atom evaluates to true if `A` is a substring of `B`.

Example 5.3.6 Evaluating these rules

$$address(\text{“file://home/staff/roman/examples/wine.owl”}).$$
$$address(\text{“http://www.example.org/food.owl”}).$$
$$webAddress(A) \leftarrow \&strstr[\text{“http://”}, A], address(A).$$

will yield $webAddress(\text{“http://www.example.org/food.owl”})$. ◇

Splitting a string The atom

```
&split[A,D,N](X)
```

splits the string `A` along the delimiter specified by `D` and returns the $\texttt{N}^{\text{th}}$ (starting with zero) element.

Example 5.3.7 The program

$$date(\text{“2006”}).$$
$$date(\text{“31-12-2006”}).$$
$$year(Y) \leftarrow \&split[D, \text{“-”}, 2](Y), date(D).$$

returns $year(\text{“2006”})$. ◇

String comparison A lexicographical comparison of two strings can be achieved by the atom

```
&cmp[A,B]
```

which returns true if `A` is lexicographically smaller than `B`.

Example 5.3.8 The rule

$$in2006(D) \leftarrow \&cmp[D, \text{“2007-01-01”}], \&cmp[\text{“2005-12-31”}, D], date(D).$$

singles out date-strings that belong to the year 2006. ◇

Checksum computation SHA1 (160-bit) checksums are used for instance to publish a hash for someone's email address without revealing the address to spammers. The atom

```
&sha1sum[A](X)
```

returns the checksum `X` of a given string `A`.

Example 5.3.9 The rule

$$ownerID(X) \leftarrow \&sha1sum[X](Y), mailbox(X).$$

computes the SHA1 checksum of an email address. ◇

5.3.3 The RDF Plugin

The RDF-plugin provides a single external atom

```
&rdf[url](S,P,O)
```

which queries a specified set of RDF knowledge bases and returns its triples. The knowledge base is specified by means of the extension of the input predicate.

Example 5.3.10 The following program simply reads all triples from a *foaf*-description:

$$triple(S,P,O) \leftarrow \&rdf[in](S,P,O).$$
$$in(\text{“http://example.org/foaf.rdf”}).$$

Naturally, the extension of *in* can be determined by other subprograms, so that the number of RDF-sources can grow dynamically at runtime. ◇

Two examples of the *&rdf* atom were already given in Subsection 4.6.4, when the computation of HEX-programs was described.

Number	Search
1	Antonyms
2	Hypernyms
3	Hyponyms
4	Entailment
5	Similar
6	Member Holonyms
7	Substance Holonyms
8	Part Holonyms
9	Member Meronyms
10	Substance Meronyms
11	Part Meronyms
14	Cause to
15	Participle of Verb
18	Attribute of
20	Derivationally related forms
23	Synonyms
38	Instance of
39	Instances

Table 5.1: WordNet search types and their encodings.

5.3.4 The WordNet Plugin

WordNet is a lexical reference database, containing structural information about relationships between words, based on psycho-lingual theories of human lexical memory. The WordNet database exists as an online interface as well as a standalone application, providing an API for custom applications. This interface was exploited by Bock in [2006] for developing a WordNet plugin, supplying a layer between the lexicographic database and an answer-set program and thus setting the ground for a number of declarative approaches for linguistic reasoning tasks.

The WordNet database is queried with respect to a specific (English) word and a certain query. Queries are specified by a search type that has a numerical encoding (see Table 5.1). The plugin provides a single atom, which expects the actual word and the desired search number as input:

```
&wordnet[S,W](P,WS,R,RS)
```

The first input parameter is the search number from Table 5.1, followed by the input word. Each query yields in general a number of output tuples, where the first term `P` denotes the "part of speech" (*Noun* is encoded by 1, *Verb* by 2, *Adjective* by 3, and *Adverb* by 4). `WS` represents the input word sense number that this result tuple is based on. the third term, `R`, is the actual result word and `RS` is its sense number.

Example 5.3.11 The following rules return the synonyms of a given word:

$word(\text{“}provide\text{”}).$
$synonyms(W, WS, SY) \leftarrow \&wordnet[23, W](\_, WS, SY, \_), word(W).$

Here, we ignore the sense numbers of input and result words. The answer set of this program contains these facts:

$synonyms(\text{“}provide\text{”}, 1, \text{“}furnish\text{”}, 2)$, $synonyms(\text{“}provide\text{”}, 2, \text{“}ply\text{”}, 2)$,
$synonyms(\text{“}provide\text{”}, 1, \text{“}render\text{”}, 2)$, $synonyms(\text{“}provide\text{”}, 2, \text{“}supply\text{”}, 2)$,
$synonyms(\text{“}provide\text{”}, 1, \text{“}supply\text{”}, 2)$, $synonyms(\text{“}provide\text{”}, 2, \text{“}provide\text{”}, 2)$,
$synonyms(\text{“}provide\text{”}, 1, \text{“}provide\text{”}, 2)$, $synonyms(\text{“}provide\text{”}, 3, \text{“}provide\text{”}, 2)$,
$synonyms(\text{“}provide\text{”}, 2, \text{“}cater\text{”}, 2)$, ...

A concrete application using the WordNet-plugin is described in Subsection 6.3, where the lexicographic background knowledge is used for the task of merging ontologies. ◇

5.4 Writing a Plugin

In order to ease the process of developing a plugin for dlvhex as much as possible, we created a skeleton plugin package, which is also embedded in the GNU autotools environment. This template frees the plugin author from being occupied with the low-level, system-specific build process.

The necessary interactions between dlvhex and a plugin are:

- Registering the plugin and its atoms
- Evaluating an atom

For both tasks, we provide a base class which the plugin author has to subclass. As a running example, we will use the aforementioned `&concat`-atom. To begin with, the following header files from dlvhex need to be included:

```
#include "dlvhex/PluginInterface.h"
#include "dlvhex/Error.h"
```

If dlvhex was installed correctly, these headers should be available.

5.4.1 The External Atom

First, we have to subclass **PluginAtom** to create our own **ConcatAtom** class:

```
class ConcatAtom : public PluginAtom
{
```

The constructor of **ConcatAtom** will be used to define some properties of the atom:

```
public:
    ConcatAtom()
    {
        addInputConstant();
        addInputConstant();
        setOutputArity(1);
    }
```

Here, we need to specify the number and types of input parameters. The `&concat`-atom as we used it above, has two constant input strings. In this case, we need two consecutive calls to **addInputConstant()**. Recall that input parameters can be of type *constant* of

predicate, the latter denoting that the extension of a predicate has to be considered rather than the parameter string itself. If we wanted to have, say, two input parameters which will represent predicate names and a third one which will be a constant, we would put this sequence of calls in the constructor:

```
addInputPredicate();
addInputPredicate();
addInputConstant();
```

The call `setOutputArity(1)` ensures that occurrences of this atom with other than one output term will cause an error at parsing time.

The only member function of the atom class that needs to be defined is **retrieve**:

```
virtual void
retrieve(const Query&  query,
                Answer& answer) throw(PluginError)
{
```

retrieve() always takes a query object as input and returns the result tuples in an answer object. The input parameters at call time are accessed by **Query::getInputTuple()**:

```
Tuple parms = query.getInputTuple();
```

The actual input strings have to be extracted from the input tuple:

```
Term s1 = parms[0];
Term s2 = parms[1];
```

At this point we could test the input terms for correct types. The author is able to throw an exception of type **PluginError** which will be caught inside dlvhex, e.g.:

```
if (s1.isInt() || s2.isInt())
    throw PluginError("Wrong input argument type");
```

If the evaluation function needed to take the interpretation into account, it would have to be retrieved by **Query::getInterpretation()**:

```
Interpretation i = query.getInterpretation();
```

For more information about the datatypes of dlvhex, please refer to the comments of the respective include files provided with the development resp. source package of dlvhex.

At this point, the plugin author will implement the actual function of the external atom, either within this class or by calling other functions.

Before returning from the retrieve-function, the answer-object needs to be prepared:

```
    std::vector<Tuple> out;

    // fill the answer object...

    answer.addTuples(out);
}
```

5.4.2 Registering the Atoms

So far, we described the implementation of a specific external atom. In order to integrate one or more atoms in the interface framework, the plugin author needs to subclass **PluginInterface()**:

```
class StringPlugin : public PluginInterface
{
public:
```

At the current stage of dlvhex, only the function **getAtoms()** needs to be defined inside this class:

```
    virtual void
    getAtoms(AtomFunctionMap& a)
    {
        a["concat"] = new ConcatAtom;
    }
```

Here, a **PluginAtom** object is related to a string in a map - as soon as dlvhex encounters an external atom (in this case: `&concat`) during the evaluation of the program, it finds the corresponding atom object (and hence its **retrieve()** function) in this map. Naturally, a plugin can comprise several different PluginAtoms, which are all registered here:

```
        a["concat"] = new ConcatAtom;
        a["split"] = new SplitAtom;
        a["cmp"] = new CmpAtom;
```

5.4.3 Importing the Plugin

Eventually, we need to import the plugin to dlvhex. this is achieved by the dynamic linking mechanism of shared libraries. The author needs to define this function:

```
extern "C"
StringPlugin*
PLUGINIMPORTFUNCTION()
{
    return new StringPlugin();
}
```

replacing the type StringPlugin by her own plugin class. For each found library, dlvhex will call this function and receive a pointer to the plugin object, thus being able to call its **getAtoms()** function. The identifier of this function, `PLUGINIMPORTFUNCTION`, is a macro defined by dlvhex.

The entire example code for two atoms from the string plugin can be found in Appendix B.

The next two sections describe the implementation of a plugin rewriter and the handling of command-line options for plugins, both optional extensions that need not to be implemented for a plugin to work.

5.4.4 The Rewriter

A rewriter is represented by an own class, **PluginRewriter**. If the user desires to implement a rewriter, she has to subclass it:

```
class MyRewriter : public PluginRewriter
{
public:
```

The constructor is initialized with two streams, one for the input program and one for the rewritten program, which both have to be passed to the base class:

```
MyRewriter(std::istream& i, std::ostream& o)
    : PluginRewriter(i,o)
{ }
```

The variables for the streams in the base class are **input** and **output**, both of type **std::istream***. With these, the actual rewriting method operates:

```
void
rewrite()
{
    // something like:
    std::stringbuf sb;
    input->get(sb, '\n');
    std::string line = sb.str();

    ...

    *output << line;
}
```

It lies in the responsibility of the rewriting function to ensure that a syntactically valid program is written to the output-stream. For more information, we recommend to have a look at the source code of the description logic plugin, where a rewriter is implemented. To associate the rewriter-object with the corresponding plugin, the method **createRewriter()** has to be overloaded in the plugin-class. The simplest way would be to just create the rewriter and return it (assuming to create a rewriter for the string-plugin):

```
PluginRewriter*
StringPlugin::createRewriter(std::istream& i, std::ostream& o)
{
    return new MyRewriter(&i, &o)
}
```

5.4.5 Command Line Option Handling

In order to propagate dlvhex command-line parameters to the plugin, another method of **PluginInterface** has to be overloaded. The function **setOptions()** will be called by dlvhex with a list of all unknown parameters. If the plugin processes a parameter, it has to remove it from the list, since dlvhex will exit with an error if any parameters are left after "asking" its plugins (again we continue the example with the StringPlugin from above):

```
void
StringPlugin::setOptions(bool help,
                         std::vector<std::string>& argv,
                         std::ostream& out)
{
    // loop though the vector argv, remove "own" options
    // or
    // dump help-msg to out, if requested
}
```

If the parameter **help** is true, then dlvhex was called with `-h` and the plugin can display a help message by streaming it into **out**, explaining its own command line strings, if necessary. However, unknown or wrong command line options have precedence and should be recognized also when help was requested.

5.4.6 Building the Plugin

We provide a toolkit for building plugins based on the GNU autotools environment. After downloading and extracting the toolkit skeleton, the user can customize the top-level `configure.ac` to her own needs regarding dependencies on other packages. The only source-directory in this template is `src`, where `Makefile.am` sets the name of the library and its source-files. A rudimentary source file exists, `plugin.cpp`, which includes the mandatory subclasses and methods as shown above. When adding further source-subdirectories (like `include`), the user has to take care of referencing them in the top-level `Makefile.am`. Calling

```
./bootstrap.sh
```

in the top-level directory creates the configure file. Subsequently calling

```
./configure
```

and

```
make install
```

installs the shared library. If no further arguments are given to `configure`, the plugin will be installed into the system-wide plugin-directory (specified by the package configuration file `lib/pkgconfig/dlvhex.pc` of the devel-package). To install the plugin into `~/.dlvhex/plugins`, run

```
./configure --enable-userinstall
```

However, dlvhex itself provides a command line switch to provide a further directory at runtime where to search for plugin libraries.

Chapter 6

HEX-Program Applications

This chapter presents some application scenarios where dlvhex is already used or considered to be integrated. They show the benefits of the specification of custom plugins as well as the expressivity of the answer-set semantics by means of real-world examples.

6.1 Optimizing Trust Negotiations

Trust management is an important field of research in the domain of open distributed and decentralized systems. One possible approach to trust management is based on policies, relying on objective security mechanisms such as signed certificates and trusted certification authorities in order to regulate the access of users to specific services. Access decision is usually based on mechanisms with well-defined semantics, that allow for verification and analysis support. The result of such a policy-based trust management approach usually consists of a binary decision whether the requester is trusted or not, respectively the service or resource is allowed or denied. These decision can be based on "non-subjective" attributes which might be certified by certification authorities via digital credentials.

Systems that enforce policy-based trust can profit from declarative logic programming. In a typical scenario, every party in such a negotiation can define its own policies to control external use of its resources by logic programming rules. The interaction between two agents that formulate requests and disclosure information based on their own rule-based policies demands spontaneous negotiation protocols whose behavior is difficult to predict. It is not clear whether the negotiation is going to succeed even when the policies in principle allow it, since they might be protected as sensitive resources and therefore not disclosed in the negotiation. Moreover, when credentials are released incrementally, it is desirable, but probably not possible for the peers to minimize the sensitivity of the information disclosed during negotiations.

