FAKULTÄT FÜR !NFORMATIK

MASTER THESIS

Default Reasoning on Top of Ontologies with dl-Programs

carried out at the

Institute for Information Systems
Knowledge Based Systems Group

Vienna University of Technology

under the instruction of

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

and the accompanying care of

Dipl.-Ing. Thomas Krennwallner

by

Bsc. DAO Tran Minh

Tigergasse 23–27, A-1080 Wien

June 18, 2008

Default Reasoning on Top of Ontologies with dl-Programs

DAO Tran Minh

June 18, 2008

Abstract

We study the usefulness of dl-programs in implementing Reiter's default logic on top of a Description Logic knowledge base (DL-KB). To this end, we investigate transformations from default theories to description logic programs (dl-programs) based on different established algorithms for computing default theory extensions, namely select-defaults-and-check and select-justifications-and-check algorithm. In each transformation, additional constraints are exploited to prune the search space based on conclusion-conclusion or conclusion-justification relations. The implementation was deployed as a new component for the dl-plugin for dlvhex, and evaluated with various experimental test ontologies, which showed promising results.

Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgements

First of all, I would like to express my deepest gratitude to my supervisor, Prof. Thomas Eiter, for giving me a chance working on this interesting topic, and for his orientation and support since the beginning with numerous meetings every week. For me, this was the best experience so far working with and learn a lot from one of the top professors in Computational Logic.

I would like to give a big thank to Thomas Krennwallner. This work is based on his master thesis and I received a lot of help from him, from the first suggestion of using boost spirit for parsing the input and finally reading and providing comments for my thesis. Without his support, I would not be able to stay here writing this acknowledgment to finish my thesis.

I would like to thank all professors in Facudade de Ciêncisa e Technologia, Universidade Nova de Lisboa; Institut für Informationssysteme, Technischen Universität Wien for their excellence courses which gave me the love to Computational Logic. I have learnt from them not only basic and advanced topics in Computational Logic, but also how professors encourage and inspire students to do their best.

I express my gratefulness to the European Commission for granting me the Erasmus Mundus scholarship to study in Portugal and Austria. The last two years were a wonderful experience in my life having an opportunity to study in an international environment, meet top professors in Computational Logic, visit European countries and learn about the beauty of Europe and its culture.

I would like to thank my family and my fiancée for everything they have done for me, their care, encouragement, and love, especially in these two years when I was studying abroad. For me, family is the most important thing in my life and I am happy to always have you beside me.

1 Introduction

To convey the flavor of our work, we will start this section by discussing a motivating example. Assume that we have an ontology describing knowledge about students, concentrating more on graduate students and their work in which a *student* can be either an *undergraduate student* or a *graduate student*. Furthermore, a *graduate student* can be either a *master student* or a *PhD student*. A *student* takes some courses and can have a *scholarship*. A diagram representation of this example is given in Example 1.2, other representations in terms of a description logic knowledge base (DL-KB) and the Web Ontology Language OWL are given in Example 2.5 and 2.6, respectively.

We know that normally, a graduate student needs to work as an *assistant*, either *teaching assistant* or *research assistant* to earn for his/her studying, but for some graduate students who win a scholarship or fellowship, it is not needed. Given a graduate student in such a knowledge base (KB) without any further information, we would like to conclude that he/she is an assistant; and later if we know that he/she has a scholarship, an opposite conclusion would be our favorite. However, such kinds of normal reasoning is not possible in description logics (DLs) due to its monotonicity property (see Section 1.1 for a discussion on monotonicity and nonmonotonicity). If one would like to impose such kinds of reasoning, one possibility is to embed default logics into terminological knowledge representation formalisms. The first attempt in this field was presented in [Baader and Hollunder, 1993]; however, there has not been further development on this approach since then.

Recently, an interesting topic has emerged among the Logic Programming, Description Logics, and Semantic Web communities, i.e., the problem of integrating rules and ontologies. For example, in our example about *student*, one would add a constraint saying that *"A teaching assistant cannot take a course that he/she is an assistant of, because if it happens then he/she will grade his/her own exam, which is not fair."* This is in fact not a trivial problem. There have been many proposals for integration of rules and ontologies such as framework combining Horn rules and Description Logics in [Levy and Rousset, 1996], r-hybrid knowledge bases [Rosati, 2005a], hybrid MKNF KBs [Motik et al., 2006], and dl-programs [Eiter et al., 2007b] (see more in section 1.4). Some of these approaches have mentioned the abilities to enable nonmonotonic reasoning, but none has provided a concrete implementation for default reasoning over ontologies. Based on these results, in order for users to express a nonmonotonic reasoning application, they have to know about such formalizations thoroughly, how to write a program and what is the semantics of the programs, this may not be necessary if they are experts in different fields and just want to work with some simple nonmonotonic rules of the form:

"If A is true and B can be consistently assumed then conclude C."

In this thesis, we would like to investigate in-depth the problem of enabling one formalization of nonmonotonic reasoning, i.e., default reasoning, on top of ontologies based on an available mechanism that allows for integrating rules and ontologies, namely dl-programs. We do not stop at some theoretical results but go one step further by providing a front-end.

The main contributions of this thesis, briefly summarized, are as follows:

1. We present three transformations embedding default rules over dl-rules, which are based on two well-known algorithms for evaluating extensions of a default theory, namely *Select-defaults-and-check* and *Select-justifications-and-check* algorithm [Cholewinski and Truszczynski]. The first transformation was proposed in [Eiter et al., 2007b] and the other two are proposed in this work motivated by the first one. Furthermore, pruning rules are also investigated for optimization purposes.

2. We show the equivalence between default theories over DL-KBs and the transformed dl-programs. i.e., if there exists an extension in the original default theory, then there will be a corresponding answer set in the transformed dl-program, and vice versa.

3. We report the implementation of our front-end which allows users to easily specify their specific knowledge in terms of defaults in a simple syntax together with an ontology as an OWL file, and have the benefit from the system without worrying about what is a dl-program, how to write such a program in that formalization, and how the extensions are evaluated. The extensions are returned as answer sets of the transformed dl-programs consisting of auxiliary predicates whose names represent the intuition of the concept/role in relation to the extensions. The implementation is called the df-converter and was deployed as a new component of the dl-plugin, a plug-in for the HEX program solver, dlvhex [Schindlauer, 2006].

4. We report some experimental results in comparing the performance of different transformations with respect to evaluation time. The results show that the two new transformations are much faster compared to the first one. On the other experimenting dimension, the caching technique provided by the dl-plugin leads a significant performance. The details on the experiment results are presented in Section 5.3. Moreover, these results reveals many interesting facts from which we can exploit to improve our system.

To prepare the reader with the necessary background, the upcoming sections will give more hints on the underlying machinery used in this work.

1.1 Nonmonotonic Reasoning

In this section, we start with a famous slogan given by Benjamin Franklin in 1789 "Nothing is certain, but death an taxes." Bypassing the implication about taxes, we consider the aspect that most of the things in daily life are uncertain, and this is the way how humans usually reason. We use statements such as: "In my experience, it must be the case that ...," or "There is no good reason not to believe that ..." This kind of reasoning is known as commonsense reasoning.

The discussion on the need for the automation of common-sense reasoning by McCarthy in [McCarthy, 1959] then opened a new research direction of Nonmonotonic Reasoning (NMR) which approximately started in 1975/1976, and developed resplendently from the late 1970s to early 1990s. In this period, many important results were published. Among these publications, the first two rules of negation were closed world assumption (CWA) in [Reiter, 1978] and the **if-and-only-if** (**iff**) statements by Clark in [Clark, 1977]. And it is also important to mention circumscription [McCarthy, 1977], the truth maintenance systems [Doyle, 1979], default reasoning [Reiter, 1980], and the use of modal logic to handle nonmonotonicity [McDermott and Doyle, 1980]. For a historical view of NMR, see [Minker, August 1991].

In this thesis, we will concentrate on enabling default reasoning on top of ontologies. To bring the flavor of NMR to readers, the rest of this section will show the crucial point of nonmonotonicity versus monotonicity and give an example asserting that classical logic is not sufficiently powerful to adequately reason as humans do.

In classical logic, it is well known that if we have a theory T from which we can conclude a formula ϕ, then later if we extend the T to a theory T' by adding more facts, ϕ still holds in T'. This property is called **monotonicity**.

But life is not that simple. People in the past once believed that the earth was flat, and later this belief was defeated by many explores and developments in geography that the earth is spherical. Many other conclusions considered true in the past now are revised as not. And we trust that there are still current beliefs now which will be defeated in future.

Example 1.1. Take a formal, yet simple example to see how different classical logic and common-sense reasoning are. We know that:

- Birds usually fly.
- Penguins are birds, but they do not fly.

This knowledge can be represented in classical logic by the following rules:

$$\begin{array}{rcl} \mathit{flies}(X) & \leftarrow & \mathit{bird}(X) \land \neg \mathit{penguin}(X). \\ \mathit{bird}(X) & \leftarrow & \mathit{penguin}(X). \\ \neg \mathit{flies}(X) & \leftarrow & \mathit{penguin}(X). \end{array}$$

Now we know that *"Tweety is a bird"*, then add a fact $\mathit{bird}(\mathit{tweety})$ into T, it is easy to see that by classical logic, we can conclude neither $flies(tweety)$ nor $\neg flies(tweety)$ since there is no information regarding *"Tweety is a penguin or not"*.

But in daily life, humans can overcome this kind of obstacles by making assumptions about incomplete information. By statistical information and experience, we know that *"Birds usually fly,"* and when there is no information against making an assumption that *"Tweety can fly,"* we come up with the conclusion that *"Tweety flies."* Later, if we know that *"Tweety is a penguin,"* then the assumption is no longer acceptable and we conclude that *"Tweety does not fly"*. Classical logic also concludes the same fact in this case.

Obviously, classical logic does not work for commonsense reasoning and we need to come up with something different. It is now an appropriate moment to present Minsky's general definition of NMR:

Definition 1.1. [Minsky, 1974]
1. By *nonmonotonic reasoning* we understand the drawing of conclusions which may be invalidated in the light of new information.
2. A logical system is *nonmonotonic* iff its provability relation violates the property of monotonicity.

The principal difference between classical logic and nonmonotonic logics is that classical logic formalises *truth* and *valid conclusions*, while nonmonotonic logics formalise *rationality* and *plausible conclusions.*

Rationality is central for common-sense reasoning. The intuition behind rationality is that although conclusions can be invalidated by new information, *they are not chosen at random.* At least one rational justification is required to accept a conclusion in common-sense reasoning. Rationality is surprisingly difficult to formalize. The reason is that it cannot be measured by number as in mathematic where classical logic plays a major role; furthermore, rationality has the following properties [Lukaszewicz, 1990]:

- *agent-dependent*: different agents may have different opinions on what is rational in a certain situation
- *purpose-dependent*: the acceptance of a proposition as a rational conclusion depends on the purpose it is used for.

It is also not surprising that different approaches for capturing different aspects of rationality bring us different NMR formalisms listed at the beginning of this section.

Plausible conclusions, also known as beliefs, are the subjective nature of common-sense reasoning. The following definition of beliefs was given in [Perlis, 1986]:

Definition 1.2. A proposition A is a belief of an agent G, i.e., G considers A as a rational conclusion, if G is prepared to use A as if it were true.

The following example from [Winograd, 1980] elucidates this definition. Assume that I plan a trip by car. To begin with, I must decide where my car actually is. Given no better evidence, the following is possible: (A) *"The car is there, where I parked it last."* According to definition 1.2, (A) is considered as a belief if I act as if it were true. That is, if I ignore all circumstances in which (A) could be false, and I base my actions on the assumption that (A) is true, then I believe in (A), *even if I cannot be sure that (A) is actually true.*

On the other hand, even if I am almost certain that (A) holds, but at the same time I improve my chances by checking whether a bus will pass by, in case the car is missing, then (A) is *not regarded as a belief*, rather *a very likely contingency.*

Finally, we present here a definition of *Nonmonotonic Inference Rules*:

Definition 1.3. By a nonmonotonic inference pattern, or a nonmonotonic inference rule, we understand a rule of the following form:

Given A, in the absence of evidence B, infer C.

To sum up, NMR deals with reasoning under incomplete information, uncertainty to reflect common-sense reasoning as humans do.

Among different formalisms for nonmonotonic reasoning, *default logic* proposed in [Reiter, 1980] is one of the most famous approaches. Under default logic, Example 1.1 can be represented as *"Birds fly* ***by default,****"* therefore if we know that *"Tweety is a bird"* and nothing more, then by default, we conclude that *"Tweety flies"*. The exception when we know that *"Tweety is known as a penguin"* is treated as in Example 1.1.

In daily life, default logic is applied quite often. For example, by default, I know that the Complexity Analysis class this week is on Friday, as the other weeks; or by default, we can consider that the solution found so far is the best solution; or by default, a person has an appendix, etc. Embedding default reasoning into ontologies therefore is meaningful and interesting to explore. Hence default logic is one of the most important ingredients for this thesis and we have Section 2.7 to present it more formally.

1.2 Ontology

The word *"Ontology"* comes from Philosophy where it is concerned with the study of being or existence. Researchers in Computer Science, especially in Artificial Intelligence (AI) then borrowed this term for the purpose of supporting sharing and reusing of formally represented knowledge among AI systems. This approach was proposed in [Neches et al., 1991], in which authors claimed that the ontology layer plays an important role as a standard component of knowledge systems. The definition of ontology then was given in [Gruber, 1995] as an *"explicit specification of a conceptualization"*, i.e., *"the objects, concepts, and other entities that are presumed to exist in some area of interest and the relationships that hold among them"* [Genesereth and Nilsson, 1987].

Also in this article, Gruber emphasized what is so-called *ontological commitments*. A common ontology defines the vocabulary with which queries and assertions are exchanged among a community of agents. An agent is said to be committed to a common ontology if its observable actions are consistent with the definition in the ontology. This property guarantees consistency, but not completeness; i.e., each agent needs not share its own knowledge base nor answer all queries which can be formulated in the shared vocabulary. Furthermore, the internal knowledge of each agent needs not be represented by the terms specified in the ontology. For example, provider and customer must agree on provisions of a contract between them, but each can have different point of views about the contract on how it brings benefits to them.

Gruber proposed a preliminary set of design criteria for ontologies whose purpose is knowledge sharing and interoperation among programs based on a shared conceptualization, namely:

1. **Clarity**: an ontology should be objectively defined by means of formalism, most possibly logical axioms.

2. **Coherence**: the defining axioms should be logically consistent, and the ontology should agree with not only implications consistent with the definition but also the concepts which are defined informally.

3. **Extendability**: based on existing ontologies, one should be able to define new terms for his own specialization without requiring the revision of existing definitions. This criteria implies monotonicity of ontology languages.

4. **Minimal encoding bias**: conceptualization is an abstract notion, hence it should be specified at the knowledge/semantics level independently of a particular symbol-level encoding.

5. **Minimal ontological commitment**: to fulfill the knowledge sharing purpose, an ontology should minimize ontological commitment to give participating agents the freedom specialize and instantiate the ontology as needed.

To end this section, we will point out some ontology languages classified by their structures and give a simple ontology example.

- **Frame-based** ontology languages: F-Logic, OKBC, and KM

- **Description logic-based** ontology languages: KL-ONE, and OWL (Web Ontology Language)

- **First-order logic-based** ontology languages: CycL and KIF

Among those ontology languages, the Web Ontology Language OWL has been recommended by the World Wide Web Consortium (W3C) as a standard for providing a framework for asset management, enterprise integration and the sharing and reuse of data on the Web. More details on OWL will be given in Section 2.4.

Example 1.2. We present here an example ontology in a diagram. A representation of this ontology in OWL is provided in Section 2.4. Basically, this ontology describes knowledge about students, concentrates on graduate students and their work.

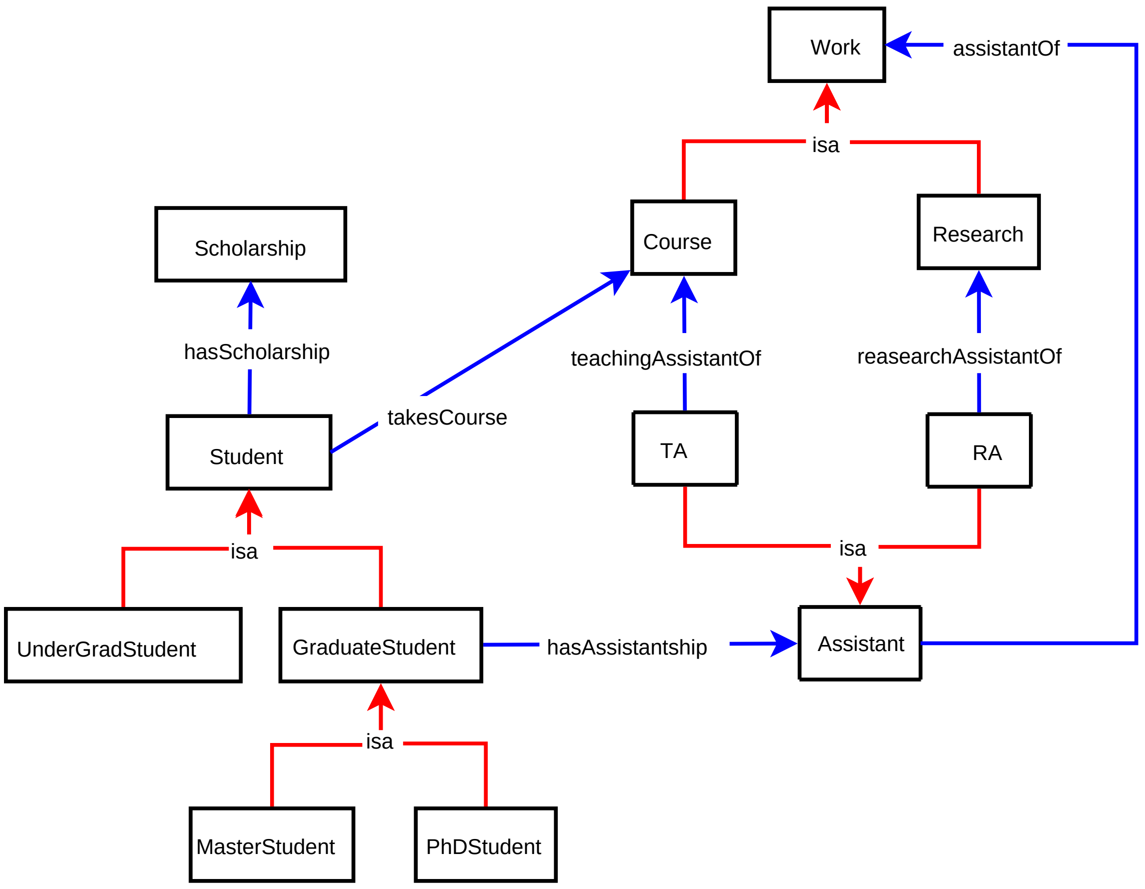


Figure 1.1: An example ontology

The concepts/classes are represented by boxes, the roles/relationships are represented by arrows. We select red arrows for the *is-a* relationship, which means that a class is a subclass of another one, while blue arrows are used for other relationships between classes. The intuitive meaning of this ontology is:

- A *Student* can be either *UnderGradStudent* or *GraduateStudent*;
- A *GraduateStudent* can be either *MasterStudent* or *PhDStudent*;
- A *Student* can have *Scholarship*;
- A *Student* takes *Course*(s);
- *Work* can be teaching a *Course* or doing a *Research*;
- An *Assistant* can be *TA* (TeachingAssistant) or *RA* (ResearchAssistant);

- An *Assistant* has to do *Work*;
- A *GraduateStudent* can have an assistantship; and
- A *TA* works on *Course* and an *RA* works on *Research*.

For this moment, we will keep the ontology simple. Later, we will pose some nonmonotonic rules on top of it and see how default logic can help to represent such spices.

1.3 Answer Set Programming

As an attempt to overcome limitations of traditional logic programming in which Prolog is a typical example, answer-set semantics, originally called stable model semantics, was proposed in [Gelfond and Lifschitz, 1988] and opened a new paradigm of Knowledge Representation and Reasoning (KRR). Based on this innovative idea, Answer Set Programming (ASP) has not only attracted researchers in AI but has also been applied successfully to many problems, such as plan generation and product configuration problems in AI, graph-theoretic problems arising in VLSI design and in historical linguistic. Its initial paper is among the top 5 AI source documents in terms of citeseer citations.

While having a similar syntax to Prolog, ASP differs from traditional logic programming in the following ways:

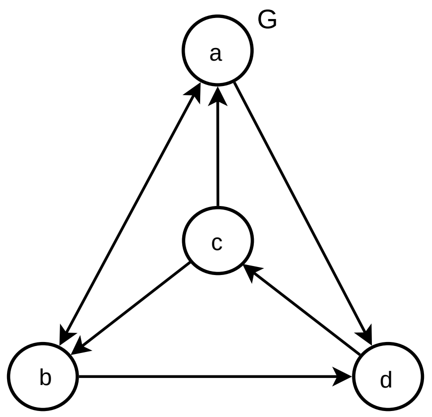
- Firstly, ASP is declarative, i.e., the order of rules in an Answer-Set program does not matter, while it does in Prolog.
- Secondly, ASP uses strong/classical and weak negation, also known as *Negation as Failure* (NAF), while there are many other proposals for the semantics of *"not"* in Logic Programming.
- Thirdly, in contrast to classical logic, ASP is non-monotonic (section 1.1).

ASP is very suitable for dealing with solving problems with uncertain, incomplete and inconsistent information.

The idea of evaluating an answer-set program P contains a guess-and-check strategy as follows: firstly, a set of atoms I will be guessed to be true. Then, the original program is reduced with respect to this guess, called P^I. The intuition of this reduction is that: if $a \in I$ and *"not a"* belongs to the body of a rules r, then r is no longer applicable, therefore r should be deleted; if $a \notin I$ then a is implicitly assigned the truth value $false$, hence *"not a"* has the value *true* and its appearance in the body of a rule r' can also be deleted. The reduct P^I then is a positive logic program and its least fixed point can be evaluated easily in polynomial time. If the least fixed point of P^I coincides with I, then I is an answer-set of P. Each answer-set of P is a solution for the encoded problem; if P has no answer-set, the corresponding problem has no solution.

An answer-set program usually has 3 parts: an encoding of the problem instance, a guessing/generating part which guesses all possible answers and a checking part which *"kills"* all wrong answers. To see how declarative ASP is and how this schema works, we will examine the Hamiltonian Cycle Problem (HAM), a well-known NP-complete problem [Garey and Johnson, 1979]: given a graph G, is there any cycle through G that visits each node exactly once?

Example 1.3. Figure 1.2 represents a graph G. It is easy to see that G^{HC} is one of the Hamiltonian cycles of G. The following answer-set program P is encoded in such a way that G has a Hamiltonian cycle iff P has an answer-set:



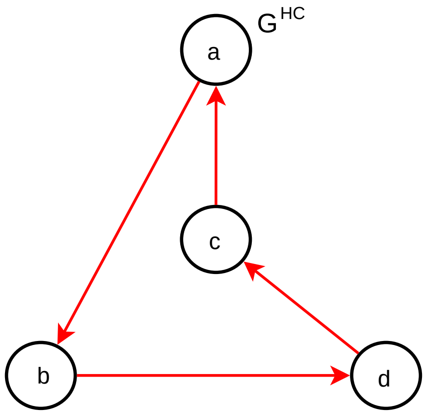


Figure 1.2: An example graph and one of its Hamiltonian cycles

Graph encoding:

(0) $arc(a,b).\ \ arc(a,d).\ \ arc(b,a).\ \ arc(b,d).\ \ arc(c,a).\ \ arc(c,b).\ \ arc(d,c).\ \ start(a).$

Guess solution:

$$\begin{array}{lrcl}
(1) & reached(X) & \leftarrow & in\_hm(\_,X). \\
(2) & in\_hm(X,Y) \vee out\_hm(X,Y) & \leftarrow & start(X), arc(X,Y). \\
(3) & in\_hm(X,Y) \vee out\_hm(X,Y) & \leftarrow & reached(X), arc(X,Y).
\end{array}$$

Check solution:

$$\begin{array}{lcl}
(4) & \leftarrow & in\_hm(X,Y), in\_hm(X,Y_1), Y \neq Y_1. \\
(5) & \leftarrow & in\_hm(X,Y), in\_hm(X_1,Y), X \neq X_1. \\
(6) & \leftarrow & arc(X,\_),\ \text{not}\ reached(X).
\end{array}$$

The intuition of this program is as follows: all facts in line (0) encode the graph G in terms of a binary predicate *arc* and specify the starting node by a fact $start(a)$. Since we are interested in cycles, the starting node is actually not important; any node can be chosen to play this role. Rules (2) and (3) guess whether an *arc* is in a Hamiltonian cycle from the starting node or a reached node. The guess is stored in two binary predicates, namely *in_hm* and *out_hm*. A node is reached if it is already guessed to be in the Hamiltonian cycle. From rules (4) to (6), we use rules without heads, also called *constraints*. A constraint has the power to *kills* all guesses making all atoms in its body become *true*. Therefore, the constraints (4), (5) eliminate all guesses such that there exists a node visited more than once; while constraint (6) rejects all guesses in which not all nodes are visited.

Note that this program not only solves the decision problem, whether G contains a Hamiltonian cycle or not, but also provides all Hamiltonian cycles, if any exists in its answer-sets. In fact, G has two Hamiltonian cycles corresponding to two answer-sets of P. We show here only the predicate *in_hm* which is directly related to the Hamiltonian cycles:

$$\{in\_hm(a,d), in\_hm(b,a), in\_hm(d,c), in\_hm(c,b), \ldots\}$$

$$\{in\_hm(a,b), in\_hm(b,d), in\_hm(d,c), in\_hm(c,a), \ldots\}$$

We have seen how powerful ASP is. In order to solve a problem, what we need is to specify the problem in terms of its semantics and translate it to a syntax supported by an ASP solver. Two famous ASP solvers are DLV[1] and Smodels. In this thesis, we use dlvhex,[2] a solver based on DLV; its dl-plugin gives us power to communicate with ontologies so that we can put some rules on top of ontologies. Next, we will see the necessity of the integration of rules and ontologies, the challenges of this work and what we expect to extend the available results to enable default reasoning on top of ontologies.

1.4 Integration of Rules and Ontologies

Section 1.2 provides an overview of ontology for the purpose of *supporting, sharing and reusing of formally represented knowledge among AI systems.* In the context of Semantic Web, the ontology layer has reached a certain level of maturity with W3C recommendations such as RDF and OWL. Recently, researchers have been interested in the integration of this layer and the Rule Layer for the following purposes:

- from the logic programming point of view, we can make use of ontologies to provide identified individuals, objects from different sources for sharing and reusing.

- from the ontology point of view, rule languages like logic programs are capable of overcoming obstacles in ontology formalisms based on Description Logics such as higher relational expressivity, polyadic predicates, integrity constraints and modeling exceptions (see [Krennwallner, 2007] for details on these motivations).

However, several approaches to this problem have not provided straightforward solutions due to many difficulties. Next, we will address issues arising and two strategies for integrating rules and ontologies.

1.4.1 Issues

CWA vs. OWA First-order logic and its fragment description logics adopt the *Open World Assumption* (OWA), which means *"if a statement cannot be inferred from what is expressed in a system, then it still cannot be inferred to be false."* The OWA applies when we represent knowledge as we discover it and the reality described by the system can never be known to have been fully described. It is plausible to consider the Semantic Web as such system and apply OWA to it.

On the other hand, the *Closed World Assumption* (CWA) [Reiter, 1978] presumes that *"what is not currently known to be true is assumed to be false."* This assumption is indeed reasonable in real life. Take a typical example, a direct flight database, which is assumed to be complete in the sense that if there is no direct flight explicitly recorded in the database, then such direct flight does not exist. In logic programming, CWA is closely related to *Negation as Failure* (NAF) which derives *not p* from the failure in deriving *p*.