In this application example, we restrict our focus to the special case, where the server fully publishes its policy and credentials. This is likely the case for commercial web services, enabling clients to protect their privacy and minimize information disclosure. In such a scenario, it is indeed possible to find a minimum of sensitivity of disclosed information, since the client has all the information it needs to make an optimal choice. This is called the *credential selection problem* (cf. [Bonatti et al., 2006]).

6.1.1 The Credential Selection Problem

Roughly said, the instances of the credential selection problem contain

- a finite, stratified logic program P, representing the server's and client's policies,

- a goal G modeling the authorization requested by the client,
- a finite set of integrity constraints IC, representing forbidden combinations of credentials,
- a finite set of ground facts C, representing the portfolio of credentials and declarations of the client, and
- a *sensitivity aggregation function* $sen : 2^C \to \Sigma$, where Σ is a finite set (of *sensitivity values*) partially ordered by $\preceq$.

A solution for the credential selection problem is a set $S \subseteq C$ such that

1. $P \cup S \models G$,
2. $P \cup S \cup IC$ is consistent, and
3. $sen(S)$ is minimal among all S which satisfy 1. and 2.

Given that the sensitivity levels are real numbers and *sen* is a simple function, such as the sum or maximum of all sensitivity values, it is possible to embed the credential selection into an ASP program. As an example, we consider the ASP system SMODELS, which provides constructs for minimizing or maximizing a function given as a linear sum of weights of literals. Thus, by assigning weights to individual ground atoms, an optimization facility is provided by minimizing the total weight of a stable model.

Such an optimization statement is given as

$$\mathit{minimize}\ \{A_1 = w_1, \ldots, A_n = w_n\},$$

where $w_1, \cdots, w_n$ are the weights associated with the atoms $A_1, \cdots, A_n$, respectively. This directive specifies that the solution is the stable model S of the program with the smallest value for the sum

$$\sum_{l \in L \text{ and } S \models l} w(l)$$

with $L = \{A_1, \ldots, A_n\}$ and $w(a_i) = w_i$.

Embedding the selection problem into SMODELS results in the following program:

$$P \cup IC \cup \{c \leftarrow \mathit{not}\ \bar{c} \mid c \in C\} \cup \{\bar{c} \leftarrow \mathit{not}\ c \mid c \in C\} \cup \{\leftarrow \mathit{not}\ G\} \cup A_{sen} \cup \mathit{minimize}\ O_{sen}$$

Here, $\bar{c}$ denotes a fresh propositional symbol for $c \in C$ in order to create the search space for all combinations of credentials by recursive use of weak negation (note that SMODELS does not provide head disjunction). A_{sen} represents auxiliary rules that are needed for the optimization statement O_{sen}, both of which are different for each specific aggregation function *sen*. The most straightforward function would be $sen = \text{sum}$, i.e., the sum of all sensitivity values in a stable model:

$$A_{sen} = \emptyset \text{ and } O_{sen} = \{c_1 = sen(\{c_1\}), \ldots, c_n = sen(\{c_n\})\}$$

where $C = \{c_1, \ldots, c_n\}$ is the set of selected credentials.

It becomes more complicated to express the optimization function within the logic program if we need to find the maximum sensitivity, i.e., $sen = \max$. Then, let A_{sen} be

the following set of rules, for $i = 1, \ldots, n$, where the predicates *lev* and *maxlev* are new predicates:

$$\begin{aligned} lev(i) &\leftarrow c \qquad \text{where } c \in C \text{ and } sen(\{c\}) = i \\ lev(i) &\leftarrow lev(i+1) \\ maxlev(i) &\leftarrow lev(i), not\ lev(i+1) \end{aligned}$$

and $O_{sen} = \{maxlev(1) = 1, \ldots, maxlev(m) = m\}$, where $m = \max\{sen(\{c\}) \mid c \in C\}$. Intuitively, for all sensitivity values i below the maximal one, $lev(i)$ holds, and hence $maxlev(j)$ holds only for the maximal level j.

It is obvious that for sophisticated aggregation functions, the encoding with rules quickly becomes very intricated or even impossible. We will show next how dlvhex together with an appropriate external atom can generalize and simplify this task to a great extent.

6.1.2 Aggregation by External Computation

The idea is to *externally* compute the overall sensitivity of a stable model based on the selected credentials and then use standard optimization constructs like weak constraints to find the "best" model. Since the interfacing mechanism of dlvhex does not constrain the user to a specific language, she can implement the aggregation function in any suitable formalism. In the following, we will outline the encoding of the credential selection problem within dlvhex.

In contrast to SMODELS, dlvhex provides head disjunction, making it a bit more intuitive than using recursion though negation to encode the search space of credentials:

$$c \vee \neg c \mid c \in C$$

We assume that we have a predicate that associates each credential with its sensitivity, e.g., $sens(c_i, sen(\{c_i\}))$ for each $c_i \in C$ and an external atom $\&policy$, that expects such a binary predicate as input and returns the aggregated sensitivity. We can then state the following weak constraint:

```
:~ &policy[sens](S). [S:1]
```

The extension of *sens* that is passed to the external atom corresponds to the subset of credentials selected in each stable model. The evaluation function of the external atom then can, based on the credentials and their sensitivity values, compute the overall sensitivity for this subset. We then can use this value directly in the weight specification of the weak constraint. Considering that the answer sets of a program P with a set W of weak constraints are those answer sets of P which minimize the number of violated weak constraints, the resulting model(s) are those with the lowest overall sensitivity value.

In the case of *sen* = max and a direct encoding in C++, this function could be implemented as follows:

```
double
PolicySensFunction::eval(const std::vector<double>& values)
{
    double ret(0);
```

```
        for (std::vector<double>::const_iterator di = values.begin();
            di != values.end();
            ++di)
        {
            if ((di == values.begin()) || (*di > ret))
                ret = *di;
        }

        return ret;
    }
```

Here we assume that the vector **values** contains the sensitivity values in *sens* and do not consider the credentials themselves.

Appendix C contains a simplified policy specification together with the optimization part as a HEX-program.

6.2 Implementing a SPARQL Reasoning Engine

An interesting application of HEX-programs and especially the features of dlvhex is the approach of using a plugin's rewriter to transform the input data into a valid HEX-program. We already presented such a rewriting facility to simulate a reasoner for dl-programs by rewriting dl-atoms into external atoms provided by the Description Logics plugin. However, since such a rewriter is a completely generic module, the input data is not limited at all to a set of rules. In this section, we present a method of rewriting a query on RDF data stated in the language SPARQL [Prud'hommeaux and Seaborne, 2006][1] into a set of rules, the answer set of which represents the query result.

6.2.1 The RDF Query Language SPARQL

SPARQL is a query language for RDF data which shares some similarities with relational database management languages, such as SQL, but also shows considerable differences. SPARQL is capable of extracting values from RDF data such as literals and URIs as well as entire subgraphs and can construct new RDF graphs from these results. The basic building blocks of a query are the SELECT clause, selecting the output variables, and the WHERE clause, specifying a so-called *basic graph pattern* to be matched with the input graph, i.e., the RDF data.

The relation between RDBMS query languages and Datalog has been studied thoroughly. In general the former are considered to be more expressive than the latter, especially since, for instance, SQL is already capable of recursion under stratified negation. Compared to SQL, plain Datalog lacks features such as aggregates and built-in arithmetic functions. SPARQL, on the other hand, is also restricted w.r.t. SQL, missing aggregates, recursion and nested queries. However, it introduces other peculiarities which make it difficult to find a straightforward translation to Datalog, such as blank nodes, or the UNION and OPTIONAL constructs, which allow for the specification of pattern alternatives. Another particularity of SPARQL is the possibility to specify graph patterns which do not have to be matched with the input data, but only if the respective information is available. Hence, triples that do not match such an optional pattern are not automatically discarded.

[1] The semantics of SPARQL is not entirely finalized at the time of writing.

Extensions of Datalog such as default negation, disjunctive rules and aggregate atoms go beyond the expressivity of SQL. In [Polleres and Schindlauer, 2006] we investigate the relationship between SPARQL and such languages and present a strategy how to combine them.

6.2.2 Rewriting SPARQL to Rules

A plugin for dlvhex does not necessarily have to provide any external atoms, but can merely apply a rewriter to the input data. To this end, the SPARQL-plugin has been implemented, which uses a parser for SPARQL to process an RDF query and transform it into an appropriate set of rules. Thus, a formal translation from SPARQL into ASP was given, capturing most of the available SPARQL constructs.

Importing the RDF data can be accomplished with the *&rdf* atom. The translation then takes care of selecting the respective variables from the graph pattern in the query, particularly considering UNION and OPTIONAL clauses. The latter can be elegantly approximated by weak negation in the logic program.

The example below demonstrates the translation strategy for a simple SPARQL query.

Example 6.2.1 The following query returns the names of individuals from a FOAF-URI. If their mailboxes or homepages are available, they are added to the respective triple:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name ?mbox ?hpage
FROM <http://www.kr.tuwien.ac.at/staff/roman/foaf.rdf>
WHERE  { ?x foaf:name  ?name .
         OPTIONAL { ?x foaf:mbox ?mbox } .
         OPTIONAL { ?x foaf:homepage ?hpage }
       }
```

The translation delivers the following rules:

$$\#namespace(\text{“}foaf\text{”}, \text{“}http://xmlns.com/foaf/0.1/\text{”})$$

(1) $triple(S, P, O, default) \leftarrow \&rdf[\text{“http://www.kr.tuwien.ac.at/staff/roman/foaf.rdf”}](S, P, O).$

(2) $answer_1(Xname, Xx) \leftarrow triple(Xx, \text{“}foaf{:}name\text{”}, Xname, default).$

(3) $answer_2(Xmbox, Xx) \leftarrow triple(Xx, \text{“}foaf{:}mbox\text{”}, Xmbox, default).$

(4) $answer_3(Xhpage, Xx) \leftarrow triple(Xx, \text{“}foaf{:}homepage\text{”}, Xhpage, default).$

(5) $answer\_opt_1(Xmbox, Xname, Xx) \leftarrow answer_1(Xname, Xx), answer_2(Xmbox, Xx).$

(6) $answer\_opt_1(null, Xname, Xx) \leftarrow answer_1(Xname, Xx), not\ answer'_2(Xx).$

(7) $answer'_2(Xx) \leftarrow answer_2(Xmbox, Xx).$

(8) $answer\_opt_2(Xhpage, Xmbox, Xname, Xx) \leftarrow answer\_opt_1(Xmbox, Xname, Xx), answer_3(Xhpage, Xx).$

(9) $answer\_opt_2(null, Xmbox, Xname, Xx) \leftarrow answer\_opt_1(Xmbox, Xname, Xx), not\ answer'_3(Xx).$

(10) $answer'_3(Xx) \leftarrow answer_3(Xhpage, Xx).$

(11) $answer(Xhpage, Xmbox, Xname) \leftarrow answer\_opt_2(Xhpage, Xmbox, Xname, Xx).$

The first four rules import the RDF dataset into the HEX-program, creating one predicate for each graph pattern. Rules (5) and (6) collect those tuples that have both a name and mailbox as well as those without a mailbox, realizing the semantics of the first OPTIONAL clause (using a projection created by the auxiliary Rule (7)). These results are then used in Rules (8) and (9), gathering tuples with a homepage and tuples without, corresponding to the second OPTIONAL. Both of these outer joins insert the constant *null* for non-existing optional values. The final result is the extension of the predicate *answer*. ◇

Currently, the translation comprises the following features:

- SELECT queries with simple graph patterns, UNION and OPTIONAL
- N3, RDF/XML, Turtle syntax for input
- Full N3 syntax for triple patterns, including blank nodes and arbitrary nesting of patterns
- Simple conjunctive FILTER expressions: conjunction, isBound, isBlank, binary comparison operators; for the latter, we use nonstandard semantics that are based on available `DLV`-builtins (for details about the proper semantics of such operators, cf. [Prud'hommeaux and Seaborne, 2006])

At its current stage, the prototype has a number of limitations. First of all, CONSTRUCT, ASK and DESCRIBE result forms are not supported, since the query result is simply given as the extension of a dedicated predicate. The support of filters is still preliminary. Moreover, there is no awareness of language tags or typed literals.

Future work in this direction not only comprises the enhancement of this prototype in order to cover SPARQL to a greater extent, but also to take advantage of the insights gained from the combinations of SPARQL and answer-set programming and identify possible extensions for the query language itself, e.g., adding features such as aggregates, recursion and nested queries.

6.3 Ontology Merging

Merging and aligning of ontological knowledge bases is one prominent problem in the context of the Semantic Web. Different such ontologies might be developed independently, with partly overlapping knowledge and structures. A straightforward strategy for merging such repositories usually does not exist. The approaches regarding merging and aligning can generally be categorized in *elementary matchers*, which follow a single, specific algorithm, and *hybrid solutions*, combining several elementary matchers. Among the schema-based approaches, several methods rely on linguistic resources, deriving structural knowledge from common knowledge thesauri, such as WordNet. Provided that the respective ontologies use labels from human language, these methods are based on linguistic relations, such as synonymy or meronymy.

In his honours thesis, Jürgen Bock [2006] has pursued a novel approach for ontology merging, using answer-set programming as the logical underpinning for the implementation of an elementary matching algorithm, resorting to WordNet as a background knowledge base for lexical and domain knowledge. Such a declarative programming formalism allows for the specification of a merging algorithm in an intuitive and maintainable fashion. The universal coupling mechanism of HEX-programs suggested to implement an interface to the database of WordNet. To this end, Bock developed the WordNet-plugin, which was already

presented in Subsection 5.3.4. The WordNet plugin provides a single external atom, which is able to perform all query types that are supported by the API of WordNet. This section summarizes the relevant portions of [Bock, 2006].

The condition for this method is the availability of ontologies in a basic ontology language. Most public ontologies, such as web dictionaries, thesauri, and other types of controlled vocabulary, are simple enough to avoid the usage of expressive ontology languages such as OWL or even RDF Schema. For such ontologies the *Simple Knowledge Organization System* (*SKOS*) [Miles and Brickley, 2005] provides a model for expressing the basic structure and content of concept schemes, based on RDF.

6.3.1 Merging Algorithm

The merging algorithm uses WordNet to access two important linguistic relations:

Hyponymy A word w_1 is a hyponym of w_2, if w_1 is more specific than w_2. For example, *steamboat* is a hyponym of *ship*. The inverse relation is *hypernymy*.

Meronymy A word w_1 is a meronym of w_2, if w_1 and w_2 are in a "part-whole" relationship, i.e., , w_1 is a part of, a substance of, or a member of w_2. For instance, *aeroplane* has the meronyms *fuselage*, *wing*, *landing gear*, etc. The inverse relation is *holonymy*.

The proposed strategy is meant to be semi-automatic, letting the human user actively adjust and influence the merging process. First of all, this is guaranteed by providing multiple possible solutions, based on the model-generating property of the answer-set semantics. The models are regarded as *suggestions* for possible merging results. Moreover, the user can define specific parameters according to various requirements regarding the structure of the merging result.

If $c_1$is a concept of an ontology O_1 and c_2 a concept of an ontology O_2, Bock defines the following two relationships:

$$c_1 \simeq c_2 \quad \text{iff} \quad \exists l_1 \in L(c_1) \text{ and } \exists l_2 \in L(c_2) \text{ s.t. } l_1 \text{ and } l_2 \text{ are either identical or synonyms} \tag{6.1}$$

$$c_1 \prec c_2 \quad \text{iff} \quad \exists l_1 \in L(c_1) \text{ and } \exists l_2 \in L(c_2) \text{ s.t. } l_1 \text{ is linguistically narrower than } l_2 \tag{6.2}$$

$L(c)$ denotes all labels of a concept c.[2] A label l_1 is *linguistically narrower* than l_2, if l_2 is a hyponym or meronym of l_2.

The first basic principle of the algorithm is *concept melding* (i.e., merging two concepts into one). Two concept can be melded, if they fulfill condition 6.1:

$$meld(c_1, c_2) \vee \neg meld(c_1, c_2) \leftarrow c_1 \simeq c_2$$

However, melding is optional, since two synonymous concepts can simultaneously satisfy condition 6.2. If this is not the case, the melding is forced:

$$\leftarrow c_1 \simeq c_2, not\, c_1 \prec c_2, not\, c_2 \prec c_1, \neg meld(c_1, c_2)$$

The second principle is *restructuring of hierarchies*. Subsets of all potential narrowing relations are created:

$$pot\_narr(c_1, c_2) \leftarrow c_1 \sqsubset c_2 \quad (c_1, c_2 \in O_i \text{ with } i \in \{1, 2\})$$

$$pot\_narr(c_1, c_2) \leftarrow c_1 \prec c_2 \quad (c_1 \in O_1, c_2 \in O_2)$$

$$m\_narr(c_1, c_2) \vee \neg m\_narr(c_1, c_2) \leftarrow pot\_narr(c_1, c_2)$$

[2] In SKOS, a concept can have multiple labels (called *preferred* and *alternative* labels).

Here, $c_1 \sqsubseteq c_2$ denotes a subclass-relation that already exists in one of the input ontologies.

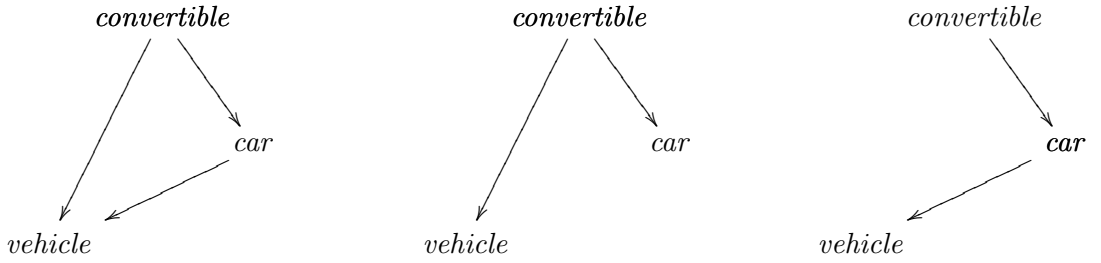
A further selection from this search space is made based on a number of criteria which are applied as constraints. Currently, the following parameters are supported:

no singles Every concept must occur in a narrowing relation.

one root A single root concept must exist in the narrowing hierarchy.

single parent A concept may not have two incoming narrowing relations. This "thins out" the graph as shown in Figure 6.1.

always meld Meld concepts whenever possible.



a) b) c)

Figure 6.1: Effect of *single parent* on a structure of narrowing relations. Figure a) shows the potential narrowing relations, while b) and c) are minimized by *single parent*.

Furthermore, the user is able to adjust the desired length of chained narrowing relations, for instance to prefer a more nested hierarchy of concepts over a flat one (called *brave* merging).

6.3.2 Implementation

A HEX-program is used to implement the mechanism outlined above, using the *&rdf* atom to import the SKOS-ontologies and the *&wordnet* atom to obtain the linguistic relations between concept labels. The parameters are realized as propositional atoms. The result of the process is again transformed into an RDF graph (expressed as ternary atoms), using SKOS vocabulary. Below, we will exemplarily print some of the important parts of the merging program.

The following rules extract the similarity and narrowing relations:

```
#namespace(skos,"http://www.w3c.rl.ac.uk/2003/11/21-skos-core#")

ont_narrower(ONT,B,N) :- triple(ONT,N,"skos:broader", B).
ont_narrower(ONT,B,N) :- triple(ONT,B,"skos:narrower",N).

ling_narrower(C1,C2) :- label(ONT1,C1,L1), label(ONT2,C2,L2),
                        &wordnet[S,L1](1,_,L2,_),
                        linguistic_narrower_search(S), ONT1 != ONT2.

synonym(C1,C2) :- label(1,C1,L1), label(2,C2,L2),
                  &wordnet[23,L1](1,_,L2,_).
```

The effect of the input parameters is achieved by constraints:

```
chained(C,D) :- potential_narrower(C,E), potential_narrower(E,D).
chained(C,D) :- chained(C,E), potential_narrower(E,D).

:- merged_narrower(C,E), chained(C,E), potential_narrower(C,E), brave.

:- merged_narrower(C,D), merged_narrower(B,D), C != B, single_parent.

b(C) :- merged_narrower(_,C).
n(C) :- merged_narrower(C,_).

root(C) :- b(C), not n(C).
root(C) :- not b(C), not n(C), merged_concept(C,_).

:- root(C), root(D), C!=D, one_root.