To see the difference between OWA and CWA, consider the following concrete example:

Example 1.4.

$$\begin{array}{rcl} wine(X) & \leftarrow & whiteWine(X). \\ nonWhite(X) & \leftarrow & \text{not } whiteWine(X). \\ & & wine(myDrink). \end{array}$$

[1]`http://www.dlvsystem.com/`

[2]`http://www.kr.tuwien.ac.at/research/dlvhex/`

Under the non-availability of $whiteWine(myDrink)$, this program concludes $nonWhite(myDrink)$, whereas a similar representation under description logics would not justify the same conclusion:

Example 1.5.

$$\begin{array}{rcl} \mathit{WhiteWine} & \sqsubseteq & \mathit{Wine} \\ \neg\,\mathit{WhiteWine} & \sqsubseteq & \mathit{NonWhite} \\ \mathit{myDrink} & \in & \mathit{Wine} \end{array}$$

The reason for this behaviour is that, under OWA, there is not enough information to guarantee either $myDrink \in WhiteWine$ or $myDrink \in \neg WhiteWine$, hence no further conclusion can be made.

Although CWA is not adopted in the Semantic Web since ontologies are based on description logics, it is still needed in many applications, e.g., in information integration. Thus, integration of a nonmonotonic formalism and ontologies plays an important role for such a purpose.

Strong Negation vs. Classical Negation Sometimes, people equate strong negation with classical negation, but strong negation as used in ASP is in fact slightly different from each its classical counterpart. The following example demonstrates this divergence:

Example 1.6.

$$\begin{array}{lcl} \mathit{Wine}(X) \leftarrow \mathit{WhiteWine}(X). & \qquad & \mathit{WhiteWine} \sqsubseteq \mathit{Wine}. \\ -\,\mathit{Wine}(\mathit{myDrink}). & & \mathit{myDrink} \in \neg\,\mathit{Wine}. \end{array}$$

While in the description-logic knowledge base, we would conclude $\mathit{myDrink} \in \neg\,\mathit{WhiteWine}$, the fact $-\,\mathit{WhiteWine}(\mathit{myDrink})$ cannot be justified in the logic programming setting. However, adding a rule $\mathit{WhiteWine}(X) \vee -\,\mathit{WhiteWine}(X)$ in this particular example will help to derive this fact.

UNA vs. non-UNA ASP (and typically all logic-based programming languages) operates under the Unique Names Assumption (UNA), which basically says that the function relating constants on the language and objects on the domain of the interpretation is a bijection. This assumption is not valid in general on DL. The function that correlates constants and objects in the domain may not be injective nor surjective. As a simple example, the following answer-set program does not have any model:

Example 1.7.

$$\begin{array}{r} \leftarrow \mathit{friendOf}(\mathit{tweety}, X), \mathit{friendOf}(\mathit{tweety}, Y), X \neq Y. \\ \mathit{friendOf}(\mathit{tweety}, \mathit{joe}). \\ \mathit{friendOf}(\mathit{tweety}, \mathit{pluto}). \end{array}$$

while the corresponding DL representation,

$$\begin{array}{l} \mathit{tweety} \in \leq 1 \mathit{friendOf} \\ \mathit{friendOf}(\mathit{tweety}, \mathit{joe}) \\ \mathit{friendOf}(\mathit{tweety}, \mathit{pluto}) \end{array}$$

has a model in which $\mathit{joe} = \mathit{pluto}$. This difference in basic assumption, when not taken care of, will certainly pose problems in how we attach semantic to the integration of the two systems.

Decidability Lastly, we want to archive an integration that has the nice property of maintaining decidability while allowing for as much expressiveness as possible. Ideally, we would want a system that tries to be as expressive as possible under the constraint of some class of computational complexity. The problem arises in combining the two systems of DL and logic-based programming, when we consider the fact that these two systems tries to approach and tackle the problem of decidability from two different angles:

- Decidability in ASP is attained from the fact that it is based on function-free Horn Logic, where ground-entailment can be checked using model-checking in finite subsets of the Herbrand base of the program. In other words, decidability depends on the finite-ness of the domain. Even the (more simple) Prolog semantic (using SLDNF), when we consider functions in the language combined with full left-right recursion could results in non-decidability.
- On the other hand, DL semantic maintains decidability by limiting the constructs it offers to end up in a certain subset of first-order logic. The reasoning tasks of deciding class membership, subsumption, satisfiability etc. rest on the fact that there are only such a finite number of constructs allowed in the terminology.

Because of these different approaches, a naive attempt at combining both worlds could result in non-decidability, even on simple cases as in [Levy and Rousset, 1996], many detailed subtleties in the combination of the two system could lead to undecidability, if not limited accordingly.

1.4.2 Strategies for Integrating Rules and Ontologies

[Eiter et al., 2006] proposed two strategies for combining the two worlds of logic programming and classical logic, in particular description logics, namely:

- *tight semantic integration* (Figure 1.3)
- *strict semantic separation* (Figure 1.4)

We will now analyze the principles of these two strategies and review their related work.

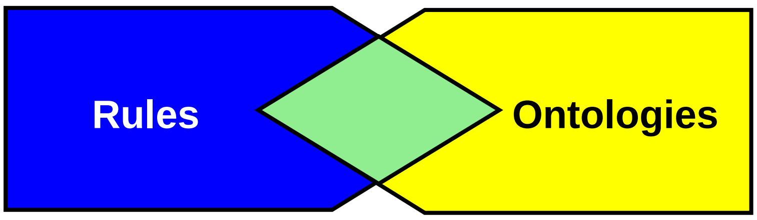


Figure 1.3: Tight semantic integration

Integration of rules and ontologies with tight semantic integration In this strategy, rules are introduced directly in the Ontology Layer, i.e., concepts' and roles' names can be used as predicate names in rules. Such an approach can easily lead to undecidability, for example CARIN [Levy and Rousset, 1996] and SWRL [Horrocks et al., 2004]. On the other hand, DLP proposed in [Grosof et al., 2003] which preserves decidability is very restricted in its syntax, hence limits the expressivity. SWRL and DLP can be seen as two opposite extremes at two bridgeheads, letting a big scale in between for other approaches to fit in, such as $\mathcal{DL}$-log [Rosati, 1999], DL-safe Rules [Motik et al., 2005], safe hybrid KBs [Rosati, 2005b], r-hybrid KBs [Rosati, 2005c], and the expressive $\mathcal{DL}$+log [Rosati, 2006a,b]. These approaches, in order to maintain decidability and extend expressivity, require a safety condition forcing variables in rules to occur in certain places.

Another approach is to reduce DL-KBs, in particular $\mathcal{SHIQ}$ knowledge bases, to equivalent disjunctive logic programs [Hufstadt et al., 2004]. Hence, reasoning in DL is carried out by standard reasoning algorithms of such programs. Since in the end one needs to work only with rules, it is easy to add rules to a translated DL-KB.

We also need to mention one more class of proposed formalisms which can be considered as unifying formalisms of logic programming and first-order theories, including Hybrid MKNF KB [Motik et al., 2006, Motik and Rosati, 2007], [de Bruijn et al., 2007a] which uses *Autoepistemic Logic*, and [de Bruijn et al., 2007b] uses *Quantified Equilibrium Logic*.

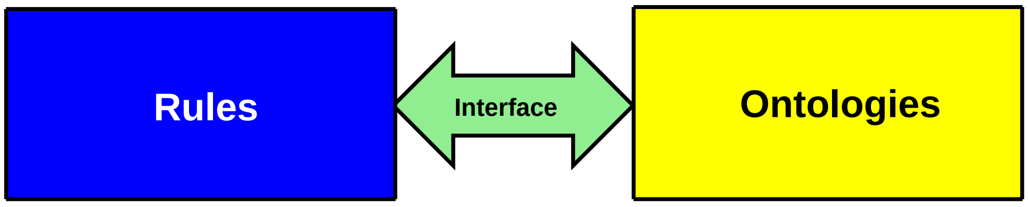


Figure 1.4: Strict semantic separation

Integration of rules and ontologies with strict semantic separation In this approach, rules (ASP) and ontologies (OWL/RDF) play in different fields. While ASP concentrates on reasoning jobs, OWL/RDF flavors aims at their purpose of description languages. The two components are not forced to any syntactic restrictions, as long as their own sides are decidable; and then communicate to each other via a *"safe interface."*

From the Rules Layer point of view, ontologies serve as an external source of information with an independent semantics which can be updated and/or queried via a special predicate. Such approaches are [Eiter et al., 2005, 2007b, Lukasiewicz, 2005, Eiter et al., 2008] and the TRIPLE rules engine [Sintek and Decker, 2002] which calls external description logic reasoners.

For excellent surveys, we refer the interested readers to [Antoniou et al., 2005] and [Pan et al., 2004].

This thesis is motivated by the results published in [Eiter et al., 2007b, 2008], namely dl-programs, a framework for integrating rules and ontologies. A dl-program consists of a normal logic program and a DL-KB. The main idea of dl-programs is the use a special atom called dl-atom which has the ability of updating and querying a DL-KB, therefore provides a safe and bidirectional data flow from rules to ontologies and vice versa. Based on answer-set semantics, dl-programs guarantee decidability as long as the DL-KB is decidable. But above all, the most interesting fact is that dl-programs allow a clean integration of rules and ontologies so that users who familiar with logic programming can easily adopt this mechanism without having to worry about what kind of description logics is used underneath. Moreover, switching between different description logics will not cause a big problem as long as the used description logic is decidable. Therefore, we are very motivated to use dl-programs as a means to enable default reasoning on top of ontologies. The rest of this thesis will present our results in the order specified in the following section.

1.5 Thesis Organization

This chapter has discussed the underlying machinery related to our work. The next chapters will deal with the following problems:

- Chapter 2 provides preliminaries for logic and answer-set programming, the family of description logics and its relationship to the OWL Web Ontology Language, the notions of dl-programs and cq-programs, and default logic.
- Chapter 3 discusses the main contribution of this thesis, including analyzing the transformation from default logic to dl-programs proposed in [Eiter et al., 2007b], proposing some small modifications for a clearer intuition, new transformations justified by proofs, and some pruning techniques with the hope of bringing better performances.
- Chapter 4 describes the implementation of the df-converter as an additional component in the available dl-plugin in the plug-in environment provided by dlvhex.
- Some classical examples of default logics are given in Chapter 5. Experimental result will be provided in order to compare performances of different transformations and test the efficiency of pruning rules.
- Finally, Chapter 6 concludes the thesis and discusses future work based on this result.

2 Preliminaries

This chapter provides a more technical view of the underlying machineries used in this thesis. Firstly, we introduce the basics and principles of Answer-Set Programming. Description logics then are described as a formalism for ontology languages in the Semantic Web context, in particular the Web Ontology Language OWL. Then we outline both syntax and semantics of dl- and cq-programs, which are used to represent the result of our transformations from default logics, which is finally considered.

2.1 Declarative Logic Programming

In computer science, programming languages can be categorized into two big programming concepts, namely *imperative programming* and *declarative programming.* On the one hand, an imperative program comprises of a sequence of commands for the computer to perform, hence focuses on *how* to solve a problem by an algorithm. Such imperative programming languages are Fotran, C, C++, Java,... On the other hand, a declarative program concentrates on representing *what* are the properties of the desired solution. Therefore, programmers using purely declarative programming usually do not know how the solver process their programs. Further classifications in declarative programming bring us functional programming, logic programming, and constraint programming with LISP, Haskell, and variants of Prolog as typical languages.

Compared to imperative programs, declarative programs are usually more concise and more powerful in terms of reasoning abilities. One has tried to solve the Hamiltonian Cycle problem in C++ or Java can easily verify that he/she must use a deep-first-search implemented in a recursive way, and it cannot be written in just 6 lines like our example in section 1.3.

Hereafter, we will focus on *Declarative Logic Programming.* A programmer using this paradigm needs to specify in his/her logic program the relationships in the domain of discourse obeying the syntax of a language, and then gets the output through the semantics of the program.

There are logic programming languages such as Prolog which are not purely declarative. Evaluating such a program depends on the order of rules in the program and the order of atoms in a rule, hence makes it not very comprehensible and not easy to be modified. However, *answer-set programming* [Gelfond and Lifschitz, 1988] is purely declarative and guarantees termination. We have introduced ASP in section 1.3, the next section will present ASP's syntax and semantics in a more technical way.

Name	restriction
definite Horn	$k = 1$, $n = m$
Horn	$k \leq 1$, $n = m$
normal	$k \leq 1$
definite	$k \geq 1$, $n = m$
positive	$n = m$
disjunctive	no restriction

Table 2.1: Program classes

2.2 Logic Programming under the Answer-Set Semantics

2.2.1 Syntax of Answer-Set Programs

Let $\mathcal{P}$, $\mathcal{C}$, $\mathcal{V}$ be disjoint sets of predicate, constant, and variable symbols from a first-order vocabulary Φ, respectively, where $\mathcal{V}$ is infinite and $\mathcal{P}$ and $\mathcal{C}$ are finite. Assume that elements from $\mathcal{C}$ and $\mathcal{P}$ are string constants that begin with a lowercase letter or double-quoted, and elements from $\mathcal{C}$ can also be integer numbers; elements from $\mathcal{V}$ begin with an uppercase letter. A *term* is either a constant or a variable. Given $p \in \mathcal{P}$, 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 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*. *not l* evaluates to *true* if l cannot be proved, i.e., either l is false or we do not know whether l is true or false.