:- not b(C), not n(C), merged_concept(C,_), no_singles.
```

Testing the algorithm on two small SKOS ontologies with four resp. five concepts each resulted in 1 to 72 merging suggestions, depending on the algorithm's parameters. It shows that brave merging has the strongest effect and greatly reduces the number of answer sets. In general, a high number of solution shows that such an elementary merging approach is best applied as part of a hybrid solution instead of a stand-alone application. However, the algorithm delivers intuitively "suitable" results, and its encoding as a declarative and concise ruleset under a semantics that produces multiple models proves to be a promising approach.

For further information about the merging method as well as the WordNet plugin, we refer to [Bock, 2006].

Chapter 7

Conclusion

The idea of this work was to contribute to the ongoing developments in the Semantic Web and logic programming communities regarding an integration of a rule-layer in the current Semantic Web architecture. Specifically, we tried to exploit the features of non-monotonic reasoning in the form of answer-set programming towards reasoning tasks in the Semantic Web.

After a brief survey over the status quo and an introduction to the basic concepts of answer-set programming and knowledge representation in the Semantic Web through Description Logics and ontologies, we proposed two novel languages.

First, we aimed at an integration of rules and ontologies by combining logic programming under the answer-set semantics with the Description Logics $\mathcal{SHIF}(\mathbf{D})$ and $\mathcal{SHOIN}(\mathbf{D})$, which stand behind OWL Lite and OWL DL, respectively. We have introduced dl-programs, which consist of a description logic knowledge base L and a finite set P of dl-rules, which may also contain queries to L, possibly default-negated, in their bodies. Importantly, we chose an approach of strict semantic separation, in order to avoid common pitfalls when combining such diverge formalisms as logic programming and description logics, which are fragment of first-order logic. Specifically, we avoid undecidability issues due to their entirely different model theories.

We have defined Herbrand models for dl-programs, and shown that satisfiable positive dl-programs have a unique least Herbrand model. More generally, consistent stratified dl-programs can be associated with a unique minimal Herbrand model that is characterized through iterative least Herbrand models. We have then generalized the unique minimal Herbrand model semantics for positive and stratified dl-programs to a strong answer-set semantics for all dl-programs, which is based on a reduction to the least model semantics of positive dl-programs. We have also defined a weak answer-set semantics based on a reduction to the answer sets of ordinary logic programs. We have given fixpoint characterizations for the unique minimal Herbrand model semantics of positive and stratified dl-programs, and shown how to compute these models by finite fixpoint iterations. Furthermore, we have given a precise picture of the complexity of deciding strong and weak answer set existence for a dl-program. Concrete examples have shown the applicability of dl-program applications regarding typical notions of non-monotonic reasoning, such as the closed-world assumption or default logic. Eventually, we presented an implementation of a reasoner for dl-programs, outlining its algorithms and optimization techniques.

The second language can be seen as a further development of dl-programs, generalizing the Description Logics interface to a more general class of external sources of computation. These so-called HEX-programs retain the elegant problem-solving paradigm of answer-set programming and enrich this formalism by the interoperability with other software. At

the same time, they support abstract problem modeling by higher-order features, which are needed for a wide range of applications but missing in ASP systems today. The interface mechanism of HEX-programs, similar to the idea behind dl-programs, follows the approach of strictly separating the semantics of the logic program and the external source of knowledge. Akin to dl-programs, this allows us to extend the answer-set semantics in a straightforward and intuitive fashion, ensuring decidability and finiteness of the domain. Since HEX-programs can integrate heterogenous sources of knowledge in a single logic program, a wide variety of knowledge merging and aligning strategies can be encoded in a concise fashion. User-defined libraries can be incorporated, and thus customization to specific applications is enabled. The non-monotonic constructs as well as the possibility to generate multiple models from a single problem specification are well-suited for many reasoning tasks related to the Semantic Web.

We defined the semantics of general HEX-programs by generalizing the traditional answer-set semantics, defining the model of an external atom and using the FLP-reduct instead of the classical GL-reduct. We have sketched typical examples of the usability of external atoms in the domain of Semantic Web applications, such as importing and manipulation external theories or defining specific ontology semantics by rules of a HEX-program. We then have outlined the general principle of computation of a HEX-program, relying on the dependency information that underlies the program. We aimed at integrating existing reasoners instead of creating an entirely new one and so designed a framework that builds upon available answer-set solver as well as the engines behind external atoms. From this viewpoint, we created an algorithm that builds the models of a HEX-program by decomposing it and subsequently calling the external reasoners. Moreover, we specified syntactic safety constraints to ensure decidability. A prototype implementation, called dlvhex, of these strategies was created, along with a number of so-called plugins which provide several external atoms, interfacing for instance RDF repositories, DL knowledge bases, the WordNet database or simple, but useful string operations. We designed a software architecture that adheres to widespread software standards and allows for quick development of custom plugins.

Eventually, we presented concrete, currently developed applications of HEX-programs, which demonstrate the versatility of both the underlying semantics as well as the computational framework.

We strongly believe that reasoning in the Semantic Web would benefit greatly from advanced, non-monotonic reasoning techniques enforced by declarative languages such as dl-programs or HEX-programs. Especially the latter can act as a glue, integrating heterogenous sources under different semantics in a single inference framework. Even if the Web as a whole is seen as an "open world" that calls for open-world reasoning, it is very likely that in some bounded areas non-monotonic reasoning under the closed-world assumption is the right way to go to be able to eventually reach a conclusion. To this end, we have shown how to use our languages to explicitly apply the closed-world assumption or default reasoning to dedicated parts of knowledge bases.

Currently, we are developing further plugins to import and reason over XML documents, using XPath as query language. Moreover, we plan to implement a plugin for Xcerpt, a rule-based query language for graph-structured data such as XML or RDF. Future work will mainly address the efficiency of dlvhex and its algorithms, as well as useful syntactic extensions, such as typing of parameters corresponding to RDF (literals vs. resource identifiers). We also aim at establishing a more general framework for this class of logic programs that reason about imported knowledge, which could capture also other existing approaches of combining rules and ontologies and facilitate the specification of a

standardized, web-based reasoning access to inference engines such as dlvhex.

Practical experiments with dlvhex have shown that the current computation method lacks efficiency in case of unstratified HEX-programs, where external atoms occur in negative cycles. This stems from the inevitable guess-and-check algorithm and cannot be avoided within the chosen framework of calling an external answer-set solver for all intermediate and final model generations. A solution to this problem would be a tighter integration between the evaluation of external atoms and the actual ASP model generator, avoiding the current splitting set method. This could be carried out in two ways: (i) The actual implementation of the answer-set semantics is natively integrated in dlvhex module. (ii) The interface mechanism of external atoms is transferred to DLV itself. The first approach would require great efforts and human resources to reach a level of efficiency compared with pure ASP solvers, since they use highly sophisticated and optimized techniques. The second approach would be less elaborate, but eventually depends on the strategies and priorities of the DLV development team.

Another feature that is currently missing in existing answer-set solvers is the possibility of typed constants. Such typing is crucial in the Semantic Web context when for instance a URL must be distinguished from a string. Currently, this is not possible, because in DLV, a URI has to be quoted. The usage of the RDF plugin has shown that without typing facilities in the logic program, RDF reasoning is seriously compromised, and thus we plan to include typing facilities in dlvhex.

Appendix

A Proofs

The proofs of the theorems in Subsections 3.7.1 and 3.7.2 are lengthy and therefore given here instead of being included in the main text.

Proof of Theorem 3.7.1. We prove the upper complexity bounds for stratified and general dl-programs, and the lower bounds for positive and general dl-programs.

In the stratified case, by Theorem 3.4.3, KB has a strong answer set iff KB is consistent. By Theorem 3.6.6, the latter is equivalent to $M_k \neq HB_P$, where M_k is defined by (a sequence of) fixpoint iterations and can be computed in exponential time. Hence, deciding whether KB has a strong answer set is in EXP. As for deciding whether KB has a weak answer set, we explore (one by one) the exponentially many possible inputs of the dl-atoms in $ground(P)$. For each input, evaluating the dl-atoms and removing them from $ground(P)$ is feasible in exponential time. Since we are then left with an ordinary stratified program KB', by Theorem 3.4.7, we try to compute $M_{KB'}$ by (a sequence of) fixpoint iterations, and check compliance with the inputs of the dl-atoms, which can both be done in exponential time. In summary, deciding whether KB has a weak answer set is also in EXP.

In the general case, we can guess an (exponential size) interpretation $I \subseteq HB_P$ and compute the transform sP_L^I (resp., wP_L^I). Since evaluating all dl-atoms in $ground(P)$ and removing (i) all default-negated literals and dl-atoms, and (ii) all not necessarily monotonic (resp., all) other dl-atoms from $ground(P)$ is feasible in exponential time, computing the transform sP_L^I (resp., wP_L^I) is also feasible in exponential time. Since we are then left with a positive KB', we try to compute $M_{KB'}$ by a fixpoint iteration, and check compliance with the guessed I, which can both be done in exponential time. In summary, deciding whether KB has a strong (resp., weak) answer set can be done in nondeterministic exponential time.

Hardness for EXP of deciding answer set existence in the positive case holds by a reduction from the EXP-complete problem of deciding whether a description logic knowledge base L in $\mathcal{SHIF}(\mathbf{D})$ is satisfiable, since the latter is equivalent to the positive dl-program $KB = (L, \{\neg p \leftarrow, p \leftarrow DL[;\top \sqsubseteq \bot]()\})$, where p is a fresh propositional symbol, having a strong answer set, which is by Theorems 3.4.3, 3.4.6, and 3.4.8 in turn equivalent to KB having a weak answer set.

Hardness for NEXP of deciding answer set existence in the general case follows immediately from Theorems 3.4.1 and 3.4.5, and the fact that deciding whether an ordinary normal program has an answer set is NEXP-complete [Dantsin et al., 2001]. □

Proof of Theorem 3.7.2. For the proof of this Theorem (and also of Theorem 3.7.4 following below), we recall the concept of a domino system, which is defined as follows. A *domino system* $\mathcal{D} = (D, H, V)$ consists of a finite nonempty set D of *tiles* and two relations $H, V \subseteq D \times D$ expressing horizontal and vertical compatibility constraints between the

tiles. For positive integers s and t, and a word $w = w_0 \dots w_{n-1}$ over D of length $n \leq s$, we say that $\mathcal{D}$ *tiles* the torus $U(s,t) = \{0,1,\dots,s-1\} \times \{0,1,\dots,t-1\}$ with initial condition w iff there exists a mapping $\tau\colon U(s,t) \to D$ such that for all $(x,y) \in U(s,t)$: (i) if $\tau(x,y) = d$ and $\tau((x+1) \bmod s, y) = d'$, then $(d,d') \in H$ (*horizontal constraint*), (ii) if $\tau(x,y) = d$ and $\tau(x,(y+1) \bmod t) = d'$, then $(d,d') \in V$ (*vertical constraint*), and (iii) $\tau(i,0) = w_i$ for all $i \in \{0,\dots,n\}$ (*initial condition*).

We prove the upper complexity bounds for positive and general dl-programs, and the lower bounds for positive and stratified dl-programs.

To prove the NEXP-membership in the positive case, observe that a positive KB has a strong (resp., weak) answer set iff there exists an interpretation I and a subset $S \subseteq \{a \in DL_P \mid I \not\models_L a\}$ such that the ordinary positive program $P_{I,S}$, which is obtained from $\mathit{ground}(P)$ by deleting each rule that contains a dl-atom $a \in S$ and all remaining dl-atoms, has a model included in I. A suitable I and S, along with proofs $I \not\models_L a$ for all $a \in S$, can be guessed and verified in exponential time.

As for the general case, observe first that for each dl-program KB, the number of ground dl-atoms a is polynomial, and every ground dl-atom a has in general exponentially many different concrete inputs I_p (that is, interpretations I_p of its input predicates $p = p_1,\dots,p_m$), but each of these concrete inputs I_p has a polynomial size. Furthermore, notice that during the computation of the canonical model of a positive dl-program by fixpoint iteration, any ground dl-atom a needs to be evaluated only polynomially often, as its input can increase only that many times.

We can thus guess inputs I_p for all dl-atoms, and evaluate them with a NEXP oracle in polynomial time. For the (monotonic) ones remaining in sP_L^I, we can further guess a chain $\emptyset = I_p^0 \subset I_p^1 \subset \dots \subset I_p^k = I_p$, along which their inputs are increased in a fixpoint computation for sP_L^I, and evaluate the dl-atoms on it in polynomial time with a NEXP oracle. We then ask a NEXP oracle if an interpretation I exists which is the answer set of sP_L^I (resp., wP_L^I) compliant with the above inputs (and thus the valuations) of the dl-atoms and such that their inputs increase in the fixpoint computation as in the above chain. This yields the $\mathsf{NP}^{\mathsf{NEXP}} = \mathsf{P}^{\mathsf{NEXP}}$ upper bound.

Hardness for NEXP of deciding (strong or weak) answer set existence in the positive case holds by a reduction from the NEXP-complete problem of deciding whether a description logic knowledge base L in $\mathcal{SHOIN}(\mathbf{D})$ is satisfiable, using the same line of argumentation as in the proof of Theorem 3.7.1.

Hardness for $\mathsf{P}^{\mathsf{NEXP}}$ of deciding answer set existence in the stratified case is proved by a generic reduction from Turing machines M, exploiting the NEXP-hardness proof for $\mathcal{ALCQIO}$ by Tobies [Tobies, 2001]. Informally, the main idea behind the proof is to use a dl-atom to decide the result of the j-th oracle call made by a polynomial-time bounded M with access to a NEXP oracle, where the results of the previous calls are known and input to the dl-atom. By a proper sequence of dl-atom evaluations, the result of M's computation on input v can then be obtained.

More concretely, let M be a polynomial-time bounded deterministic Turing machine with access to a NEXP oracle, and let v be an input for M. Since every oracle call can simulate M's computation on v before that call, once the results of all the previous oracle calls are known, we can assume that the input of every oracle call is given by v and the results of all the previous oracle calls. Since M's computation after all oracle calls can be simulated within an additional oracle call, we can assume that the result of the last oracle call is the result of M's computation on v. Finally, since any input to an oracle call can be enlarged by "dummy" bits, we can assume that the inputs to all oracle calls have all the same length $n = 2 \cdot (k+l)$, where k is the size of v, and $l = p(k)$ is the number of all

oracle calls: We assume that the input to the m+1-th oracle call (with $m \in \{0, \ldots, l-1\}$) has the form

$$v_k\, 1\, v_{k-1}\, 1\, \ldots\, v_1\, 1\, c_0\, 1\, c_1\, 1\, \ldots\, c_{m-1}\, 1\, c_m\, 0\, \ldots\, c_{l-1}\, 0\,,$$

where $v_k, v_{k-1}, \ldots, v_1$ are the symbols of v in reverse order, which are all marked as valid by a subsequent "1", $c_0, c_1, \ldots, c_{m-1}$ are the results of the previous m oracle calls, which are all marked as valid by a subsequent "1", and $c_m, \ldots, c_{l-1}$ are "dummy" bits, which are all marked as invalid by a subsequent "0".