A *rule* r is an expression of the form

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

where $a_1, \ldots, a_k, b_1, \ldots, b_n$ are classical literals. We say that $a_1, \ldots, a_k$ is the *head* of r while the conjunction $b_1, \ldots, b_m, \text{not } b_{m+1}, \ldots, \text{not } b_n$ is the *body* of r. We use $H(r)$ to denote r's head literals, and $B(r)$ to denote the set of all its body literals $B^+(r) \cup B^-(r)$, where $B^+(r) = \{b_1, \ldots, b_m\}$ and $B^-(r) = \{b_{m+1}, \ldots, b_n\}$. A rule r without head literals (i.e., $k = 0$) is an *integrity constraint.* A rule r with exactly one head literal (i.e., $k = 1$) is a *normal rule.* If the body of r is empty (i.e., $m = n = 0$), 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 NAF, i.e., for all $r \in P, B^-(r) = \emptyset$ is called a *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 *normal logic program* (NLP). The generalization of an NLP by allowing default negation in the heads of rules is called *generalized logic program* (GLP). Additional program classes of logic programming with the corresponding restrictions on the rules in a program a summarized in Table 2.1.

2.2.2 Semantics of Answer-Set Programs

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

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

The *Herbrand universe* of a program P, denoted HU_P, is the set of all constant symbols $C \subseteq \mathcal{C}$ appearing in P. If there is no such constant symbol, then $HU_P = \{c\}$, where c is an arbitrary constant symbol from Φ. 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 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 in HU_P. We use $\mathit{ground}(P)$ 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 $X \supseteq HB_P$ is *consistent* iff $\{p, \neg p\} \subsetneq X$ 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*) from a program P relative to an interpretation $I \subseteq HB_P$, denoted P^I, as the ground positive program obtained from $\mathit{ground}(P)$ 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.

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 dismissed from the set of answer sets.

Example 2.1. Consider the following program P:

$$
\begin{aligned}
p &\leftarrow \text{not } q.\\
q &\leftarrow \text{not } p.
\end{aligned}
$$

Let $I_1 = \{p\}$; then, $P^{I_1} = \{p \leftarrow\}$ with the unique model $\{p\}$, 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 $\{p, q\}$.

Example 2.2. Let P be the following program:

$$
\begin{aligned}
\mathit{flies}(X) &\leftarrow \mathit{bird}(X), \text{not } \neg\mathit{flies}(X).\\
\mathit{bird}(X) &\leftarrow \mathit{penguin}(X).\\
\neg\mathit{flies}(X) &\leftarrow \mathit{penguin}(X).\\
\mathit{penguin}(\mathit{tweety})&.\ \mathit{bird}(\mathit{joe}).
\end{aligned}
$$

This program shows how ASP is used to handle nonmonotonic reasoning.

It is easy to see that $I = \{\mathit{penguin}(\mathit{tweety}), \mathit{bird}(\mathit{tweety}), \mathit{bird}(\mathit{joe}), \mathit{flies}(\mathit{joe}), \neg\mathit{flies}(\mathit{tweety})\}$

is an answer set of P. The reduct P^I is:

$$\begin{aligned} \mathit{flies}(\mathit{joe}) &\leftarrow \mathit{bird}(\mathit{joe}). \\ \mathit{bird}(\mathit{joe}) &\leftarrow \mathit{penguin}(\mathit{joe}). \\ \neg\mathit{flies}(\mathit{joe}) &\leftarrow \mathit{penguin}(\mathit{joe}). \\ \mathit{bird}(\mathit{tweety}) &\leftarrow \mathit{penguin}(\mathit{tweety}). \\ \neg\mathit{flies}(\mathit{tweety}) &\leftarrow \mathit{penguin}(\mathit{tweety}). \\ \mathit{penguin}(\mathit{tweety}).\ \mathit{bird}(\mathit{joe}). \end{aligned}$$

Now, if `penguinjoe` is added to P, then I is not an answer set of P anymore. Instead, we have the following answer set:
$I = \{\mathit{penguin}(\mathit{tweety}), \mathit{bird}(\mathit{tweety}), \mathit{bird}(\mathit{joe}), \mathit{penguin}(\mathit{joe}), \neg\mathit{flies}(\mathit{joe}), \neg\mathit{flies}(\mathit{tweety})\}$

The main reasoning tasks 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.

2.3 Description Logics

In the early days of Artificial Intelligence, ontologies were represented resorting to non-logic-based formalisms. Among these approaches, *semantic networks* and *frames systems* were the two most famous ones. Semantics Networks developed in the 1970s uses network-shaped cognition to represent concepts and relationships between them. The same purposes were shared in frames systems which appeared in [Minsky, 1975]. These two formalisms use graphical representation which was argued to be easy to design, but then it became difficult to manage with complex pictures. Furthermore, the lack of formal semantics limits their reasoning abilities; reasoning was actually done by manipulating *ad hoc* data structures.

In attempts to enhance the reasoning capability for semantic networks and frames systems, first-order logic was adopted to describe the semantics of core features of these networks, making use of unary predicates for describing sets of individuals and binary predicates for relationships between individuals. Then an important feature was discovered: ordinary frames systems and semantic networks do not require all first-order logic, but could be regarded as fragments of it. Therefore, typical reasoning used in structure-based representations does not require the full power of first-order theorem provers. In fact, specialized reasoning techniques can be applied. Furthermore, reasoning in different fragments of first-order logic amount computational problems with different complexities.

From frames systems to description logics, the name changed over time, starting with *terminological systems* emphasizing the language is used to define a terminology. Then, *concept language* was used to underline the concept-forming constructs of the languages. And *"Description Logics"* is used today, moving attention to the properties, including decidability, complexity, and expressivity, of the languages.

Description Logics also have a wide range of applications, whose domains spread from software engineering, configuration, medicine to digital libraries and Web-based information systems, etc.; and application areas focus on natural language and database management. The most well-known application of Description Logics, of course, is their use for ontologies and Semantic Web research. For an excellent introduction to Description Logics, we refer readers to [Baader et al., 2003, Baader and Lutz, 2006, Baader et al., 2007].

In this section, we will review the two description logics $\mathcal{SHIF}(\mathbf{D})$ and $\mathcal{SHOIN}(\mathbf{D})$ which are the foundation for the Web Ontology Language OWL-Lite and OWL-DL, respectively, and provide basis for dl-programs, cq-programs, on which we exploit for our purpose of enabling default reasoning on top of ontologies. Syntax and semantics of description logics can be defined in different ways. For our purpose, we use the definition in [Eiter et al., 2007b]

2.3.1 Syntax of $\mathcal{SHIF}(\mathbf{D})$ and $\mathcal{SHOIN}(\mathbf{D})$

We now recall the syntax of $\mathcal{SHIF}(\mathbf{D})$ and $\mathcal{SHOIN}(\mathbf{D})$. We first describe the syntax of the latter, which has the following datatypes and elementary ingredients. We assume a set of *elementary datatypes* and a set of *data values*. A *datatype* is either an elementary datatype or a set of data values (called *datatype oneOf*). A *datatype theory* $\mathbf{D} = (\Delta^{\mathbf{D}}, \cdot^{\mathbf{D}})$ consists of a *datatype* (or *concrete*) *domain* $\Delta^{\mathbf{D}}$ and a mapping $\cdot^{\mathbf{D}}$ that assigns to every elementary datatype a subset of $\Delta^{\mathbf{D}}$ and to every data value an element of $\Delta^{\mathbf{D}}$. The mapping $\cdot^{\mathbf{D}}$ is extended to all datatypes by $\{v_1, \ldots\}^{\mathbf{D}} = \{v_1^{\mathbf{D}}, \ldots\}$. Let $\mathbf{A}$, $\mathbf{R}_A$, $\mathbf{R}_D$, and $\mathbf{I}$ be pairwise disjoint finite nonempty sets of *atomic concepts, abstract roles, datatype* (or *concrete*) *roles*, and *individuals*, respectively. We denote by $\mathbf{R}_A^-$ the set of inverses R^- of all $R \in \mathbf{R}_A$.

Roles and concepts are defined as follows. 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 $C \in \mathbf{A}$ 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 $\neg C$ are concepts (called *conjunction, disjunction,* and *negation*, respectively). If C is a concept, R is an abstract 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). We use $\top$ and $\bot$ to abbreviate the concepts $C \sqcup \neg C$ and $C \sqcap \neg C$, respectively, and we eliminate parentheses as usual.

Example 2.3. Assume that we have an atomic concept *Scholarship* and a role *hasScholarship.* Then $\leq 1 \mathit{hasScholarship}$ is a concept denoting all individuals that are related to another individual via the role *hasScholarship* at most once. $\forall \mathit{hasScholarship}.\mathit{Scholarship}$ is another concept that consists of all individuals whose all related individuals via the role *hasScholarship* belong to the concept *Scholarship.*

We next define axioms and knowledge bases as follows. An *axiom* is an expression of one of the following forms: (1) $C \sqsubseteq D$ (called *concept inclusion axiom*), where C and D are concepts; (2) $R \sqsubseteq S$ (called *role inclusion axiom*), where either $R, S \in \mathbf{R}_A$ or $R, S \in \mathbf{R}_D$; (3) $\mathit{Tran}(R)$ (called *transitivity axiom*), where $R \in \mathbf{R}_A$; (4) $C(a)$ (called *concept membership axiom*), where C is a concept and $a \in \mathbf{I}$; (5) $R(a,b)$ (resp., $U(a,v)$) (called *role membership axiom*), 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 is a data value); and (6) $a = b$ (resp., $a \neq b$) (called *equality* (resp., *inequality*) *axiom*), where $a, b \in \mathbf{I}$. A (*description logic*) *knowledge base* L is a finite set of axioms.

Example 2.4. Assume that we have atomic concepts *Student*, *UnderGradStudent*, and *GraduateStudent.* To say that all undergraduate students are students, all graduate students are students, the following concept inclusion axioms must be specified:

$$\begin{aligned} \mathit{UnderGradStudent} &\sqsubseteq \mathit{Student} \\ \mathit{GraduateStudent} &\sqsubseteq \mathit{Student} \end{aligned}$$

For a role inclusion axioms example, assume that we have the following roles: *assistantOf*, *teachingAssistantOf*, *researchAssistantOf*, the suitable role inclusion axioms are:

$$\begin{aligned} \mathit{teachingAssistantOf} &\sqsubseteq \mathit{assistantOf} \\ \mathit{researchAssistantOf} &\sqsubseteq \mathit{assistantOf} \end{aligned}$$

For an abstract role $R \in \mathbf{R}_A$, we define $\mathit{Inv}(R) = R^-$ and $\mathit{Inv}(R^-) = R$. Let the transitive and reflexive closure of $\sqsubseteq$ on abstract role relative to L, denote $\sqsubseteq^*$, be defined as follows. For two abstract roles R and S in L, let $S \sqsubseteq^* R$ relative to L iff either (a) $S = R$, (b) $S \sqsubseteq R \in L$, (c) $\mathit{Inv}(S) \sqsubseteq \mathit{Inv}(R) \in L$, or (d) some abstract role Q exists such that $S \sqsubseteq^* Q$ and $Q \sqsubseteq^* R$ relative to L. An abstract role R is *simple* relative to L iff for each abstract role S such that $S \sqsubseteq^* R$ relative to L, it holds that (i) $\mathit{Trans}(S) \notin L$ and (ii) $\mathit{Trans}(\mathit{Inv}(S)) \notin L$. For decidability, number restrictions in L are restricted to simple abstract roles [Horrocks et al., 1999].

Observe that in $\mathcal{SHOIN}(\mathbf{D})$, concept and role membership axioms can equally be expressed through concept inclusion axioms. The knowledge that the individual a is an instance of concept C can be expressed by the concept inclusion axiom $\{a\} \sqsubseteq C$, while the knowledge that the pair (a, b) (resp., (a, v)) is an instance of role R (resp., U) can be expressed by $\{a\} \sqsubseteq \exists R.\{b\}$ (resp., $\{a\} \sqsubseteq \exists U.\{v\}$).

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.

Note that other definitions of description logic knowledge bases exist. A widely used definition is the notion of DL-KB $\mathcal{K} = \langle \mathcal{T}, \mathcal{R}, \mathcal{A} \rangle$, where $\mathcal{T}$ is called *TBox* and consists of a set of concept inclusion axioms (the terminological knowledge), $\mathcal{R}$ is called *RBox* and consists of a set of axiom of form (2), or (3) (the role hierarchy) and $\mathcal{A}$ is called *ABox* and consists of a set of concept or role membership axioms (the assertional knowledge, or extensional part). $\mathcal{T}$ and $\mathcal{R}$ builds the intentional part of a DL-KB. We do not use this clear separation in our framework, but sometimes refer to the extensional part of L as ABox and the intentional part of L as TBox.

2.3.2 Semantics of $\mathcal{SHIF}(\mathbf{D})$ and $\mathcal{SHOIN}(\mathbf{D})$

We now define the semantics of $\mathcal{SHIF}(\mathbf{D})$ and $\mathcal{SHOIN}(\mathbf{D})$ in terms of general first-order interpretations, as usual, and we recall some important reasoning problems in description logics.

An *interpretation* $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ with respect to a datatype theory $\mathbf{D} = (\Delta^{\mathbf{D}}, \cdot^{\mathbf{D}})$ consists of a nonempty (*abstract*) *domain* $\Delta^{\mathcal{I}}$ disjoint from $\Delta^{\mathbf{D}}$, and a mapping $\cdot^{\mathcal{I}}$ that assigns to each atomic concept $C \in \mathbf{A}$ a subset of $\Delta^{\mathcal{I}}$, to each individual $o \in \mathbf{I}$ an element of $\Delta^{\mathcal{I}}$, to each abstract role $R \in \mathbf{R}_A$ a subset of $\Delta^{\mathcal{I}} x \Delta^{\mathcal{I}}$, and to each datatype role $U \in \mathbf{R}_D$ a subset of $\Delta^{\mathcal{I}} x \Delta^{\mathcal{D}}$. The mapping $\cdot^{\mathcal{I}}$ is extended to all concepts and roles as usual (where $\#S$ denotes the cardinality of a set S):

- $(R^-)^{\mathcal{I}} = \{(a, b) \mid (b, a) \in R^{\mathcal{I}}\}$;
- $\{o_1, \ldots, o_n\}^{\mathcal{I}} = \{o_1^{\mathcal{I}}, \ldots, o_n^{\mathcal{I}}\}$;
- $(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}$, $(C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}}$, and $(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$;

- $(\exists R.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \exists y\colon (x,y) \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$;
- $(\forall R.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \forall y\colon (x,y) \in R^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}}\}$;
- $({\geq}nR)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \#(\{y \mid (x,y) \in R^{\mathcal{I}}\}) \geq n\}$;
- $({\leq}nR)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \#(\{y \mid (x,y) \in R^{\mathcal{I}}\}) \leq n\}$;
- $(\exists U.D)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \exists y\colon (x,y) \in U^{\mathcal{I}} \wedge y \in D^{\mathbf{D}}\}$;
- $(\forall U.D)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \forall y\colon (x,y) \in U^{\mathcal{I}} \rightarrow y \in D^{\mathbf{D}}\}$;
- $({\geq}nU)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \#(\{y \mid (x,y) \in U^{\mathcal{I}}\}) \geq n\}$;
- $({\leq}nU)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \#(\{y \mid (x,y) \in U^{\mathcal{I}}\}) \leq n\}$.