Let M' be a nondeterministic Turing machine with time- (and so space-) bound 2^n, deciding a NEXP-complete language $\mathcal{L}(M')$ over the alphabet Σ (consisting of 0, 1, and the blank symbol " "). By [Börger et al., 1997], Theorem 6.1.2, there exists a domino system $\mathcal{D} = (D, H, V)$ and a linear-time reduction *trans* that takes any input $b \in \Sigma^\star$ to a word $w \in D^\star$ with $|b| = n = |w|$ such that M' accepts b iff $\mathcal{D}$ tiles the torus $U(2^{n+1}, 2^{n+1})$ with initial condition w. Here, D is defined as the set of all triples of elements from $\Sigma \cup Q \times \Sigma \cup \{\#, e\}$, where Q is the set of all states of M', and $\#$ and e are two fresh symbols. Moreover, the linear-time reduction *trans* that transforms any string $b = b_0\, b_1 \ldots b_{n-1}$ over Σ into a string $w = w_0 w_1 \ldots w_{n-1}$ over D (of the same length) is defined as follows (where q_0 is the start state of M'):

$$\begin{array}{rcl} w_0 & = & (\#, (q_0, b_0), b_1), \\ w_1 & = & ((q_0, b_0), b_1, b_2), \\ w_2 & = & (b_1, b_2, b_3), \\ & \vdots & \\ w_{n-1} & = & (b_{n-2}, b_{n-1}, \text{“ ”})\,. \end{array}$$

As shown in [Tobies, 2001], Lemma 5.18 and Corollary 5.22, for domino systems $\mathcal{D} = (D, H, V)$ and initial conditions $w = w_0 \ldots w_{n-1}$, there exist description logic knowledge bases L_n, $L_\mathcal{D}$, and L_w in $\mathcal{SHOIN}(\mathbf{D})$ (which can be constructed in polynomial time in n from $\mathcal{D}$, and w) such that $L_n \cup L_\mathcal{D} \cup L_w$ is satisfiable iff $\mathcal{D}$ tiles $U(2^{n+1}, 2^{n+1})$ with initial condition w. Informally, L_n encodes the torus $U(2^{n+1}, 2^{n+1})$, and $L_\mathcal{D}$ represents the domino system $\mathcal{D}$, while L_w encodes the initial condition w. Intuitively, the elements of the torus $U(2^{n+1}, 2^{n+1})$ are encoded by objects, and any mapping $\tau\colon U(2^{n+1}, 2^{n+1}) \to D$ satisfying the compatibility constraints is encoded by the membership of these objects to concepts C_d with $d \in D$, while L_w explicitly represents some of such memberships to encode the initial condition w. More precisely, L_w has the form $\{C_{i,0} \sqsubseteq C_{w_i} \mid i \in \{0, 1, \ldots, n-1\}\}$, where every $C_{i,0}$ is a concept containing exactly the object representing $(i, 0) \in U(2^{n+1}, 2^{n+1})$.

Let the stratified dl-program $KB = (L, P)$ now be defined as follows:

$$\begin{array}{rcl} L & = & L_n \cup L_\mathcal{D} \cup \{C_{i,0} \sqcap S_{i,d} \sqsubseteq C_d \mid i \in \{0, 1, \ldots, n-1\},\, d \in D\} \cup \\ & & \{C_{i,0}(o_i) \mid i \in \{0, 1, \ldots, n-1\}\}\,, \\ P & = & \{\neg b^l_{2l-2}(0) \leftarrow\} \cup \bigcup_{j=0}^{l} P^j\,, \end{array}$$

where $P^j = P^j_v \cup P^j_q \cup P^j_{w \leftarrow b} \cup P^j_{s \leftarrow w}$ for every $j \in \{0, \ldots, l\}$. Informally, every set of dl-rules P^j generates the input of the j+1-th oracle call, where the input of the "dummy" l+1-th oracle call contains the result of the l-th (i.e., the last) oracle call. More concretely, the bitstring $a_{-2k} \cdots a_{2l-1}$ is the input of the j+1-th oracle call iff $b^j_{-2k}(a_{-2k}), \ldots, b^j_{2l-1}(a_{2l-1})$ are in the canonical model of KB. Every $P^j = P^j_v \cup P^j_q \cup P^j_{w \leftarrow b} \cup P^j_{s \leftarrow w}$ with $j \in \{0, \ldots, l\}$ is defined as follows:

1. P_v^0 writes v into the input of the first oracle call, and every P_v^j with $j \in \{1, \ldots, l\}$ copies v into the input of the j+1-th oracle call:

$$\begin{array}{rcl} P_v^0 & = & \{b_{-2i}^0(v_i) \leftarrow \mid i \in \{1, \ldots, k\}\} \cup \{b_{-2i+1}^0(1) \leftarrow \mid i \in \{1, \ldots, k\}\}, \\ P_v^j & = & \{b_{-i}^j(x) \leftarrow b_{-i}^{j-1}(x) \mid i \in \{1, \ldots, 2k\}\}. \end{array}$$

2. P_q^0 initializes the rest of the input of the first oracle call with "dummy" bits, and every P_q^j with $j \in \{1, \ldots, l\}$ writes the result of the j-th oracle call into the input of the j+1-th oracle call and carries over all the other result and dummy bits from the input of the j-th oracle call:

$$\begin{array}{rcl} P_q^0 & = & \{b_i^0(0) \leftarrow \mid i \in \{0, \ldots, 2l-1\}, \\ P_q^j & = & \{b_i^j(x) \leftarrow b_i^{j-1}(x) \mid i \in \{0, \ldots, 2l-1\},\ i \notin \{2j-2, 2j-1\}\} \cup \\ & & \{b_{2j-2}^j(0) \leftarrow DL[\forall i, d\colon S_{i,d} \uplus s_{i,d}^{j-1}\,;\, \top \sqsubseteq \bot](); \\ & & \ b_{2j-2}^j(1) \leftarrow not\, b_{2j-2}^j(0); \\ & & \ b_{2j-1}^j(1) \leftarrow\}. \end{array}$$

3. Every $P_{w \leftarrow b}^j$ with $j \in \{0, \ldots, l\}$ realizes the above-mentioned linear-time reduction *trans*, which transforms any input b^j of the Turing machine M into an initial condition w^j of the same length of M's domino system $\mathcal{D}$. That is, $P_{w \leftarrow b}^j$ is a positive program consisting of $(n-2) \cdot 8 + 2 \cdot 4$ ground rules.

4. Every $P_{s \leftarrow w}^j$ with $j \in \{0, \ldots, l\}$ transforms the initial condition w^j of $\mathcal{D}$ into an input s^j to the j+1-th dl-atom via the predicates $s_{i,d}^j$:

$$P_{s \leftarrow w}^j \quad = \quad \{s_{i,d}^j(o_i) \leftarrow w_i^j(d) \mid i \in \{0, 1, \ldots, n-1\},\ d \in D\}.$$

Observe then that M accepts v iff the last oracle call returns "yes". The latter is equivalent to $b_{2l-2}^l(1)$ (and not $b_{2l-2}^l(0)$) being derived from KB, which is in turn equivalent to KB having a strong (resp., weak) answer set. In summary, M accepts v iff KB has a strong (resp., weak) answer set. □

Proof of Theorem 3.7.3. We prove the upper complexity bounds for stratified and general dl-programs, and the lower bounds for positive and general dl-programs.

As for the upper complexity bounds, deciding whether l belongs to every (resp., some) strong or weak answer set of $KB = (L, P)$ can be reduced to the complement of answer set existence (resp., answer set existence itself) by adding to P the two rules $p \leftarrow l$ and $\neg p \leftarrow l$ (resp., the two rules $p \leftarrow not\, l$ and $\neg p \leftarrow not\, l$), where p is a fresh propositional symbol. Adding these rules does not change KB's property of being stratified or general. By Theorem 3.7.1, answer set existence is in EXP in the stratified case and in NEXP in the general case. Thus, deciding whether l belongs to every (resp., some) strong or weak answer set of KB is in EXP when KB is stratified, and in coNEXP (resp., NEXP) when KB is a general dl-program.

The lower complexity bounds hold by a reduction from the complement of answer set existence (resp., answer set existence itself), since a dl-program $KB = (L, P)$ has no (resp., some) strong or weak answer set iff the classical literal p belongs to every (resp., some) strong or weak answer set of $KB' = (L, P \cup \{\neg p \leftarrow\})$ (resp., $KB' = (L, P \cup \{p \leftarrow\})$, where p is a fresh propositional symbol. Adding the rule $\neg p \leftarrow$ (resp., $p \leftarrow$) does not change KB's property of being positive or general. By Theorem 3.7.1, answer set existence is hard for

EXP in the positive case and hard for NEXP in the general case. Thus, deciding whether l belongs to every (resp., some) strong or weak answer set of KB is hard for EXP when KB is positive and hard for coNEXP (resp., NEXP) when KB is a general dl-program. □

Proof of Theorem 3.7.4. We prove the upper complexity bounds for positive and general dl-programs, and the lower bounds for positive and stratified dl-programs.

We first prove the upper complexity bounds for all above cases except for brave reasoning from positive dl-programs. Using the same line of argumentation as in the proof of Theorem 3.7.3, deciding whether l belongs to every (resp., some) strong or weak answer set of $KB = (L, P)$ can be reduced to the complement of answer set existence (resp., answer set existence itself) by adding to P the two rules $p \leftarrow l$ and $\neg p \leftarrow l$ (resp., the two rules $p \leftarrow not\, l$ and $\neg p \leftarrow not\, l$), where p is a fresh propositional symbol. In all cases except for brave reasoning from positive dl-programs, adding these rules does not change KB's property of being positive or general. By Theorem 3.7.2, answer set existence is in NEXP in the positive case and in $\mathsf{P}^{\mathsf{NEXP}}$ in the general case. Thus, deciding whether l belongs to every strong or weak answer set of KB is in coNEXP when KB is positive, and in $\mathsf{P}^{\mathsf{NEXP}}$ when KB is a general dl-program. Furthermore, deciding whether l belongs to some strong or weak answer set of KB is in $\mathsf{P}^{\mathsf{NEXP}}$ when KB is a general dl-program.

Membership in $\mathsf{P}^{\mathsf{NEXP}}$ of brave reasoning under the weak answer set semantics in the positive case follows from the membership in $\mathsf{P}^{\mathsf{NEXP}}$ of deciding weak answer set existence in the stratified case, since (as argued above) a classical literal $l \in HB_P$ belongs to some weak answer set of the positive dl-program $KB = (L, P)$ iff the stratified dl-program $KB' = (L, P \cup \{p \leftarrow not\, l, \neg p \leftarrow not\, l\})$, where p is a fresh propositional symbol, has a weak answer set.

As for the membership in D^{exp} of brave reasoning under the strong answer set semantics in the positive case, observe first that a classical literal $l \in HB_P$ belongs to some strong answer set of the positive dl-program KB iff (i) KB has some strong answer set, and (ii) KB has no strong answer set I with $l \notin I$. The latter is equivalent to: (i) there exists an interpretation I and a subset $S \subseteq \{a \in DL_P \mid I \not\models_L a\}$ such that the ordinary positive program $P_{I,S}$, which is obtained from $ground(P)$ by deleting each rule that contains a dl-atom $a \in S$ and all remaining dl-atoms, has a model included in I, and (ii) there exists no interpretation I with $l \notin I$ and subset $S \subseteq \{a \in DL_P \mid I \not\models_L a\}$ such that the ordinary positive program $P_{I,S}$, which is obtained from $ground(P)$ by deleting each rule that contains a dl-atom $a \in S$ and all remaining dl-atoms, has a model included in I. As argued in the proof of Theorem 3.7.2, (i) and (ii) are in NEXP and coNEXP, respectively. This shows that brave reasoning in the positive case under the strong answer set semantics is in D^{exp}.

The lower complexity bounds for all above cases except for brave reasoning from positive dl-programs hold by a reduction from answer set non-existence (resp., existence), using the same line of argumentation as in the proof of Theorem 3.7.3, since a dl-program $KB = (L, P)$ has no (resp., some) strong or weak answer set iff the classical literal p belongs to every (resp., some) strong or weak answer set of $KB' = (L, P \cup \{\neg p \leftarrow\})$ (resp., $KB' = (L, P \cup \{p \leftarrow\})$, where p is a fresh propositional symbol. Adding the rule $\neg p \leftarrow$ (resp., $p \leftarrow$) does not change KB's property of being positive or stratified. By Theorem 3.7.2, answer set existence is NEXP-hard in the positive case and $\mathsf{P}^{\mathsf{NEXP}}$-hard in the stratified case. Hence, deciding whether l belongs to every strong or weak answer set of KB is coNEXP-hard when KB is positive and $\mathsf{P}^{\mathsf{NEXP}}$-hard when KB is stratified. Moreover, deciding whether l is in some strong or weak answer set of KB is $\mathsf{P}^{\mathsf{NEXP}}$-hard when KB is stratified.

Hardness for D^{exp} of brave reasoning under the strong answer set semantics in the positive case holds by a reduction from a D^{exp}-hard problem involving domino systems.

More concretely, by a slight adaptation of the proof of Corollary 5.14 in [Tobies, 2001], it can be shown that there exists a domino system $\mathcal{D} = (D, H, V)$ such that the following problem is hard for D^{exp}:

($\star$) Given two initial conditions $v = v_0 \dots v_{n-1}$ and $w = w_0 \dots w_{n-1}$ over D of length n, decide whether (1) $\mathcal{D}$ tiles the torus $U(2^{n+1}, 2^{n+1})$ with initial condition v, and (2) $\mathcal{D}$ does not tile the torus $U(2^{n+1}, 2^{n+1})$ with initial condition w.

We now reduce ($\star$) to brave reasoning under the strong answer set semantics in the positive case. As shown in [Tobies, 2001], Lemma 5.18 and Corollary 5.22, for domino systems $\mathcal{D} = (D, H, V)$ and initial conditions $v = v_0 \dots v_{n-1}$ and $w = w_0 \dots w_{n-1}$, there exist description logic knowledge bases L_n, $L_\mathcal{D}$, L_v, and L_w in $\mathcal{SHOIN}(\mathbf{D})$ (which can be constructed in polynomial time in n from $\mathcal{D}$, v, and w) such that (a) $L_n \cup L_\mathcal{D} \cup L_v$ is satisfiable iff $\mathcal{D}$ tiles the torus $U(2^{n+1}, 2^{n+1})$ with initial condition v, and (b) $L_n \cup L_\mathcal{D} \cup L_w$ is unsatisfiable iff $\mathcal{D}$ does not tile $U(2^{n+1}, 2^{n+1})$ with initial condition w. Informally, L_n encodes the torus $U(2^{n+1}, 2^{n+1})$, and $L_\mathcal{D}$ represents the domino system $\mathcal{D}$, while L_v and L_w encode the initial conditions v and w, respectively. Intuitively, the elements of the torus $U(2^{n+1}, 2^{n+1})$ are encoded by objects, and any mapping $\tau\colon U(2^{n+1}, 2^{n+1}) \to D$ satisfying the compatibility constraints is encoded by the membership of these objects to concepts C_d with $d \in D$, while L_v and L_w explicitly represent some of such memberships to encode the initial conditions v and w, respectively. More concretely, L_v and L_w are of the form $\{C_{i,0} \sqsubseteq C_{v_i} \mid i \in \{0, 1, \dots, n-1\}\}$ and $\{C_{i,0} \sqsubseteq C_{w_i} \mid i \in \{0, 1, \dots, n-1\}\}$, respectively, where every $C_{i,0}$ is a concept containing exactly the object representing the element $(i, 0) \in U(2^{n+1}, 2^{n+1})$. Let the dl-program $KB = (L, P)$ be defined by:

$$
\begin{array}{rcl}
L & = & L_n \cup L_\mathcal{D} \cup \{C_{i,0} \sqcap S_{i,d} \sqsubseteq C_d \mid i \in \{0, 1, \dots, n-1\},\, d \in D\} \cup \\
 & & \{C_{i,0}(o_i) \mid i \in \{0, 1, \dots, n-1\}\}\,, \\
P & = & \{\neg p \leftarrow,\, p \leftarrow DL[\forall i, d\colon S_{i,d} \uplus s_{i,d}\,; \top \sqsubseteq \bot]()\} \cup \\
 & & \{s_{i,d}(o_i) \leftarrow \mid i \in \{0, 1, \dots, n-1\}, d \in D, v_i = d\} \cup \\
 & & \{q \leftarrow DL[\forall i, d\colon S_{i,d} \uplus s'_{i,d}\,; \top \sqsubseteq \bot]()\} \cup \\
 & & \{s'_{i,d}(o_i) \leftarrow \mid i \in \{0, 1, \dots, n-1\}, d \in D, w_i = d\}\,.
\end{array}
$$

Observe that the dl-program KB is positive. Furthermore, KB has a strong answer set iff (1) $L_n \cup L_\mathcal{D} \cup L_v$ is satisfiable, and the strong answer set of KB contains q iff (2) $L_n \cup L_\mathcal{D} \cup L_w$ is unsatisfiable. That is, q belongs to some strong answer set of KB iff (1) $\mathcal{D}$ tiles the torus $U(2^{n+1}, 2^{n+1})$ with initial condition v, and (2) $\mathcal{D}$ does not tile the torus $U(2^{n+1}, 2^{n+1})$ with initial condition w.

Hardness for $\mathsf{P}^{\mathsf{NEXP}}$ of brave reasoning under the weak answer set semantics in the positive case is proved by a generic reduction from Turing machines. The proof is similar to the proof of $\mathsf{P}^{\mathsf{NEXP}}$-hardness of deciding strong (resp., weak) answer set existence in the stratified case (in the proof of Theorem 3.7.2). The main difference that must be taken into account in the construction is that rather than deciding whether a stratified dl-program has a strong (resp., weak) answer set, we now decide whether a literal holds in some weak answer set of a positive dl-program. Intuitively, we use a set of weak answer sets for guessing the outcomes of all oracle calls, and a literal q in one of these weak answers to identify the correct guess.

More concretely, let M be a polynomial-time bounded deterministic Turing machine with access to a NEXP oracle, and let v be an input for M. Let the positive dl-program $KB = (L, P)$ be defined as the stratified dl-program $KB = (L, P)$ in the proof of Theorem 3.7.2, except that we now add the rule

$$
q \leftarrow \mathit{guess\_ok}^1, \dots, \mathit{guess\_ok}^l, \mathit{call\_ok}^1, \dots, \mathit{call\_ok}^l\,, \tag{7.1}
$$

and that every P^j_q, $j \in \{1,\ldots,l\}$, is now defined as $P^j_{q,id} \cup P^j_{q,guess} \cup P^j_{q,call}$, where:

1. Every $P^j_{q,id}$, $j \in \{1,\ldots,l\}$, copies all the persisting result and dummy bits from the input of the j-th oracle call into the input of the j+1-th oracle call:

$$P^j_{q,id} \;=\; \{b^j_i(x) \leftarrow b^{j-1}_i(x) \mid i \in \{0,\ldots,2l{-}1\},\, i \notin \{2j{-}2, 2j{-}1\}\}\,.$$

2. Every $P^j_{q,guess}$, $j \in \{1,\ldots,l\}$, allows for guessing the outcome of the j-th oracle call, that is, exactly one fact among $b^j_{2j-2}(0)$ and $b^j_{2j-2}(1)$. The guess is verified through the predicate $guess\_ok^j$, which should evaluate to true:

$$\begin{aligned} P^j_{q,guess} \;=\; \{&b^j_{2j-2}(1) \leftarrow DL[B^j \uplus b^j_{2j-2}; B^j](1);\\ &b^j_{2j-2}(0) \leftarrow DL[B^j \uplus b^j_{2j-2}; B^j](0);\\ &\neg b^j_{2j-2}(1) \leftarrow b^j_{2j-2}(0);\\ &guess\_ok^j \leftarrow b^j_{2j-2}(0);\\ &guess\_ok^j \leftarrow b^j_{2j-2}(1)\}\,. \end{aligned}$$

3. Every $P^j_{q,call}$, $j \in \{1,\ldots,l\}$, allows for choosing among the two possible outcomes of the j-th oracle call exactly the one that matches the result of the actual outcome. That is, $call\_ok^j$ is true iff either (a) $b^j_{2j-2}(0)$ holds and the actual outcome is "No" or (b) $b^j_{2j-2}(1)$ holds and the actual outcome is "Yes":

$$\begin{aligned} P^j_{q,call} \;=\; \{&\neg b^j_{2j-2}(1) \leftarrow DL[\forall i,d\colon S_{i,d} \uplus s^{j-1}_{i,d}\,; \top \sqsubseteq \bot]();\\ &call\_ok^j \leftarrow b^j_{2j-2}(0), DL[\forall i,d\colon S_{i,d} \uplus s^{j-1}_{i,d}\,; \top \sqsubseteq \bot]();\\ &call\_ok^j \leftarrow b^j_{2j-2}(1);\\ &b^j_{2j-1}(1) \leftarrow\}\,. \end{aligned}$$

Hence, M accepts v iff (i) the last oracle call returns "yes" and (ii) the $b^j_{2j-2}(x)$'s with $j \in \{1,\ldots,l\}$ are a correct guess that matches the actual outcomes of the oracle calls. The latter is equivalent to the existence of a weak answer set of KB that contains all $guess\_ok^j$ and $call\_ok^j$ with $j \in \{1,\ldots,l\}$, or, equivalently, that contains q. In summary, M accepts v iff q holds in some weak answer set of KB. □

B String Plugin Source Code

Here, we exemplarily list the implementation of two external atoms of the string-plugin.