The *satisfaction* of a description logic axiom F in the interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ with respect to $\mathbf{D} = (\Delta^{\mathbf{D}}, \cdot^{\mathbf{D}})$, denotes $\mathcal{I} \models F$, is defined as follows:

- $\mathcal{I} \models C \sqsubseteq D$ iff $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$;
- $\mathcal{I} \models R \sqsubseteq S$ iff $R^{\mathcal{I}} \subseteq S^{\mathcal{I}}$;
- $\mathcal{I} \models \mathrm{Trans}(R)$ iff $R^{\mathcal{I}}$ is transitive;
- $\mathcal{I} \models C(a)$ iff $a^{\mathcal{I}} \in C^{\mathcal{I}}$;
- $\mathcal{I} \models R(a,b)$ iff $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}$ (resp., $\mathcal{I} \models U(a,v)$ iff $(a^{\mathcal{I}}, v^{\mathbf{D}}) \in U^{\mathcal{I}}$); and
- $\mathcal{I} \models a = b$ iff $a^{\mathcal{I}} = b^{\mathcal{I}}$ (resp., $\mathcal{I} \models a \neq b$ 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 $\mathcal{I} \models F$. The interpretation $\mathcal{I}$ *satisfies* a knowledge base L, or $\mathcal{I}$ is a *model* of L, denotes $\mathcal{I} \models L$, iff $\mathcal{I} \models F$ for all $F \in L$. We say that L is *satisfiable* (resp., *unsatisfiable*) iff L has a (resp., no) model. An axiom F is a *logical consequence* of L, denoted $L \models F$, iff every model of L satisfies F. A negated axiom $\neg F$ is a *logical consequence* of L, denoted $L \models \neg F$, iff every model of L does not satisfy F.

Some important reasoning problem related to description logic knowledge bases L are the following:

(1) decide whether a given L is satisfiable;

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

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

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

(5) given L, two individuals $a, b \in \mathbf{I}$ (resp., an individual $a \in \mathbf{I}$ and a value v), and an abstract role $R \in \mathbf{R}_A$ (resp., a datatype role $U \in \mathbf{R}_D$), decide whether $L \models R(a,b)$ (resp., $L \models U(a,v)$).

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

Example 2.5. Consider the Student ontology in example 1.2, this diagram can be represented as follows in a description logic knowledge base L:

$$\begin{aligned}
UnderGradStudent &\sqsubseteq Student && (1)\\
GraduateStudent &\sqsubseteq Student && (2)\\
UnderGradStudent &\sqsubseteq \neg GraduateStudent && (3)\\
MasterStudent &\sqsubseteq GraduateStudent && (4)\\
PhDStudent &\sqsubseteq GraduateStudent && (5)\\
MasterStudent &\sqsubseteq \neg PhDStudent && (6)\\
Course &\sqsubseteq Work && (7)\\
Research &\sqsubseteq Work && (8)\\
Course &\sqsubseteq \neg Research && (9)\\
TA &\sqsubseteq Assistant && (10)\\
RA &\sqsubseteq Assistant && (11)\\
TA &\sqsubseteq \exists teachingAssistantOf.Course && (12)\\
RA &\sqsubseteq \exists researchAssistantOf.Research && (13)\\
teachingAssistantOf &\sqsubseteq assistantOf && (14)\\
researchAssistantOf &\sqsubseteq assistantOf && (15)\\
\leq 1 hasScholarship &\sqsubseteq Student && (16)\\
\top &\sqsubseteq \forall hasScholarship.Scholarship && (17)\\
\exists assistantOf &\sqsubseteq Assistant && (18)\\
\top &\sqsubseteq \forall assistantOf.Work && (19)\\
\exists teachingAssistantOf &\sqsubseteq TA && (20)\\
\top &\sqsubseteq \forall teachingAssistantOf.Course && (21)\\
\exists researchAssistantOf &\sqsubseteq RA && (22)\\
\top &\sqsubseteq \forall reaseachAssistantOf.Research && (23)\\
\exists takesCourse &\sqsubseteq Student && (24)\\
\top &\sqsubseteq \forall takesCourse.Course && (25)\\
MasterStudent&(min) && (26)\\
PhDStudent&(tkren) && (27)\\
hasScholarship&(min, erasmus\_mundus) && (28)
\end{aligned}$$

All the concept inclusions from (1)–(25) form the TBox of L, while the assertions from (24)–(26) are the ABox of L. Basically, (1)–(11) represent the hierarchy between groups of atomic concepts as in Figure 1.1. (12) and (13) say that each *TA* must be responsible for some courses and each *RA* must do some research; (14) and (15) affirm that *teachingAssistantOf* and *researchAssistantOf* are subproperties of *assistantOf.* Axioms (16)–(25) set the domain and range for properties in the ontology. And finally, assertions (26), (27) declare two individuals, *min* as a *MasterStudent* and *tkren* as a *PhDStudent.* Assertion (28) states that *min* holds an *erasmus_mundus* scholarship.

Some simple inferences can be made from this knowledge base, for example:

- $L \models Student(min)$,
- $L \models Scholarship(erasmus\_mundus)$,

- $L \models \mathit{PhDStudent} \sqsubseteq \mathit{Student}, etc.$

2.4 Web Ontology Language

Section 1.2 has mentioned the idea behind ontologies, and pointed out that OWL was recommended by W3C to be a standard language for specifying ontologies on the Semantic Web. OWL consists of three sublanguages with increasing expressivity: *OWL Lite*, *OWL DL*, and *OWL Full*. The semantics of OWL Lite and OWL DL is based on the description logics $\mathcal{SHIF}(\mathbf{D})$ and $\mathcal{SHOIN}(\mathbf{D})$, respectively. OWL Full, on the other hand, loosens specific syntactic restrictions of OWL Lite and OWL DL, thus reasoning in OWL Full is undecidable. In this section, we provide a summary of OWL Lite and OWL DL's syntax and semantics in the tables belows, which are from [Horrocks et al., 2003].

Syntax and semantics of OWL

The abstract syntax for class descriptions and axioms in OWL DL ontologies is given in the first column of Table 2.2 and Table 2.3, respectively. The syntax for OWL Lite is basically the same, with the following restrictions:

- `oneOf`, `unionOf`, and `complementOf` are prohibited;
- `maxCardinality`, `minCardinality`, and `cardinality` restrictions may only have 0 and 1 as parameter n;
- `hasValue` restrictions are prohibited; and
- `EnumeratedClass` and `DisjointClass` axioms are not allowed.

The concrete constraints on the syntax of OWL Lite ontologies are summarized in `http://www.w3.org/TR/owl-ref/#OWLLite`.

Note that we omitted `AnnotationProperty` and `OntologyProperty` axioms, since they do not have an immediate DL equivalent expression. In fact, those property axioms are merely used for annotating extralogical information in the ontology like authorship information and textual descriptions for class and property axioms. As an aside, the `owl:imports` built-in ontology property can be used to include other ontologies (see also the discussion in [Horrocks et al., 2003]

The second column of Table 2.2 and Table 2.3 maps OWL abstract syntax to the corresponding DL syntax, hence OWL ontologies can be imagined as syntactic variants of description logics. Moreover, OWL ontologies are in general just RDF graphs, consequently they may come in form of RDF/XML[2], the usual way to denote OWL ontologies. The next example will show that this particular manner for denoting OWL ontologies is not designed for human-readability; instead, RDF/XML is built for an easy machine-to-machine communication.

Example 2.6. We again describe the Student ontology in Example 1.2, but this time in terms of OWL abstract syntax. The OWL language constructors in use fit nicely into the OWL DL language.

[2]`http://www.w3.org/TR/rdf-syntax-grammar/`

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`	$\bot$	$\texttt{owl : Nothing}^{\mathcal{I}} = \emptyset$
`intersectionOf(`$C_1\ C_2\ \ldots$`)`	$C_1 \sqcap C_2$	$(C_1 \sqcap C_2)^{\mathcal{I}} = C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}}$
`unionOf(`$C_1\ C_2\ \ldots$`)`	$C_1 \sqcup C_2$	$(C_1 \sqcup C_2)^{\mathcal{I}} = C_1^{\mathcal{I}} \cup C_2^{\mathcal{I}}$
`complementOf(`C`)`	$\neg C$	$(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
`oneOf(`$o_1\ \ldots$`)`	$\{o_1, \ldots\}$	$\{o_1, \ldots\}^{\mathcal{I}} = \{o_1^{\mathcal{I}}, \ldots\}$
`restriction(`R `someValuesFrom(`C`))`	$\exists R.C$	$(\exists R.C)^{\mathcal{I}} = \{x \mid \exists y.\langle x, y\rangle \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$
`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`))`	$\geq nR$	$(\geq nR)^{\mathcal{I}} = \{x \mid \sharp(\{y \mid \langle x, y\rangle \in R^{\mathcal{I}}\}) \geq n\}$
`restriction(`R `maxCardinality(`n`))`	$\leq nR$	$(\leq nR)^{\mathcal{I}} = \{x \mid \sharp(\{y \mid \langle x, y\rangle \in R^{\mathcal{I}}\}) \leq n\}$
`restriction(`R `cardinality(`n`))`	$= nR$	$(= nR)^{\mathcal{I}} = \{x \mid \sharp(\{y \mid \langle x, y\rangle \in R^{\mathcal{I}}\}) = n\}$
`restriction(`U `someValuesFrom(`D`))`	$\exists U.D$	$(\exists U.D)^{\mathcal{I}} = \{x \mid \exists y.\langle x, y\rangle \in U^{\mathcal{I}} \wedge y \in D^{\mathbf{D}}\}$
`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`))`	$\geq nU$	$(\geq nU)^{\mathcal{I}} = \{x \mid \sharp(\{y \mid \langle x, y\rangle \in U^{\mathcal{I}}\}) \geq n\}$
`restriction(`U `maxCardinality(`n`))`	$\leq nU$	$(\leq nU)^{\mathcal{I}} = \{x \mid \sharp(\{y \mid \langle x, y\rangle \in U^{\mathcal{I}}\}) \leq n\}$
`restriction(`U `cardinality(`n`))`	$= nU$	$(= nU)^{\mathcal{I}} = \{x \mid \sharp(\{y \mid \langle x, y\rangle \in U^{\mathcal{I}}\}) = n\}$
Data Ranges (D)		
D (URI reference)	D	$D^{\mathbf{D}} \subseteq \Delta^{\mathbf{D}}$
`oneOf(`$v_1 \ldots$`)`	$\{v_1, \ldots\}$	$\{v_1, \ldots\}^{\mathcal{I}} = \{v_1^{\mathcal{I}}, \ldots\}$
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	$U^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathbf{D}}$
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 2.2: OWL DL Syntax vs. DL Syntax and Semantics

Abstract Syntax	DL Syntax	Semantics
`Class(`A `partial` $C_1 \ldots C_n$`)`	$A \sqsubseteq C_1 \sqcap \cdots \sqcap C_n$	$A^{\mathcal{I}} \subseteq C_1^{\mathcal{I}} \cap \cdots \cap C_n^{\mathcal{I}}$
`Class(`A `complete` $C_1 \ldots C_n$`)`	$A = C_1 \sqcap \cdots \sqcap C_n$	$A^{\mathcal{I}} = C_1^{\mathcal{I}} \cap \cdots \cap C_n^{\mathcal{I}}$
`EnumeratedClass(`$A\ o_1 \ldots o_n$`)`	$A = \{o_1, \ldots, o_n\}$	$A^{\mathcal{I}} = \{o_1^{\mathcal{I}}, \ldots, o_n^{\mathcal{I}}\}$
`SubClassOf(`$C_1\ C_2$`)`	$C_1 \sqsubseteq C_2$	$C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$
`EquivalentClasses(`$C_1 \ldots C_n$`)`	$C_1 = \cdots = C_n$	$C_1^{\mathcal{I}} = \cdots = C_n^{\mathcal{I}}$
`DisjointClasses(`$C_1 \ldots C_n$`)`	$C_i \sqcap C_j = \bot, i \neq j$	$C_i^{\mathcal{I}} \cap C_j^{\mathcal{I}} = \emptyset, i \neq j$
`Datatype(`D`)`		$D^{\mathcal{I}} \subseteq \Delta^{\mathbf{D}}$
`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`)`	$\geq 1\, U \sqsubseteq C_i$	$U^{\mathcal{I}} \subseteq C_i^{\mathcal{I}} \times \Delta^{\mathbf{D}}$
`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(`$U_1 \ldots U_n$`)`	$U_1 = \cdots = U_n$	$U_1^{\mathcal{I}} = \cdots = U_n^{\mathcal{I}}$
`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(`$R_1 \ldots R_n$`)`	$R_1 = \ldots = R_n$	$R_1^{\mathcal{I}} = \ldots = R_n^{\mathcal{I}}$
`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(`$o_1 \ldots o_n$`)`	$o_1 = \cdots = o_n$	$o_1^{\mathcal{I}} = \cdots = o_n^{\mathcal{I}}$
`DifferentIndividuals(`$o_1 \ldots o_n$`)`	$o_i \neq o_j, i \neq j$	$o_i^{\mathcal{I}} \neq o_j^{\mathcal{I}}, i \neq j$

Table 2.3: OWL DL Axioms and Facts

```
SubClassOf(UnderGradStudent Student)
SubClassOf(GraduateStudent Student)
SubClassOf(UnderGradeStudent complementOf(GraduateStudent))
SubClassOf(MasterStudent GraduateStudent)
SubClassOf(PhDStudent GraduateStudent)
SubClassOf(MasterStudent complementOf(PhDStudent))
SubClassOf(Course Work)
SubClassOf(Research Work)
SubClassOf(Course complementOf(Research))
SubClassOf(TA Assistant)
SubClassOf(RA Assistant)
SubClassOf(TA restriction(teachingAssistantOf someValuesFrom(Course)))
SubClassOf(RA restriction(researchAssistantOf someValuesFrom(Research)))
SubPropertyOf(teachingAssistantOf AssistantOf)
SubPropertyOf(researchAssistantOf AssistantOf)
SubClassOf(restriction(hasScholarship maxCardinality(1)) Student)
ObjectProperty(hasScholarship range(Scholarship))
ObjectProperty(assistantOf domain(Assistant) range(Work))
ObjectProperty(teachingAssistantOf domain(TA) range(Course))
ObjectProperty(researchAssistantOf domain(RA) range(Research))
ObjectProperty(researchAssistantOf domain(Student) range(Course))
Individual(min type(MasterStudent))
Individual(tkren type(PhDStudent))
Individual(min value(hasScholarship erasmus_mundus))
```

The concrete syntax of OWL is much more verbose. For example, the following RDF/XML serialization represent the definition of the concept *GraduateStudent*:

```
<owl:Class rdf:ID="GraduateStudent">
   <rdfs:subClassOf>
      <owl:Class rdf:ID="Student"/>
   </rdfs:subClassOf>
    <owl:equivalentClass>
       <owl:Class>
         <owl:unionOf rdf:parseType="Collection">
          <owl:Class rdf:ID="PhDStudent"/>
          <owl:Class rdf:ID="MasterStudent"/>
        </owl:unionOf>
      </owl:Class>
    </owl:equivalentClass>
    <owl:disjointWith>
      <owl:Class rdf:ID="UndergraduateStudent"/>
    </owl:disjointWith>
  </owl:Class>
```

The semantics for OWL DL and OWL Lite is presented in the third column of Table 2.2 and Table 2.3. As remarked in the beginning of this section, OWL Lite and OWL DL ontologies correspond to $\mathcal{SHIF}(\mathbf{D})$ and $\mathcal{SHOIN}(\mathbf{D})$ DL-KBs, respectively. OWL uses the RDF datatyping scheme to refer to datatypes.[3] Hence, OWL uses XML Schema datatypes

[3] http://www.w3.org/TR/rdf-concepts/

like `xsd:string` or `xsd:float`. In a model $\mathcal{I}$, $\Delta^{\mathcal{I}}$ is the domain of individuals and $\Delta^{\mathbf{D}}$ is the domain of data values (cf. Section 2.3).

2.5 dl-Programs

First introduced in [Eiter et al., 2004a], dl-programs opened a new framework for integrating rules and ontologies. In this thesis, dl-programs and cq-programs play an important role as the target formalization of our transformation. Details about cq-programs will be described in Section 2.6. This section deals with syntax and semantics of dl-programs.

Description Logic programs (dl-programs) consist of a normal logic program P and a DL-KB L. The logic program P might contain special devices called dl-atoms which may occur in the body of a rule and involve queries to a DL-KB. Moreover, dl-atoms can specify an input to L before querying the external DL-KB, thus dl-programs allow for a bidirectional data flow between the description logic component and the logic program.

The way dl-programs interface DK-KBs allow them to act as a loosely coupled formalism. This feature brings the advantage of reusing existing logic programming and DL systems in order to build an implementation of dl-programs.

Next, we provide the syntax of dl-programs and an overview of the semantics. More details can be found in [Eiter et al., 2007b].

Syntax and semantics of dl-programs

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* is either:

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

A *dl-atom* has the form

$$\mathrm{DL}[S_1\, op_1 p_1, \ldots, S_m\, op_m\; p_m; Q](\mathbf{t}), \qquad m \geq 0, \tag{2.2}$$

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 $p_1, \ldots, p_m$ *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

$$a \leftarrow b_1, \ldots, b_n, \mathrm{not}\ b_{n+1}, \ldots, \mathrm{not}\ b_m, \tag{2.3}$$

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.

The next example will illustrate our main ideas.

Example 2.7 ([Schindlauer, 2006])**.** 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*):

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

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

$$\begin{aligned}
&\mathit{newnode}(\mathit{add_1}). && (1)\\
&\mathit{newnode}(\mathit{add_2}). && (2)\\
&\mathit{overloaded}(X) \leftarrow \mathrm{DL}[\mathit{wired} \uplus \mathit{connect}; \mathit{HighTrafficNode}](X). && (3)\\
&\mathit{connect}(X, Y) \leftarrow \mathit{newnode}(X), \mathrm{DL}[\mathit{Node}](X), \mathrm{not}\ \mathit{overloaded}(Y), \mathrm{not}\ \mathit{excl}(X, Y). && (4)\\
&\mathit{excl}(X, Y) \leftarrow \mathit{connect}(X, Z), \mathrm{DL}[\mathit{Node}](Y), Y \neq Z. && (5)\\
&\mathit{excl}(X, Y) \leftarrow \mathit{connect}(Z, Y), \mathit{newnode}(Z), \mathit{newnode}(X), Z \neq X. && (6)\\
&\mathit{excl}(\mathit{add_1}, n_4). && (7)
\end{aligned}$$

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 *add₁* must not be connected with n_4.

Two different semantics have been defined for dl-programs, the (strong) answer-set semantics [Eiter et al., 2004a] and the well-founded semantics [Eiter et al., 2004b]. The latter extends the well-founded semantics of [Van Gelder et al., 1991] to dl-programs. Well-founded semantics is based on the notion of greatest unfounded set and assigns a single three-valued model to every logic program. In addition, recent results define the well-founded semantics to a subclass of Hybrid MKNF KBs [Knorr et al., 2007].

2.6 cq-Programs

As we can see in Section 2.5, queries in dl-programs are restricted to either concept and role membership queries or subsumption queries. Since the semantics of logic programs is usually defined over a domain of explicit individuals, this approach may fail to derive certain consequences implicitly contained in a DL-KB. To overcome this limitation, cq-programs were introduced in [Eiter et al., 2007a] as an extension of dl-programs by allowing (union of) conjunctive queries in dl-atoms. In this section, we will give a quick overview of the new contribution of cq-programs compared to dl-programs. For more details about cq-programs, we refer the reader to [Krennwallner, 2007]

Syntax and semantics of cq-programs

Informally, a cq-program consists of a DL-KB L and a generalized disjunctive program P, which may involve queries to L. Roughly, such a query may ask whether a specific description logic axiom, a conjunction or a union of conjunctions of DL axioms is entailed by L or not.

dl-queries are defined as in Section 2.5. An extension called *conjunctive query* (CQ) $q(\vec{X})$ is defined as follows:

$$\{\vec{X} \mid Q_1(\vec{X}_1), Q_2(\vec{X}_2), \ldots, Q_n(\vec{X}_n)\}, \tag{2.4}$$

where $n \geq 0$, each Q_i is a concept or role expression and each $\vec{X}_i$ is a singleton or pair of variables and individuals matching the arity of Q_i, and where $\vec{X} \subseteq \bigcup_{i=1}^{n} \text{vars}(\vec{X}_i)$ are its *distinguished* (or *output*) variables.

A *union of conjunctive queries* (UCQ) $q(\vec{X})$ is an expression of form

$$\{\vec{X} \mid q_1(\vec{X}) \vee \cdots \vee q_m(\vec{X})\} \tag{2.5}$$

of CQs $q_i(\vec{X})$ for $m \geq 0$.

Intuitively, a CQ $q(\vec{X})$ is a conjunction $Q_1(\vec{X}_1) \wedge \cdots \wedge Q_n(\vec{X}_n)$ of concept and role expressions with possibly existential quantified variables, which is true if all conjuncts are satisfied. A UCQ $q(\vec{X})$ is satisfied, whenever some $q_i(\vec{X})$ is satisfied. We will omit the output variables $\vec{X}$ from CQs and UCQs if clear from the context, especially when (U)CQs are used in dl-atoms. Here, the output of the dl-atom and of the (U)CQ are equal.

An extended dl-atom is of form

$$\text{DL}[\lambda; q](\vec{X}), \tag{2.6}$$

where $\lambda = S_1\, op_1\, p_1, \ldots, S_m\, op_m\, p_m$ $(m \geq 0)$ is a list of expressions $S_i\, op_i\, p_i$ called *input list*, each S_i is either a concept or a role, $op_i \in \{\uplus, \cup\!\!\!-, \cap\!\!\!-\}$, p_i is a predicate symbol matching the arity of S_i, and q is a (U)CQ with output variables $\vec{X}$ (in this case, (2.6) is called a *(u)cq-atom*), or $q(\vec{X})$ is a dl-query. Each p_i is an *input predicate symbol*; intuitively, $op_i = \uplus$ increases S_i by the extension of p_i, while $op_i = \cup\!\!\!-$ increases $\neg S_i$; $op_i = \cap\!\!\!-$ constrains S_i to p_i.

A *cq-rule* r is of the form

$$a_1 \vee \cdots \vee a_k \leftarrow b_1, \ldots, b_m, \text{not}\, b_{m+1}, \ldots, \text{not}\, b_n, \tag{2.7}$$

where every a_i is a literal and every b_j is either a literal or a dl-atom. We define $H(r) = \{a_1, \ldots, a_k\}$ and $B(r) = B^+(r) \cup B^-(r)$, where $B^+(r) = \{b_1, \ldots, b_m\}$ and $B^-(r) = \{b_{m+1}, \ldots, b_n\}$. If $B(r) = \emptyset$ and $H(r) \neq \emptyset$, then r is a *fact*. If $H(r) = \emptyset$ and $B(r) \neq \emptyset$, then r is a *constraint*. We denote by $B^+_{dl}(r)$ (resp., $B^-_{dl}(r)$) the set of all dl-atoms occurring in $B^+(r)$ (resp., $B^-(r)$).

A *cq-program* $KB = (L, P)$ consists of a DL-KB L and a finite set of cq-rules P.

Example 2.8. [Eiter et al., 2007a] The following example compares dl-programs and cq-programs on a same description logic knowledge base L as follows:

$$L = \left\{ \begin{array}{c} \mathit{father} \sqsubseteq \mathit{parent}, \exists \mathit{father}.\exists \mathit{father}^-.\{\mathit{Remus}\}(\mathit{Romulus}), \\ \mathit{hates}(\mathit{Cain}, \mathit{Abel}), \mathit{hates}(\mathit{Romulus}, \mathit{Remus}), \\ \mathit{father}(\mathit{Cain}, \mathit{Adam}), \mathit{father}(\mathit{Abel}, \mathit{Adam}) \end{array} \right\}$$

Apart from the explicit facts, L states that each *father* is also a *parent* and that Romulus and Remus have a common father. To represent that a *BadChild* is an individual who hates a sibling in terms of dl-programs, we have:

$$P = \{\mathit{BadChild}(X) \leftarrow \mathrm{DL}[\mathit{parent}](X,Z), \mathrm{DL}[\mathit{parent}](Y,Z), \mathrm{DL}[\mathit{hates}](X,Y).\}$$

This dl-program can conclude *BadChild(Cain)*, but not *BadChild(Romulus)*, though it is implicitly stated that Romulus and Remus have a common father. The following cq-program solves the problem:

$$P' = \{\mathit{BadChild}(X) \leftarrow \mathrm{DL}[\mathit{parent}(X,Z), \mathit{parent}(Y,Z), \mathit{hates}(X,Y)](X,Y).\}$$

Here, the body of the rule is a CQ $\{\mathit{parent}(X,Z),\ \mathit{parent}(Y,Z),\ \mathit{hates}(X,Y)\}$ over L with distinguished variables X and Y. In the case of Romulus and Remus, Z is their implicit father which is recognized in the DL-KB, but is not mentioned in the Herbrand Base of P'.

The strong answer-set semantics for cq-programs was defined in [Krennwallner, 2007], based on the minimal-model semantics for positive cq-programs.

2.7 Default Logic

Default Logic, one of the most prominent nonmonotonic reasoning formalisms, was introduced in [Reiter, 1980]. Since then, it has been widely studied and several variants have been developed. However, the original presentation is still the most attractive one, because of its simplicity and close intuition to common-sense reasoning. In this thesis, we are also interested in this version and our purpose is to enable default reasoning on top of ontologies, which can be seen as an attempt for integrating rules and ontologies to allow nonmonotonic reasoning.

In this section, we provide the basic definitions default logic. For an excellent tutorial on default logics, we refer the reader to [Antoniou, 1999].

2.7.1 Syntax and Semantics of Default Logic

Definition 2.1. [Reiter, 1980] A *default* δ is an inference rule of the form

$$\frac{\alpha(\overline{x}) : \beta_1(\overline{x}), \ldots, \beta_n(\overline{x})}{\gamma(\overline{x})}, \qquad (n \geq 1);$$

where $\alpha(\overline{x}), \beta_1(\overline{x}), \ldots, \beta_n(\overline{x})$, and $\gamma(\overline{x})$ are formulas in a first-order language $\mathcal{L}$. The formula $\alpha(\overline{x})$ is called the *prerequisite*, $\beta_1(\overline{x}), \ldots, \beta_n(\overline{x})$ are called the *justifications*, and $\gamma(\overline{x})$ the *consequent/conclusion* of the default.

The intuitive meaning of δ is: "For all individuals $\overline{x} = (x_1, \ldots, x_m)$, if $\alpha(\overline{x})$ is provable, and assuming that, for each i, $\beta_i(\overline{x})$ is true does not lead to inconsistency then conclude $\gamma(\overline{x})$."

If $n = 0$, δ is called *justification-free* and become a monotonic rule.