```
#include "dlvhex/PluginInterface.h"
#include <string>
#include <sstream>

class ConcatAtom : public PluginAtom
{
public:

    ConcatAtom()
    {
        // first string or int
        addInputConstant();

        // second string or int
        addInputConstant();

        setOutputArity(1);
    }

    virtual void
    retrieve(const Query& query, Answer& answer) throw (PluginError)
    {
        std::stringstream in1, in2;

        Term s1 = query.getInputTuple()[0];
        Term s2 = query.getInputTuple()[1];

        bool smaller = false;

        if (s1.isInt())
            in1 << s1.getInt();
        else if (s1.isString())
            in1 << s1.getUnquotedString();
        else
            throw PluginError("Wrong input argument type");

        if (s2.isInt())
            in2 << s2.getInt();
        else if (s2.isString())
            in2 << s2.getUnquotedString();
        else
            throw PluginError("Wrong input argument type");

        Tuple out;

        out.push_back(Term(std::string(in1.str() + in2.str()), 1));
```

```
        answer.addTuple(out);
    }
};

class strstrAtom : public PluginAtom
{
public:

    strstrAtom()
    {
        // haystack
        addInputConstant();

        // needle
        addInputConstant();

        setOutputArity(0);
    }

    virtual void
    retrieve(const Query& query, Answer& answer) throw (PluginError)
    {
        std::string in1;

        std::stringstream inss;

        Term s1 = query.getInputTuple()[0];
        Term s2 = query.getInputTuple()[1];

        if (!s1.isString())
            throw PluginError("Wrong input argument type");

        in1 = s1.getUnquotedString();

        if (s2.isString())
            inss << s2.getUnquotedString();
        else if (s2.isInt())
            inss << s2;
        else
            throw PluginError("Wrong input argument type");

        std::string in2(inss.str());

        std::transform(in1.begin(), in1.end(), in1.begin(),
                       (int(*)(int))std::tolower);
        std::transform(in2.begin(), in2.end(), in2.begin(),
                       (int(*)(int))std::tolower);
```

```
        Tuple out;

        std::string::size_type pos = in1.find(in2, 0);

        if (pos != std::string::npos)
            answer.addTuple(out);
    }
};

class StringPlugin : public PluginInterface
{
public:

    // register all atoms of this plugin:
    virtual void
    getAtoms(AtomFunctionMap& a)
    {
        a["sha1sum"] = new ShaAtom;
        a["split"] = new SplitAtom;
        a["cmp"] = new CmpAtom;
        a["concat"] = new ConcatAtom;
        a["strstr"] = new strstrAtom;
    }
};


//
// now instantiate the plugin
//
StringPlugin theStringPlugin;

//
// and let it be loaded by dlvhex!
//
extern "C"
StringPlugin*
PLUGINIMPORTFUNCTION()
{
    return &theStringPlugin;
}
```

C Policy Example Source Code

Server policy:

```
% if resource is public, no authentication is necessary
allow(download,Resource) :- public(Resource).

% user may download if she has a subscription and is authenticated
allow(download,Resource) :- authenticated(User),
                            hasSubscription(User,Subscription),
                            availableFor(Resource,Subscription).

% user may download if she has paid and is authenticated
allow(download,Resource) :- authenticated(User),
                            paid(User,Resource).

% user is authenticated, if she has a valid credential
authenticated(User) :- valid(Credential),
                       attr(Credential,name,User).

% other authentication rules are possible, e.g., by password ...

% a selected credential is valid, if its type is trusted
valid(Credential) :- selectedCred(Credential),
                     attr(Credential,type,T),
                     attr(Credential,issuer,CA),
                     isa(T,id),
                     trustedFor(CA,T).

% types that are ids
isa(id,id).
isa(ssn,id).
isa(passport,id).
isa(driving_license,id).
```

Client example:

```
hasSubscription("John Doe",law_basic).
hasSubscription("John Doe",computer_basic).

availableFor("paper01234.pdf",computer_full).

resource("paper01234.pdf").

trustedFor("Open University",id).
trustedFor("Visa",id).
trustedFor("UK Government",ssn).

credential(cr01).
attr(cr01,type,id).
attr(cr01,name,"John Doe").
attr(cr01,issuer,"Open University").

credential(cr02).
```

```
attr(cr02,type,ssn).
attr(cr02,name,"John Doe").
attr(cr02,issuer,"UK Government").

credential(cr03).
attr(cr03,type,id).
attr(cr03,name,"John Doe").
attr(cr03,issuer,"Visa").

credSens(cr01,"1").
credSens(cr02,"2").
credSens(cr03,"4").
```

Optimization rules:

```
% open a search space

selectedCred(X) v -selectedCred(X) :- credential(X).

sens(C,S) :- selectedCred(C), credSens(C,S).

% remove models that don't accomplish the goal

:- not allow(download,R), resource(R).

% compute model sensitivity

sensitivity(S) :- &policy[sens](S).