Definition 2.2. [Reiter, 1980] A *default theory* is a pair $T = \langle W, D\rangle$, where W is a set of first-order sentences, *axioms of* T, and D is a set of defaults.

Element of W are the *premisses* of T, representing *certain* yet *incomplete* information about the world. D represents *plausible* although *not necessarily true* conclusions, i.e., conclusions that hold typically.

Example 2.9. The default representing that *"Birds usually fly"* is

$$\frac{bird(X) : flies(X)}{flies(X)}$$

and with the constant *tweety* in the language $\mathcal{L}$, the corresponding instance of this default is:

$$\frac{bird(tweety) : flies(tweety)}{flies(tweety)}$$

Knowing that *"Tweety is a bird,"* we can conclude from this default that *"Tweety flies."* However, if the fact $\neg flies(tweety)$ can be derived from an updated theory, for example *"Tweety is a penguin,"* then it will block the default, since assuming $flies(tweety)$ will lead to inconsistency. Hence, we can no longer conclude $flies(tweety)$.

Example 2.10. The informal discussion in Example 2.9 can be represented in a default theory $T = \langle W, D\rangle$, where

$$W = \left\{ \begin{array}{ccc} \forall x.\, penguin(x) & \rightarrow & bird(x) \\ \forall x.\, penguin(x) & \rightarrow & \neg flies(x) \\ bird(tweety) & & \end{array} \right\},$$

and

$$D = \left\{ \tfrac{bird(X):flies(X)}{flies(X)} \right\}$$

We have $T \models flies(tweety)$. Now consider another default theory $T' = \langle W', D\rangle$, where $W' = W \cup \{penguin(tweety)\}$, we can conclude $T \models \neg flies(tweety)$ and the default in D is not applicable anymore.

A default $\frac{\alpha(\overline{x}):\beta_1(\overline{x}),\ldots,\beta_n(\overline{x})}{\gamma(\overline{x})}$ is called *open* iff at least one of $\alpha(\overline{x}), \beta_1(\overline{x}), \ldots, \beta_n(\overline{x}), \gamma(\overline{x})$ contains a free variable; otherwise, it is called *closed.* A default theory is called *open* iff it contains at least one open default; otherwise, it is said to be *closed.*

An *instance* of an open default is the result of uniformly replacing all free occurrences of variables by ground terms. More specifically, an instance of an open default is any closed default of the form $\frac{\alpha(\overline{c}):\beta_1(\overline{c}),\ldots,\beta_n(\overline{c})}{\gamma(\overline{c})}$, where $\overline{c}$ is an n-tuple of ground terms.

Since an open default can be identified with the set of all its closed instances, therefore it is sufficient for us to consider only closed default theories from now on.

In order to represent the totality of knowledge induced by a default theory T, we use the notion of *extension.* The basic idea of obtaining extensions is to apply the defaults in D to the premisses in W to extend certain knowledge by plausible conclusions; and apply the defaults as long as possible, until no new knowledge can be generated. The result is an extension of T.

The definition of extensions uses the definition of the Γ operator.

Definition 2.3. [Reiter, 1980] Let $T = \langle W, D\rangle$ be a closed default theory over a first-order language $\mathcal{L}$. For any set of sentences $S \subseteq \mathcal{L}$, let $\Gamma_T(S)$ be the smallest set of sentences from $\mathcal{L}$ satisfying the following properties:

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

As usual, $\vdash$ denotes the classical derivability relation and $Cn(\mathcal{F}) = \{\phi \,|\, \mathcal{F} \vdash \phi$ and ϕ is closed $\}$, for every set $\mathcal{F}$ of closed formulas.

Definition 2.4. A set of sentences $E \subseteq \mathcal{L}$ is an extension of T iff $E = \Gamma(E)$, i.e., iff E is a fixed-point of the operator Γ_T.

Example 2.11. The extension of the default theory T in Example 2.10 is $E = Cn(W \cup flies(tweety))$.

Definition 2.5. [Reiter, 1980] Let T be a closed default theory and suppose that E is an extension of T. The set of *generating defaults* for E *wrt* T, written $GD(E,T)$, is defined by

$$GD(E,T) = \{(\alpha : \beta_1, \dots, \beta_n/\gamma) \in D : \alpha \in E \text{ and } \beta_1 \notin E, \dots, \beta_n \notin E\}.$$

2.7.2 Algorithms for evaluating extensions of a default theory

Next, we present three different algorithms used to evaluate extensions of a default theory [Cholewinski and Truszczynski], namely *Select-defaults-and-check*, *Select-justifications-and-check*, and *Select-ordering-and-check*. They were all proved to find all extensions of a default theory. The first two will be used in our transformations provided in Chapter 3.

Algorithm 1: Select-defaults-and-check

1. Select a set of defaults $S \subseteq D$
2. Check if S is a set of generating defaults. If so, output the theory generated by S as an extension
3. Repeat until all subsets of D are considered or pruned

Algorithm 2: Select-justifications-and-check

1. Select a set of justifications $J \subseteq j(D)$, where $j(D)$ is the set of justifications of all defaults in D.
2. Find a set of defaults S whose justifications belong to J.
3. Compute the set of consequences E of W that can be derived by means of defaults in S (a default "fires" if its prerequisite has been derived earlier)
4. If all justifications in J are consistent with E and every default not in S has at least one justification not consistent with E, then output E as an extension.
5. Repeat until all subsets of $j(D)$ are considered or pruned.

As mentioned earlier, the purpose of this thesis is to enable default reasoning on top of ontologies. To fulfil this purpose, what we need is the mechanisms that supports ontology reasoning and allows the integration of (nonmonotonic) rules and ontologies

Such mechanisms are implemented and tools are ready for use. On the one hand, natural candidates for ontology reasoning are Racer and Pellet; on the other hand, the HEX-program solver dlvhex has been implemented as a prover for Semantics Web reasoning

Algorithm 3: Select-ordering-and-check

```
1. Select a permutation $d_1, \dots, d_n$ of $D$
2. Mark all defaults in $D$ *available*
3. Set $S := W$
4. Repeat until no longer possible: find the smallest $i$ such that $d_i$ is marked *available*
   and is applicable w.r.t. $S$ (i.e., the prerequisite of $d_i$ is a consequence of $S$ and every
   justifications of $d_i$ is consistent with $S$). Mark $d_i$ used and add its consequent to $S$.
5. If every justification of every used default is consistent with $S$, output $Cn(S)$ as an
   extension.
6. Repeat all these steps until all permutations are used.
```

under Answer-Set Semantics. Furthermore, dlvhex provides a plugin environment that allows evaluating external atoms, including dl-atoms in which we are interested.

With all the foundations settled, our task is pretty much straightforward. The strategy is (i) find some transformations from default logic over description logics to dl-programs such that extensions of a default theory will be corresponding to answer sets of the translated dl-program (Section 2.5); then (ii) implement these transformations in the plugin environment provided by dlvhex. These two steps will be discussed in Chapters 3 and 4, respectively.

To finish this chapter, we discuss the first attempt of embedding defaults into terminological knowledge representation proposed in [Baader and Hollunder, 1993] and point out the difference between this approach and our work here.

In this approach, the authors directly applied the process of evaluating extensions in a DL-KB based on the *Compute-All-Extension(W,D)* procedure for computing the set of generating defaults of a default theory $\langle W, D \rangle$. Specifically, this procedure requires the following subprocedures:

(1) Decide whether W is consistent.

(2) Compute all maximal subsets D' of D such that $W \cup Con(D')$ is consistent, where $Con(D')$ is the set of consequents of all defaults in D'.

(3) Compute the largest subset D_0 of D' that is grounded in W.

(4) Compute all maximal subset D'' of D_0 such that $W \cup Con(D'') \not\models \neg\beta_i$, where β_i is a justification of a default in D.

When applied to a terminological default theories, subprocedures (2) and (4) are solved by algorithms for computing minimal inconsistent and maximal consistent ABoxes (see Section 6 [Baader and Hollunder, 1993]). The algorithms are extensions of the tableaux-based consistency algorithms for ABoxes [Baader and Hanschke, 1991, Junker and Konolige, 1990] and were proved to be decidable for description logics $\mathcal{ALC}$ and $\mathcal{ALCF}$.

Furthermore, this approach allows concept definitions to appear in prerequisites, premises and consequents of defaults, thus defaults of the following form can be specified:

$$\frac{doctor : \exists child.doctor}{\exists child.doctor}$$

Compared to our approach, we allow concept and role names to appear as predicate names for these components so that we can adopt defaults of the form

$$\frac{GraduateStudent(X) \wedge hasScholarship(X, S) : \neg Assistant(X)}{\neg Assistant(X)}$$

where *GraduateStudent* and *Assistant* are concept names and *hasScholarship* is a role name in a DL-KB.

Another difference is that our approach exploits the dl-safety condition of dl-programs, the separated integration of rules and ontologies, hence allow as expressive DL-KBs as possible in the terminology knowledge base, as long as querying to these DL-KBs is decidable while [Baader and Hollunder, 1993] limits its applicability in description logics $\mathcal{ALC}$ and $\mathcal{ALCF}$.

Finally, both approaches do not apply skolemization, in other words, they work under the convention that only explicit individuals from the DL-KBs are considered to ground open defaults in a default theory, i.e., in the end actually process a set of finitely many grounded defaults.

3

Embedding Defaults over Description Logics into dl-Programs

In this chapter, we describe three transformations embedding default theories over description logics to dl-programs. We start by analyzing the transformation proposed in [Eiter et al., 2007b] which brought us the motivation for the second transformation. Both of them share the *Select-defaults-and-check* algorithm. Finally, the third transformation based on the *Select-justifications-and-check* algorithm will be presented.

Before presenting the transformations in details, we first introduce some basic notions which will be used in the rest of this chapter.

As we recalled in Section 2.7, a default theory T is defined as a pair $\langle W, D\rangle$ in which W, the background theory, is a set of first-order sentences and D is a set of defaults. However, with the purpose of enabling default reasoning on top of ontologies, we need to modify our default theories in a way that the background theory represents an ontology.

Definition 3.1. Let L be a DL-KB, a *default δ over L* has the form:

$$\frac{\alpha_1(\vec{X_1}) \wedge \cdots \wedge \alpha_k(\vec{X_k}) : \beta_1(\vec{Y_1}), \ldots, \beta_m(\vec{Y_m})}{\gamma(\vec{Z})} \tag{3.1}$$

where $\alpha_i, \beta_j, \gamma$ are either concept names or role names in L, i.e., either unary or binary. A set of defaults $D = \{\delta_1, \ldots, \delta_n\}$ is called *over L* iff each of its default is *over L*. A default theory $\Delta = \langle L, D\rangle$ is called *over L* iff the set of defaults D is *over L*.

Since we only consider default theories over a DL-KB in this thesis, hence, from now on, we will use the term default theories for simplicity. Later, we will extend the structural complexity of defaults by allowing conjunctions in every component of each default, but the main idea will not change, i.e., each literal name in those components is taken from the concepts names and roles names of L. We inherit the notion of extension from default logic as in Definition 2.4.

Example 3.1. The tweety example can be formalized by a default theory $\Delta = \langle L, D\rangle$, where

$L = \left\{ \begin{array}{c} \mathit{Flier} \sqsubseteq \neg \mathit{NonFlier}, \mathit{Penguin} \sqsubseteq \mathit{Bird}, \\ \mathit{Penguin} \sqsubseteq \mathit{NonFlier}, \mathit{Penguin}(\mathit{tweety}) \end{array} \right\}$, and

$D = \left\{ \frac{\mathit{Bird}(X):\mathit{Flier}(X)}{\mathit{Flier}(X)} \right\}$

In the next sections, we will present different transformations from default theories to dl-programs, namely transformations Π, Ω, and Υ.

3.1 Transformation Π

This transformation was proposed in [Eiter et al., 2007b] and is based on the *Select-defaults-and-check* algorithm.

Definition 3.2. Let $\Delta\langle L, D\rangle$ be a default theory, $\delta \in D$ be a default of the form (3.1), the transformation $\Pi(\delta)$ consists of the following corresponding dl-rules:

(1) two rules for guessing whether the consequent $\gamma(\overrightarrow{Z})$ of δ belongs to the extension E:

$$in\_\gamma(\vec{Z}) \leftarrow \text{not } out\_\gamma(\vec{Z}). \tag{3.2}$$

$$out\_\gamma(\vec{Z}) \leftarrow \text{not } in\_\gamma(\vec{Z}). \tag{3.3}$$

(2) a rule which checks the compliance of the guess for E with L:

$$fail \leftarrow \text{DL}[\lambda'; \gamma](\vec{Z}), out\_\gamma(\vec{Z}), \text{not } fail. \tag{3.4}$$

where $\lambda' = \bigcup_{\delta_i \in D} \gamma\, op_i\, in\_\gamma_i$, $op_i = \uplus$ if the literal $\gamma_i(\vec{Y_i})$ is positive and $op_i = \cup\!\!\!\!-$ otherwise;

(3) a rule for applying δ as in $\Gamma_\Delta(E)$:

$$\begin{aligned} p\_\gamma(\vec{Z}) \leftarrow & \text{DL}[\lambda; \alpha_1](\vec{X_1}), \ldots, \text{DL}[\lambda; \alpha_k](\vec{X_k}), \\ & \text{not DL}[\lambda'; \neg\beta_1](\vec{Y_1}), \ldots, \text{not DL}[\lambda'; \neg\beta_m](\vec{Y_m}), \end{aligned} \tag{3.5}$$

where $\lambda = \bigcup_{\delta_i \in D} \gamma_i\, op_i\, p\_\gamma_i$, $op_i = \uplus$ if the literal $\gamma_i(\vec{Y_i})$ is positive, and $op_i = \cup\!\!\!\!-$ otherwise, (and double negation in $\neg\beta_j$ is canceled).