% select least sensitive model

:~ sensitivity(S). [S:1]
```

Bibliography

Güray Alsaç and Chitta Baral. Reasoning in Description Logics using Declarative Logic Programming. Technical report, Department of Computer Science and Engineering, Arizona State University, 2001.

Anastasia Analyti, Grigoris Antoniou, Carlos Viegas Damásio, and Gerd Wagner. Stable Model Theory for Extended RDF Ontologies. In Yolanda Gil, Enrico Motta, Richard V. Benjamins, and Mark Musen, editors, *Proceedings of the 4th International Semantic Web Conference (ISWC 2005), Galway, Ireland*, volume 3729 of *Lecture Notes in Computer Science (LNCS)*, pages 21–36. Springer, 2005.

Jürgen Angele, Harold Boley, Jos de Bruijn, Dieter Fensel, Pascal Hitzler, Michael Kifer, Reto Krummenacher, Holger Lausen, Axel Polleres, and Rudi Studer. Web Rule Language (WRL), September 2005. URL http://www.w3.org/Submission/WRL. W3C Member Submission.

Grigoris Antoniou. Nonmonotonic Rule Systems on Top of Ontology Layers. In Ian Horrocks and James Hendler, editors, *Proceedings of the First International Semantic Web Conference (ISWC 2002), Sardinia, Italy*, volume 2342 of *Lecture Notes in Computer Science (LNCS)*, pages 394–398. Springer, 2002.

Grigoris Antoniou, Matteo Baldoni, Cristina Baroglio, Robert Baumgartner, François Bry, Thomas Eiter, Nicola Henze, Marcus Herzog, Wolfgang May, Viviana Patti, Sebastian Schaffert, Roman Schindlauer, and Hans Tompits. Reasoning Methods for Personalization on the Semantic Web. *Annals of Mathematics, Computing and Teleinformatics*, 2 (1):1–24, 2004. Invited paper.

Grigoris Antoniou, Carlos Viegas Damásio, Benjamin Grosof, Ian Horrocks, Michael Kifer, Jan Maluszynski, and Peter F. Patel-Schneider. Combining Rules and Ontologies: A survey. Technical Report IST506779/Linköping/I3-D3/D/PU/a1, Linköping University, February 2005. IST-2004-506779 REWERSE Deliverable I3-D3. http://rewerse.net/publications/.

Krzysztof R. Apt, Howard A. Blair, and Adrian Walker. Towards a Theory of Declarative Knowledge. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, Washington DC, 1988.

Franz Baader and Bernhard Hollunder. Embedding Defaults into Terminological Representation Systems. *Journal of Automated Reasoning*, 14(1):149–180, 1995.

Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

Yuliya Babovich, Esra Erdem, and Vladimir Lifschitz. Fages' theorem and answer set programming. In Chitta Baral and Miroslaw Truszczyński, editors, *Proceedings of the 8th International Workshop on Non-Monotonic Reasoning, Breckenridge, CO, USA*. `arXiv.org` e-Print archive, April 2000.

Chitta Baral and V. S. Subrahmanian. Dualities Between Alternative Semantics for Logic Programming and Nonmonotonic Reasoning. *Journal of Automated Reasoning*, 10(3): 399–420, 1993.

Peter Baumgartner, Ulrich Furbach, and Bernd Thomas. Model-Based Deduction for Knowledge Representation. In *Proceedings of the 17th Workshop on Logic Programming (WLP'02), Dresden, Germany*, pages 156–166. Technische Universität Dresden, Fakultät für Informatik, 2002. Technische Berichte TUD-F103-03.

Sean Bechhofer, Frank van Harmelen, James Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn A. Stein. OWL Web Ontology Language Reference. W3C recommendation, W3C, February 2004. URL http://www.w3.org/TR/owl-ref/.

Rachel Ben-Eliyahu and Rina Dechter. Propositional Semantics for Disjunctive Logic Programs. *Annals of Mathematics and Artificial Intelligence*, 12:53–87, 1994.

Tim Berners-Lee. *Weaving the Web*. Harper, San Francisco, CA, 1999.

Tim Berners-Lee. Semantic Web Road Map, 1998. URL http://www.w3.org/DesignIssues/Semantic.html.

Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, May 2001.

Jürgen Bock. Ontology Merging using Answer Set Programming and WordNet. Honours Thesis, School of Information and Communication Technology, Griffith University, Brisbane, Australia, October 2006.

Harold Boley, Said Tabet, and Gerd Wagner. Design Rationale of RuleML: A Markup Language for Semantic Web Rules. In Isabel F. Cruz, Stefan Decker, Jérôme Euzenat, and Deborah L. McGuinness, editors, *Proceedings of the First Semantic Web Working Symposium (SWWS'01), Stanford University, CA, USA*, pages 381–401, 2001.

Piero A. Bonatti, Thomas Eiter, and Marco Faella. Advanced Policy Queries. REWERSE Deliverable I2-D6, Dipartimento di Scienze Fisiche - Sezione di Informatica, University of Naples "Federico II", 2006. URL http://idefix.pms.ifi.lmu.de:8080/rewerse/index.html#REWERSE-DEL-2006-I2-D6.

Egon Börger, Erich Grädel, and Yuri Gurevich. *The Classical Decision Problem*. Perspectives in Mathematical Logic. Springer, 1997.

Ronald J. Brachman and Hector J. Levesque, editors. *Readings in Knowledge Representation*. Morgan Kaufmann, San Francisco, CA, USA, 1985. ISBN 0-934613-01-X.

Dan Brickley and Ramanathan V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. W3C Recommendation, World Wide Web Consortium, February 2004. URL http://www.w3.org/TR/rdf-schema/.

Francesco Buccafurri, Nicola Leone, and Pasquale Rullo. Strong and Weak Constraints in Disjunctive Datalog. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'97), Dagstuhl, Germany*, number 1265 in Lecture Notes in Computer Science (LNCS), pages 2–17. Springer, 1997.

Francesco Buccafurri, Nicola Leone, and Pasquale Rullo. Enhancing Disjunctive Datalog by Constraints. *IEEE Transactions on Knowledge and Data Engineering*, 12(5):845–860, 2000.

Francesco Buccafurri, Wolfgang Faber, and Nicola Leone. Disjunctive Logic Programs with Inheritance. *Journal of the Theory and Practice of Logic Programming*, 2(3):293–321, 2002.

Marco Cadoli. The Complexity of Model Checking for Circumscriptive Formulae. *Information Processing Letters*, 44(3):113–118, November 1992.

Marco Cadoli and Maurizio Lenzerini. The Complexity of Propositional Closed World Reasoning and Circumscription. *Journal of Computer and System Sciences*, 48(2):255–310, 1994.

Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. A Framework for Ontology Integration. In Isabel F. Cruz, Stefan Decker, Jérôme Euzenat, and Deborah L. McGuinness, editors, *Proceedings of the First Semantic Web Working Symposium (SWWS'01), Stanford University, CA, USA*, pages 303–316, 2001.

Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and Expressive Power of Logic Programming. *ACM Computing Surveys*, 33(3):374–425, 2001.

James P. Delgrande, Torsten Schaub, and Hans Tompits. plp: A Generic Compiler for Ordered Logic Programs. In Thomas Eiter, Wolfgang Faber, and Miroslaw Truszczyński, editors, *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2001), Vienna, Austria*, volume 2173 of *Lecture Notes in Computer Science (LNCS)*, pages 411–415. Springer, 2001.

Tina Dell'Armi, Wolfgang Faber, Giuseppe Ielpa, Nicola Leone, and Gerald Pfeifer. Aggregate Functions in Disjunctive Logic Programming: Semantics, Complexity, and Implementation in DLV. In Georg Gottlob and Toby Walsh, editors, *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI 2003), Acapulco, Mexico*, pages 847–852. Morgan Kaufmann, August 2003.

Jürgen Dix. Semantics of Logic Programs: Their Intuitions and Formal Properties. An Overview. In Andre Fuhrmann and Hans Rott, editors, *Logic, Action and Information. Proceedings Konstanz Colloquium in Logic and Information (LogIn 1992), University of Konstanz, Germany*, pages 241–329. DeGruyter, 1995.

Francesco M. Donini, Maurizio Lenzerini, Daniele Nardi, and Andrea Schaerf. $\mathcal{AL}$-log: Integrating datalog and description logics. *Journal of Intelligent Information Systems (JIIS)*, 10(3):227–252, 1998.

Thomas Eiter and Georg Gottlob. Identifying the Minimal Transversals of a Hypergraph and Related Problems. *SIAM Journal on Computing*, 24(6):1278–1304, December 1995.

Thomas Eiter, Georg Gottlob, and Heikki Mannila. Disjunctive Datalog. *ACM Transactions on Database Systems*, 22(3):364–417, September 1997.

Thomas Eiter, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. The KR System DLV: Progress Report, Comparisons, and Benchmarks. In Anthony G. Cohn, Lenhart Schubert, and Stuart C. Shapiro, editors, *Proceedings of the 6th International Conference on Principles of Knowledge Representation and Reasoning (KR-98)*, pages 406–417, June 1998.

Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. The Diagnosis Frontend of the `dlv` System. *The European Journal on Artificial Intelligence (AI Communications)*, 12(1–2):99–111, 1999a.

Thomas Eiter, V. S. Subrahmanian, and George Pick. Heterogeneous Active Agents, I: Semantics. *Artificial Intelligence*, 108(1-2):179–255, 1999b.

Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Declarative Problem-Solving Using the DLV System. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 79–103. Kluwer Academic Publishers, 2000a.

Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Declarative Problem-Solving Using the DLV System. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 79–103. Kluwer Academic Publishers, 2000b.

Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. System Description: The DLV$^{\mathcal{K}}$ Planning System. In Thomas Eiter, Wolfgang Faber, and Miroslaw Truszczyński, editors, *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2001), Vienna, Austria*, volume 2173 of *Lecture Notes in Computer Science (LNCS)*, pages 429–433. Springer, 2001a. System Description (abstract).

Thomas Eiter, Michael Fink, Giuliana Sabbatini, and Hans Tompits. An Update Front-End for Extended Logic Programs. In Thomas Eiter, Wolfgang Faber, and Miroslaw Truszczyński, editors, *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2001), Vienna, Austria*, volume 2173 of *Lecture Notes in Computer Science (LNCS)*, pages 397–401. Springer, 2001b.

Thomas Eiter, Thomas Lukasiewicz, Roman Schindlauer, and Hans Tompits. Combining Answer Set Programming with Description Logics for the Semantic Web. Technical Report INFSYS RR-1843-03-13, Institut für Informationssysteme, TU Wien, 2003. URL http://www.kr.tuwien.ac.at/research/reports/rr0313.pdf.

Thomas Eiter, Thomas Lukasiewicz, Roman Schindlauer, and Hans Tompits. Combining Answer Set Programming with Description Logics for the Semantic Web. In Didier Dubois, Christopher Welty, and Mary-Anne Williams, editors, *Proceedings of the 9th International Conference on Principles of Knowledge Representation and Reasoning (KR 2004), Whistler, British Columbia, Canada*, pages 141–151. AAAI Press, June 2004a.

Thomas Eiter, Thomas Lukasiewicz, Roman Schindlauer, and Hans Tompits. Well-founded Semantics for Description Logic Programs in the Semantic Web. In Grigoris Antoniou and Harold Boley, editors, *Proceedings of the 3rd International Workshop on Rules and Rule Markup Languages for the Semantic Web (RuleML 2004), Hiroshima, Japan*, number 3323 in Lecture Notes in Computer Science (LNCS), pages 81–97. Springer, 2004b.

Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. Nonmonotonic Description Logic Programs: Implementation and Experiments. In Franz Baader and Andrei Voronkov, editors, *Proceedings of the 12th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2004), Montevideo, Uruguay*, number 3452 in LNCS, pages 511–517. Springer, March 2005a.

Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. NLP-DL: A KR System for Coupling Nonmonotonic Logic Programs with Description Logics. In *4th International Semantic Web Conference (ISWC 2005) – Posters Track, Galway, Ireland*, November 2005b. System poster.

Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. DLV-HEX: Dealing with Semantic Web under Answer-Set Programming. In *4th International Semantic Web Conference (ISWC 2005) – Posters Track, Galway, Ireland*, November 2005c. System poster.

Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. A Uniform Integration of Higher-Order Reasoning and External Evaluations in Answer Set Programming. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI 2005), Edinburgh, Scotland, UK*, pages 90–96. Professional Book Center, 2005d.

Thomas Eiter, Giovambattista Ianni, Axel Polleres, Roman Schindlauer, and Hans Tompits. Reasoning with Rules and Ontologies. In Pedro Barahona, François Bry, Enrico Franconi, Ulrike Sattler, and Nicola Henze, editors, *Reasoning Web, Second International Summer School, Lissabon, Portugal, Tutorial Lectures*, number 4126 in Lecture Notes in Computer Science (LNCS), pages 93–127. Springer, September 2006a.

Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. dlvhex: A System for Integrating Multiple Semantics in an Answer-Set Programming Framework. In Michael Fink, Hans Tompits, and Stefan Woltran, editors, *Proceedings of the 20th Workshop on Logic Programming and Constraint Systems (WLP'06), Vienna, Austria*, volume 1843-06-02 of *INFSYS Research Report*, pages 206–210. Technische Universität Wien, Austria, February 2006b. URL http://www.kr.tuwien.ac.at/wlp06/S02-final.pdf.

Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. Towards Efficient Evaluation of HEX Programs. In Jürgen Dix and Anthony Hunter, editors, *Proceedings of the 11th International Workshop on Nonmonotonic Reasoning (NMR-2006), Answer Set Programming Track, Lakeside, UK*, pages 40–46, May 2006c. Available as TR IfI-06-04, Institut für Informatik, TU Clausthal, Germany.

Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. dlvhex: A Tool for Semantic-Web Reasoning under the Answer-Set Semantics. In Axel Polleres, Stefan Decker, Gopal Gupta, and Jos de Bruijn, editors, *Informal Proceedings Workshop on Applications of Logic Programming in the Semantic Web and Semantic Web Services (ALPSWS 2006), at FLOC/ICLP 2006, Seattle, WA, USA*, number 196 in CEUR Workshop Proceedings, pages 33–39, August 2006d. URL http://CEUR-WS.org/Vol-196/.

Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. dlvhex: A Prover for Semantic-Web Reasoning under the Answer-Set Semantics. In *2006 IEEE/WIC/ACM International Conference on Web Intelligence (WI 2006), Hongkong, China*, December 2006e. To appear as a Demonstration-track paper.

Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. Effective Integration of Declarative Rules with external Evaluations for Semantic Web Reasoning. In York Sure and John Domingue, editors, *Proceedings of the 3rd European Conference on Semantic Web (ESWC 2006), Budva, Montenegro*, number 4011 in LNCS, pages 273–287. Springer, June 2006f.

Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Recursive Aggregates in Disjunctive Logic Programs: Semantics and Complexity. In José Júlio Alferes and João Alexandre Leite, editors, *Proceedings of the 9th European Conference on Logics in Artificial Intelligence (JELIA 2004), Lisbon, Portugal*, volume 3229 of *Lecture Notes in Computer Science (LNCS)*, pages 200–212. Springer, 2004.

François Fages. Consistency of Clark's Completion and Existence of Stable Models. *Journal of Methods of Logic in Computer Science*, 1:51–60, 1994.

Dieter Fensel, Frank van Harmelen, Ian Horrocks, Deborah L. McGuinness, and Peter F. Patel-Schneider. OIL: An Ontology Infrastructure for the Semantic Web. *IEEE Intelligent Systems*, 16(2):38–45, 2001.

Dieter Fensel, Wolfgang Wahlster, Henry Lieberman, and James Hendler, editors. *Spinning the Semantic Web: Bringing the World Wide Web to Its Full Potential*. MIT Press, 2002.

Erich Gamma, Richard Helm, and Ralph Johnson. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.

Michael Gelfond and Vladimir Lifschitz. The Stable Model Semantics for Logic Programming. In Robert A. Kowalski and Kenneth Bowen, editors, *Proceedings of the 5th International Conference on Logic Programming*, pages 1070–1080. MIT Press, 1988.

Michael Gelfond and Vladimir Lifschitz. Logic Programs with Classical Negation. In David Warren and Peter Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming, Jerusalem, Israel*, pages 579–597. MIT Press, 1990.

Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3–4):365–386, 1991.

Michael Gelfond, Halina Przymusinska, and Teodor C. Przymusinski. The Extended Closed World Assumption and its Relationship to Parallel Circumscription. In *Proceedings of the 5th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS), March 24-26, 1986, Cambridge, Massachusetts*, pages 133–139. ACM Press, 1986.

Benjamin N. Grosof, Ian Horrocks, Raphael Volz, and Stefan Decker. Description Logic Programs: Combining Logic Programs with Description Logics. In *Proceedings of the 12th International World Wide Web Conference (WWW 2003), Budapest, Hungary*, pages 48–57, 2003.

Volker Haarslev and Ralph Möller. RACER System Description. In Rajeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *Proceedings of the First International Joint Conference on Automated Reasoning (IJCAR 2001), Siena, Italy*, volume 2083 of *Lecture Notes in Computer Science (LNCS)*, pages 701–705. Springer, 2001.

Patrick J. Hayes. The Logic of Frames. In Dieter Metzing, editor, *Frame Conceptions and Text Understanding*, pages 46–61. Walter de Gruyter and Co., Berlin, 1979.

Jeff Heflin and James A. Hendler. Dynamic Ontologies on the Web. In *Proceedings of the 17th National Conference on Artificial Intelligence and 12th Conference on Innovative Applications of Artificial Intelligence*, pages 443–449. AAAI Press / The MIT Press, 2000. ISBN 0-262-51112-6.

Jeff Heflin and Hector Muñoz-Avila. LCW-Based Agent Planning for the Semantic Web. In Adam Pease, Jim Hendler, and Richard Fikes, editors, *Ontologies and the Semantic Web. Papers from the 2002 AAAI Workshop*, pages 63–70. AAAI Press, 2002.

James Hendler. The Semantic Web: KR's Worst Nightmare? In Dieter Fensel, Fausto Giunchiglia, Deborah L. McGuinness, and Mary-Anne Williams, editors, *Proceedings of the 8th International Conference on Principles and Knowledge Representation and Reasoning (KR 2002), Toulouse, France*. Morgan Kaufmann, 2002.

James Hendler and Deborah L. McGuinness. The DARPA Agent Markup Language. *IEEE Intelligent Systems*, 15(6):67–73, 2000.

Stijn Heymans and Dirk Vermeir. Integrating Ontology Languages and Answer Set Programming. In *Proceedings of the 14th International Workshop on Database and Expert Systems Applications (DEXA 2003), Prague, Czech Republic*, pages 584–588. IEEE Computer Society, 2003a.

Stijn Heymans and Dirk Vermeir. Integrating Semantic Web Reasoning and Answer Set Programming. In Marina De Vos and Alessandro Provetti, editors, *Proceedings on the Second International Workshop on Answer Set Programming, Advances in Theory and Implementation (ASP'03), Messina, Italy*, volume 78 of *CEUR Workshop Proceedings*, pages 194–208. CEUR-WS.org, 2003b.

Stijn Heymans, Davy Van Nieuwenborgh, and Dirk Vermeir. Nonmonotonic Ontological and Rule-Based Reasoning with Extended Conceptual Logic Programs. In Asunción Gómez-Pérez and Jérôme Euzenat, editors, *Proceedings of the Second European Semantic Web Conference (ESWC 2005), Heraklion, Crete, Greece*, volume 3532 of *Lecture Notes in Computer Science (LNCS)*, pages 392–407. Springer, 2005a.

Stijn Heymans, Davy Van Nieuwenborgh, and Dirk Vermeir. Preferential Reasoning on a Web of Trust. In Yolanda Gil, Enrico Motta, Richard V. Benjamins, and Mark Musen, editors, *Proceedings of the 4th International Semantic Web Conference (ISWC 2005), Galway, Ireland*, volume 3729 of *Lecture Notes in Computer Science (LNCS)*, pages 368–382. Springer, 2005b.

Ian Horrocks. DAML+OIL: A Reason-able Web Ontology Language. In Christian S. Jensen, Keith G. Jeffery, Jaroslav Pokorný, Simonas Saltenis, Elisa Bertino, Klemens Böhm, and Matthias Jarke, editors, *Proceedings of the 8th International Conference on Extending Database Technology, Advances in Database Technology (EDBT 2002), Prague, Czech Republic*, volume 2287 of *Lecture Notes in Computer Science (LNCS)*, pages 2–13. Springer, 2002a.

Ian Horrocks. DAML+OIL: A Description Logic for the Semantic Web. *IEEE Bulletin of the Technical Committee on Data Engineering*, 25(1):4–9, 2002b.

Ian Horrocks and Peter F. Patel-Schneider. Reducing OWL entailment to description logic satisfiability. In Dieter Fensel, Katia P. Sycara, and John Mylopoulos, editors, *Proceedings of the Second International Semantic Web Conference (ISWC 2003), Sanibel*

Island, FL, USA, volume 2870 of *Lecture Notes in Computer Science (LNCS)*, pages 17–29. Springer, 2003.

Ian Horrocks and Peter F. Patel-Schneider. A Proposal for an OWL Rules Language, 2004.

Ian Horrocks and Ulrike Sattler. Ontology Reasoning in the $\mathcal{SHOQ}(\mathbf{D})$ Description Logic. In Bernhard Nebel, editor, *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI 2001) Seattle, WA, USA*, pages 199–204. Morgan Kaufmann, 2001.

Ian Horrocks, Ulrike Sattler, and Stefan Tobies. Practical Reasoning for Expressive Description Logics. In Harald Ganzinger, David A. McAllester, and Andrei Voronkov, editors, *Proceedings of the 6th International Conference on Logic Programming and Automated Reasoning (LPAR'99), Tbilisi, Georgia*, volume 1705 of *Lecture Notes in Computer Science (LNCS)*, pages 161–180. Springer, 1999.

Ian Horrocks, Peter F. Patel-Schneider, and Frank van Harmelen. From $\mathcal{SHIQ}$ and RDF to OWL: The Making of a Web Ontology Language. *Journal of Web Semantics*, 1(1): 7–26, 2003.

Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, and Mike Dean. SWRL: A Semantic Web Rule Language Combining OWL and RuleML, May 2004. URL http://www.w3.org/Submission/SWRL. W3C Member Submission.

Ulrich Hufstadt, Boris Motik, and Ulrike Sattler. Reasoning for Description Logics around SHIQ in a Resolution Framework. Technical Report 3-8-04/04, Forschungszentrum Informatik (FZI), Karlsruhe, 76131 Karlsruhe, Germany, July 2004.

Ullrich Hustadt, Boris Motik, and Ulrike Sattler. Reducing SHIQ-Description Logic to Disjunctive Datalog Programs. In Didier Dubois, Christopher Welty, and Mary-Anne Williams, editors, *Proceedings of the 9th International Conference on Principles of Knowledge Representation and Reasoning (KR 2004), Whistler, British Columbia, Canada*, pages 152–162. AAAI Press, June 2004.

Neil Immerman. Languages that Capture Complexity Classes. *SIAM Journal on Computing*, 16(4):760–778, August 1987.

Tomi Janhunen, Ilkka Niemelä, Patrik Simons, and Jia-Huai You. Unfolding Partiality and Disjunctions in Stable Model Semantics. In Anthony G. Cohn, Fausto Giunchiglia, and Bart Selman, editors, *Proceedings of the 7th International Conference on Principles of Knowledge Representation and Reasoning (KR 2002), Breckenridge, CO, USA*, pages 411–419. Morgan Kaufmann, 2000.

Michael Kifer, Georg Lausen, and James Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. *Journal of the ACM*, 42(4):741–843, 1995.

Thomas Krennwallner. Integration of Conjunctive Queries to Description Logics into HEX-Programs. Master's thesis, Vienna University of Technology, Austria, 2007. To appear.

Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV System for Knowledge Representation and Reasoning. Technical report INFSYS RR-1843-02-14, Institut für Informationssysteme, Technische Universität Wien, A-1040 Vienna, Austria, October 2002.

Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV System for Knowledge Representation and Reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, 2006.

Alon Y. Levy and Marie-Christine Rousset. Combining Horn Rules and Description Logics in CARIN. *Artificial Intelligence*, 104(1-2):165–209, 1998.

Vladimir Lifschitz and Hudson Turner. Splitting a Logic Program. In *Proceedings of the 11th International Conference on Logic Programming (ICLP-94), Santa Margherita Ligure, Italy*, pages 23–38. MIT-Press, June 1994.

Vladimir Lifschitz and Thomas Y. C. Woo. Answer Sets in General Nonmonotonic Reasoning (Preliminary Report). In Bernhard Nebel, Charles Rich, and William R. Swartout, editors, *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR 1992), Cambridge, MA, USA*, pages 603–614. Morgan Kaufmann, 1992.

Vladimir Lifschitz, Lappoon R. Tang, and Hudson Turner. Nested Expressions in Logic Programs. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):369–389, 1999.

Thomas Lukasiewicz. Probabilistic Description Logic Programs. In *Proceedings of the 8th European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty (ECSQARU 2005), Barcelona, Spain*, volume 3571 of *Lecture Notes in Computer Science (LNCS)*, pages 737–749. Springer, 2005a. Extended version in *Int. J. Approx. Reasoning*, in press.

Thomas Lukasiewicz. Stratified Probabilistic Description Logic Programs. In Paulo Cesar G. da Costa, Kathryn B. Laskey, Kenneth J. Laskey, and Michael Pool, editors, *Proceedings of the ISWC 2005 Workshop on Uncertainty Reasoning for the Semantic Web, Galway, Ireland*, pages 87–97, 2005b.

Thomas Lukasiewicz. Fuzzy Description Logic Programs under the Answer Set Semantics for the Semantic Web. In *Proceedings of the Second International Conference on Rules and Rule Markup Languages for the Semantic Web (RuleML 2006), Athens, GA, USA*. IEEE Computer Society, 2006.

Witold Łukaszewicz. *Non-Monotonic Reasoning: Formalizations of Commonsense Reasoning*. Ellis Horwood, 1990.

Victor Marek, Ilkka Niemelä, and Miroslaw Truszczyński. Logic Programs With Monotone Cardinality Atoms. In Vladimir Lifschitz and Ilkka Niemelä, editors, *Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2004), Fort Lauderdale, Florida*, volume 2923 of *Lecture Notes in Computer Science (LNCS)*, pages 154–166. Springer, 2004.

Wolfgang May, Bertram Ludäscher, and Georg Lausen. Well-founded semantics for deductive object-oriented database languages. In François Bry, Raghu Ramakrishnan, and Kotagiri Ramamohanarao, editors, *Proceedings of the 5th International Conference on Deductive and Object-Oriented Databases (DOOD'97), Montreux, Switzerland*, volume 1341 of *Lecture Notes in Computer Science (LNCS)*, pages 320–336. Springer, 1997.

Deborah L. McGuinness and Frank van Harmelen. OWL Web Ontology Language Overview. W3C Recommendation, World Wide Web Consortium, February 2004. URL http://www.w3.org/TR/2004/REC-owl-features-20040210/.

Alistair Miles and Dan Brickley. SKOS Core Vocabulary Specification. W3C Working Draft, World Wide Web Consortium, November 2005. URL http://www.w3.org/TR/2005/WD-swbp-skos-core-spec-20051102/.

Boris Motik and Ulrike Sattler. A Comparison of Reasoning Techniques for Querying Large Description Logic ABoxes. In Miki Hermann and Andrei Voronkov, editors, *Proceedings of the 13th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR 2006)*, volume 4246 of *Lecture Notes in AI (LNAI)*. Springer, 2006.

Boris Motik, Ulrike Sattler, and Rudi Studer. Query Answering for OWL-DL with Rules. *Journal of Web Semantics*, 3(1):41–60, July 2005.

Ilkka Niemelä. Logic Programming with Stable Model Semantics as Constraint Programming Paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3–4):241–273, 1999.

Ilkka Niemelä and Patrik Simons. Smodels: An Implementation of the Stable Model and Well-Founded Semantics for Normal Logic Programs. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'97), Dagstuhl, Germany*, volume 1265 of *Lecture Notes in Computer Science (LNCS)*, pages 420–429. Springer, 1997.

Ilkka Niemelä, Patrik Simons, and Tommi Syrjänen. Smodels: A System for Answer Set Programming. In Chitta Baral and Miroslaw Truszczyński, editors, *Proceedings of the 8th International Workshop on Non-Monotonic Reasoning, Breckenridge, CO, USA*. `arXiv.org` e-Print archive, April 2000.

Jeff Z. Pan, Enrico Franconi, Sergio Tessaris, Giorgos Stamou, Vassilis Tzouvaras, Luciano Serafini, Ian Horrocks, and Birte Glimm. Specification of Coordination of Rule and Ontology Languages. Project Deliverable D2.5.1, The Knowledge Web Project, June 2004.

Christos M. Papadimitriou. *Computational Complexity*. Addison-Wesley, Reading, Massachusetts, 1994. ISBN 0-201-53082-1.

Bijan Parsia and Evren Sirin. Pellet: An OWL DL Reasoner. In *3rd International Semantic Web Conference (ISWC 2004) – Posters Track, Hiroshima, Japan*, 2004.

David Pearce, Hans Tompits, and Stefan Woltran. Encodings for Equilibrium Logic and Logic Programs with Nested Expressions. In Pavel Brazdil and Alípio Jorge, editors, *Progress in Artificial Intelligence, Knowledge Extraction, Multi-agent Systems, Logic Programming and Constraint Solving, Proceedings of the 10th Portuguese Conference on Artificial Intelligence (EPIA 2001), Porto, Portugal*, volume 2258 of *Lecture Notes in Computer Science (LNCS)*, pages 306–320. Springer, 2001.

Axel Polleres and Roman Schindlauer. SPAR^{2}QL: From SPARQL to Rules. In *5th International Semantic Web Conference (ISWC 2006) – Posters Track, Athens, GA, USA*, November 2006. URL http://www.polleres.net/publications/poll-schi-2006.pdf.

Axel Polleres, Cristina Feier, and Andreas Harth. Rules with contextually scoped negation. In York Sure and John Domingue, editors, *Proceedings of the 3rd European Semantic*

Web Conference (ESWC 2006), Budva, Montenegro, volume 4011 of *Lecture Notes in Computer Science (LNCS)*, pages 332–347. Springer, 2006.

David Poole. A Logical Framework for Default Reasoning. *Artificial Intelligence*, 36:27–47, 1988.

Alessandro Provetti, Elisa Bertino, and Franco Salvetti. Local Closed-World Assumptions for reasoning about Semantic Web data. In Francesco Buccafurri, editor, *Proceedings of the 2003 Joint Conference on Declarative Programming (APPIA-GULP-PRODE 2003), Reggio Calabria, Italy*, pages 314–323, 2003.

Eric Prud'hommeaux and Andy Seaborne. SPARQL query language for RDF. W3C Candidate Recommendation, World Wide Web Consortium, April 2006. URL http://www.w3.org/TR/2006/CR-rdf-sparql-query-20060406/.

Teodor C. Przymusinski. On the Declarative and Procedural Semantics of Stratified Deductive Databases. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann, Washington DC, 1988.

Teodor C. Przymusinski. Stable Semantics for Disjunctive Programs. *New Generation Computing*, 9(3/4):401–424, 1991.

Prasad Rao, Konstantinos F. Sagonas, Terrance Swift, David S. Warren, and Juliana Freire. XSB: A System for Efficiently Computing WFS. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'97), Dagstuhl, Germany*, volume 1265 of *Lecture Notes in Computer Science (LNCS)*, pages 430–440. Springer, 1997.

Raymond Reiter. On Closed World Data Bases. In Hervé Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 119–140. Plenum Press, New York, 1978.

Raymond Reiter. A Logic for Default Reasoning. *Artificial Intelligence*, 13:81–132, 1980.

Raymond Reiter. A Theory of Diagnosis From First Principles. *Artificial Intelligence*, 32: 57–95, 1987.

Riccardo Rosati. Towards Expressive KR Systems Integrating Datalog and Description Logics: Preliminary report. In Patrick Lambrix, Alexander Borgida, Maurizio Lenzerini, Ralf Möller, and Peter F. Patel-Schneider, editors, *Proceedings of the 1999 International Workshop on Description Logics (DL'99), Linköping, Sweden*, volume 22 of *CEUR Workshop Proceedings*, pages 160–164. CEUR-WS.org, 1999.

Riccardo Rosati. On the Decidability and Complexity of Integrating Ontologies and Rules. *Journal of Web Semantics*, 3(1):61–73, 2005.

Riccardo Rosati. $\mathcal{DL}$+*log*: Tight Integration of Description Logics and Disjunctive Datalog. In Patrick Doherty, John Mylopoulos, and Christopher A. Welty, editors, *Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning (KR 2006), Lake District, UK*, pages 68–78. AAAI Press, 2006a.

Riccardo Rosati. Integrating Ontologies and Rules: Semantic and Computational Issues. In Pedro Barahona, François Bry, Enrico Franconi, Nicola Henze, and Ulrike Sattler, editors, *Reasoning Web, Second International Summer School 2006, Lissabon, Portugal, September 4–8, 2006, Tutorial Lectures*, volume 4126 of *Lecture Notes in Computer Science (LNCS)*, pages 128–151. Springer, 2006b.

Kenneth A. Ross. Modular Stratification and Magic Sets for Datalog Programs with Negation. *Journal of the ACM*, 41(6):1216–1266, 1994.

Roman Schindlauer. Nonmonotonic Logic Programs for the Semantic Web. In Maurizio Gabbrielli and Gopal Gupta, editors, *Proceedings of the 21st International Conference on Logic Programming (ICLP 2005), Sitges, Spain*, number 3668 in Lecture Notes in Computer Science (LNCS), pages 446–447. Springer, September 2005.

Alan L. Selman. A Taxonomy on Complexity Classes of Functions. *Journal of Computer and System Sciences*, 48(2):357–381, 1994.

Jeremy G. Siek, Lee-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual.* Addison-Wesley, 2002.

Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and Implementing the Stable Model Semantics. *Artificial Intelligence*, 138(1–2):181–234, June 2002.

Michael Sintek and Stefan Decker. TRIPLE - A Query, Inference, and Transformation Language for the Semantic Web. In Ian Horrocks and James Hendler, editors, *Proceedings of the First International Semantic Web Conference (ISWC 2002), Sardinia, Italy*, volume 2342 of *Lecture Notes in Computer Science (LNCS)*, pages 364–378. Springer, 2002.

Terrance Swift. Deduction in Ontologies via ASP. In Vladimir Lifschitz and Ilkka Niemelä, editors, *Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2004), Fort Lauderdale, Florida*, volume 2923 of *Lecture Notes in Computer Science (LNCS)*, pages 275–288. Springer, 2004.

Stefan Tobies. *Complexity Results and Practical Algorithms for Logics in Knowledge Representation.* PhD thesis, RWTH Aachen, Germany, 2001.

Jeffrey D. Ullman. *Principles of Database and Knowledge Base Systems.* Computer Science Press, 1989.

Kristof van Belleghem, Mark Denecker, and Danny De Schreye. A Strong Correspondence between Description Logics and Open Logic Programming. In Lee Naish, editor, *Proceedings of the 14th International Conference on Logic Programming (ICLP 1997), Leuven, Belgium*, pages 346–360. MIT Press, 1997.

Allen van Gelder. Negation as Failure using Tight Derivations for General Logic Programs. In *Foundations of Deductive Databases and Logic Programming*, pages 149–176. Morgan Kaufmann, San Francisco, CA, USA, 1988. ISBN 0-934613-40-0.

Allen van Gelder, Kenneth A. Ross, and John S. Schlipf. The Well-Founded Semantics for General Logic Programs. *Journal of the ACM*, 38(3):620–650, 1991.

Moshe Y. Vardi. The Complexity of Relational Query Languages (Extended Abstract). In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing (STOC'82), San Francisco, CA, USA*, pages 137–146. ACM Press, 1982.

Kewen Wang, Grigoris Antoniou, Rodney W. Topor, and Abdul Sattar. Merging and Aligning Ontologies in dl-Programs. In Asaf Adi, Suzette Stoutenburg, and Said Tabet, editors, *Proceedings of the First International Conference on Rules and Rule Markup Languages for the Semantic Web (RuleML 2005), Galway, Ireland*, volume 3791 of *Lecture Notes in Computer Science (LNCS)*, pages 160–171. Springer, 2005.