(4) rules which check whether E and $\Gamma_\Delta(E)$ coincide:

$$fail \leftarrow \text{not DL}[\lambda; \gamma](\vec{Z}), in\_\gamma(\vec{Z}), \text{not } fail. \tag{3.6}$$

$$fail \leftarrow \text{DL}[\lambda; \gamma](\vec{Z}), out\_\gamma(\vec{Z}), \text{not } fail. \tag{3.7}$$

Definition 3.3. Let $\Delta = \langle L, D\rangle$ be a default theory, then KB^{df}_{Π} is the dl-program (L, P), where

$$P = \bigcup_{\delta \in D} \Pi(\delta)$$

Here after, we use $Cn(L(I; \lambda))$ for $Cn(L \cup \lambda(I))$, which means the deductive closure of the DL-KB L updated by the input list λ and the interpretation I.

Theorem 3.2. [Eiter et al., 2007b] Let $\Delta = \langle L, D\rangle$ be a default theory. Then:

(1) For each extension E of Δ, there exists a (unique) strong answer set M of KB^{df}_{Π} such that

$$E = Cn(L(M; \lambda')) = Cn(L(M; \lambda))$$

(2) For each strong answer set M of KB^{df}_{Π}, the set

$$E = Cn(L(M; \lambda')) = Cn(L(M; \lambda))$$

is an extension of Δ.

The proof for this theorem can be found in Appendix B, [Eiter et al., 2007b].

The intuition behind this transformation is the following. First of all, for each default, we make a guess whether its conclusion is *in* an extension. The guess is then used in λ' to check the compliance of it with the original DL-KB. Rule (3.4), which is a constraint, does not allow the cases in which we guess something *out* but actually it can be concluded from L. Having a compliant guess E, we can apply the Γ_Δ operator as in Definition (2.3) to compute $\Gamma_\Delta(E)$. Notice that in rule (3.5), we use another update, namely λ which uses auxiliary predicates whose names start with the prefix *"p_"* as its input. Finally, the two constraints (3.6) and (3.7) assert an extension by checking the condition $E = \Gamma_\Delta(E)$. Basically, constraint (3.6) says that we cannot have a case in which we guess something *in* which cannot be derived from L; and (3.7) prevents situation when we guess something *out* but which can actually be derived from L.

Analyzing this transformation, we can see that it has two explicit guessing rules (3.2) and (3.3) for auxiliary predicates whose names start with *"in_"*. Notice that these predicates denote the same meaning as the ones whose names start with *"p_"*, and under answer-set semantics, another implicit guess needs to be made in order to evaluate the program.

The question here is: can we simplify the transformation in a way that only one type of auxiliary predicates is needed, therefore the guess will be reduced and the transformation will be more effective.

Before answering this question and coming up with a new transformation, we describe here another issue of this transformation. Notice that rules (3.2) and (3.3) do not distinguish the cases where $\gamma_i(\vec{Z}_i)$ is positive or negative. Instead, this problem is only taken care of in defining λ and λ'. A problem then arises when we have defaults with dual consequents.

Example 3.3 (Nixon diamond)**.** A prominent example for this case is the *Nixon diamond* example in which default assumptions can lead to inconsistent conclusion. This scenario is as follows: usually, Quakers are pacifists, while Republicans usually are not. Richard Nixon is both a Quaker and a Republican. What can we conclude about him?

In terms of ontologies and default reasoning, we can represent this example by a default theory $\Delta = \langle L, D\rangle$, where $L = \{Q(nixon), R(nixon)\}$ and its signature contains concepts P, Q, and R. The default part is

$$D = \left\{ \delta_1 = \frac{Q(X) : P(X)}{P(X)},\ \delta_2 = \frac{R(X) : \neg P(X)}{\neg P(X)} \right\}$$

Now, if we do apply the above transformation naively, the first two guessing rules produce the same result, namely:

$$\begin{aligned} in\_P(X) &\leftarrow \text{not } out\_P(X) \\ out\_P(X) &\leftarrow \text{not } in\_P(X) \end{aligned}$$

These rules lead to a very dangerous situation where $\lambda' = P \uplus in\_P, P \uminus in\_P$. The guess for consequents of both defaults are not distinguished. Furthermore, guessing any $P(a)$, where a is a constant, will lead to inconsistency of the updated DL-KB. This inconsistency will block all non-justification-free defaults in D and therefore throw some possible extensions away. For instance, in this "Nixon diamond" example, we get no extension with this transformation, while the correct result should be two extensions, namely $E_1 = \{Q(nixon), R(nixon), P(nixon)\}$ and $E_2 = \{Q(nixon), R(nixon), \neg P(nixon)\}$.

However, this problem can be solved by giving explicit auxiliary predicate names for positive and negative literals:

- let $name(\gamma)$ be the predicate name of the literal γ; and

- let aux_γ be $in\_name(\gamma)$ (resp., $in\_not\_name(\gamma)$) if γ is positive (resp., negative).

and a change has to be made for the input list $\lambda' = \bigcup_{\delta \in D} \gamma \uplus in\_name(\gamma), \gamma \cup\!\!\!\!- in\_not\_name(\gamma)$. p_γ is changed to p_aux_γ and $\lambda = \bigcup_{\delta \in D} \gamma \uplus p\_in\_name(\gamma), \gamma \cup\!\!\!\!- p\_in\_not\_name(\gamma)$.

The updated transformation $\Pi(\delta)$ corresponding to this new naming approach can be summarized as follows:

$$aux\_\gamma(\vec{Z}) \leftarrow \text{not } out\_aux\_\gamma(\vec{Z}). \tag{3.8}$$

$$out\_aux\_\gamma(\vec{Z}) \leftarrow \text{not } aux\_\gamma(\vec{Z}). \tag{3.9}$$

$$fail \leftarrow \mathrm{DL}[\lambda'; \gamma](\vec{Z}), out\_aux\_\gamma(\vec{Z}), \text{not } fail. \tag{3.10}$$

$$p\_\gamma(\vec{Z}) \leftarrow \mathrm{DL}[\lambda; \alpha_1](\vec{X}_1), \ldots, \mathrm{DL}[\lambda; \alpha_k](\vec{X}_k), \tag{3.11}$$
$$\text{not } \mathrm{DL}[\lambda'; \neg\beta_1](\vec{Y}_1), \ldots, \text{not } \mathrm{DL}[\lambda'; \neg\beta_m](\vec{Y}_m).$$

$$fail \leftarrow \text{not } \mathrm{DL}[\lambda; \gamma](\vec{Z}), aux\_\gamma(\vec{Z}), \text{not } fail. \tag{3.12}$$

$$fail \leftarrow \mathrm{DL}[\lambda; \gamma](\vec{Z}), out\_aux\_\gamma(\vec{Z}), \text{not } fail. \tag{3.13}$$

$$\tag{3.14}$$

where λ and λ' are as in Definition 3.2.

Theorem 3.2 still holds for this new transformation.

So far, we used ordinary dl-atoms in our transformation. As described in Section 2.6, using conjunctive queries might introduce more conclusions. And in cases of conjunctive prerequisite, we have another choice for rules (3.5) and (3.11) by exploiting CQs as follows:

$$p\_\gamma(\vec{Z}) \leftarrow \mathrm{DL}[\lambda; \alpha_1(\vec{X}_1), \ldots, \alpha_k(\vec{X}_k)](\vec{X}), \tag{3.15}$$
$$\text{not } \mathrm{DL}[\lambda'; \neg\beta_1](\vec{Y}_1), \ldots, \text{not } \mathrm{DL}[\lambda'; \neg\beta_m](\vec{Y}_m).$$

where $\vec{X} = \bigcup_{j=1}^{k} Vars(\vec{X}_j)$

3.2 Transformation Ω

In this section, we will discuss transformation Ω based on the same algorithm with this transformation, but much more simplified.

Being inspired by the analysis in Section 3.1, a new transformation is proposed here from default theories to dl-programs. The main idea of this transformation is to use only one input list λ instead of λ and λ', hence only the auxiliary predicates starting with *"p_"* are needed. However, to keep the intuition of the result, we will use the prefix *in_* instead.

Definition 3.4. Let $\Delta\langle L, D\rangle$ be a default theory, $\delta \in D$ be a default of the form (3.1), the transformation $\Omega(\delta)$ consists of the following corresponding dl-rule:

$$aux\_\gamma(\vec{Z}) \leftarrow \mathrm{DL}[\lambda; \alpha_1](\vec{X}_1), \ldots, \mathrm{DL}[\lambda; \alpha_k](\vec{X}_k), \tag{3.16}$$
$$\text{not } \mathrm{DL}[\lambda; \neg\beta_1](\vec{Y}_1), \ldots, \text{not } \mathrm{DL}[\lambda; \neg\beta_m](\vec{Y}_m).$$

where $\lambda = \bigcup_{\delta_i \in D} (\gamma_i \uplus in\_name(\gamma_i), \gamma_i \cup\!\!\!\!- in\_not\_name(\gamma_i))$

Definition 3.5. Let $\Delta = \langle L, D\rangle$ be a default theory, then KB^{df}_{Ω} is the dl-program (L, P), where

$$P = \bigcup_{\delta \in D} \Omega(\delta)$$

Theorem 3.4. Let $\Delta = \langle L, D\rangle$ be a default theory. Then:

(1) For each extension E of Δ, there exists a (unique) strong answer set M of KB^{df}_{Ω} such that
$$E = Cn(L(M;\lambda))$$

(2) For each strong answer set M of KB^{df}_{Ω}, the set
$$E = Cn(L(M;\lambda))$$
is an extension of Δ.

This transformation is quite intuitive and follows exactly the usual way of evaluating extensions in default theories: *"If the prerequisites can be derived, and the justifications can be consistently assumed, then the consequent can be concluded"*. The proof of Theorem 3.4 is given in Appendix A.1.

As in Section 3.1, we can also use CQs here instead of a list of dl-atoms to check the prerequisites. Rule (3.16) hence can be rewritten as follows:

$$\begin{aligned} aux\_\gamma(\vec{Z}) \leftarrow & \mathrm{DL}[\lambda; \alpha_1(\vec{X}_1), \ldots, \alpha_k(\vec{X}_k)](\vec{X}), \\ & \text{not } \mathrm{DL}[\lambda; \neg\beta_1(\vec{Y}_1)](\vec{Y}_1), \ldots, \text{not } \mathrm{DL}[\lambda; \neg\beta_m(\vec{Y}_m)](\vec{Y}_m). \end{aligned} \tag{3.17}$$

Next, we will see how more complex defaults can be transformed. Consider the case when the consequent is a conjunction, i.e., $\gamma(\vec{Z}) = \gamma_1(\vec{Z}_1) \wedge \cdots \wedge \gamma_n(\vec{Z}_n)$.

The idea in this case is to introduce an auxiliary predicate name for concluding that the whole conclusion is *in*, and then to conclude that each literal belonging to that conjunction has to be *in*. We have more than one rule in this case:

$$\begin{aligned} all\_in\_\gamma(\vec{Z}) &\leftarrow \mathrm{DL}[\lambda; \alpha_1(\vec{X}_1), \ldots, \alpha_k(\vec{X}_k)](\vec{X}), \\ &\qquad \text{not } \mathrm{DL}[\lambda; \neg\beta_1](\vec{Y}_1), \ldots, \text{not } \mathrm{DL}[\lambda; \neg\beta_m](\vec{Y}_m). \\ aux\_\gamma_1(\vec{Z}_1) &\leftarrow all\_in\_\gamma(\vec{Z}). \\ &\vdots \\ aux\_\gamma_n(\vec{Z}_n) &\leftarrow all\_in\_\gamma(\vec{Z}). \end{aligned}$$
where $Z = \bigcup_{1\leq j\leq n} Vars(\vec{Z}_j)$

Finally, we go one step forward by allowing justifications to be conjunctions. Assume that $\beta_j(\vec{Y}_j) = \mu_{j,1}(\vec{Y}_{j,1}) \wedge \cdots \wedge \mu_{j,l_j}(\vec{Y}_{j,l_j})$. Checking the consistency condition for β_j in this case requires us to pose a union of conjunctive query (UCQ) to the knowledge base:

$$\begin{aligned} all\_in\_\gamma(\vec{Z}) &\leftarrow \mathrm{DL}[\lambda; \alpha_1(\vec{X}_1), \ldots, \alpha_k(\vec{X}_k)](\vec{X}), \\ &\qquad \text{not } \mathrm{DL}[\lambda; \neg\beta_1(\vec{Y}_1)](\vec{Y}_1), \\ &\qquad\qquad \vdots \\ &\qquad \text{not } \mathrm{DL}[\lambda; \neg\mu_{j,1}(\vec{Y}_{j,1}) \vee \cdots \vee \neg\mu_{j,l_j}(\vec{Y}_{j,l_j})](\vec{Y}_j), \\ &\qquad\qquad \vdots \\ &\qquad \text{not } \mathrm{DL}[\lambda; \neg\beta_m(\vec{Y}_m)](\vec{Y}_m). \\ aux\_\gamma_1(\vec{Z}_1) &\leftarrow all\_in\_\gamma(\vec{Z}). \\ &\cdots \\ aux\_\gamma_n(\vec{Z}_n) &\leftarrow all\_in\_\gamma(\vec{Z}). \end{aligned}$$

Theorem 3.4 still holds for these updated transformations.

Notice that in general cases, asking a UCQ to the knowledge base is undecidable for expressive DLs [Rosati, 2007], but in active domain semantics where we just consider all explicit individuals of the knowledge base, this problem is decidable.

3.3 Transformation Υ

In this section, we provide another transformation based on a different algorithm for evaluating extensions of default theories, the *Select-justifications-and-check* algorithm (see Algorithm 2). In this algorithm, the guessing part concentrates on whether a justification of a default is consistent with an extension. Hence, we introduce the following auxiliary predicate names for this purpose: let $auxc\_\beta_j$ be $cons\_name(\beta_j)$ (resp., $cons\_not\_name(\beta_j)$) if β_j is a positive (resp., negative) literal.

Definition 3.6. Let $\Delta\langle L, D\rangle$ be a default theory, $\delta \in D$ be a default of the form (3.1), the transformation $\Pi(\delta)$ consists of the following corresponding dl-rules:

(1) rules that guess whether a justification is consistent with the extension E:

$$auxc\_\beta_j(\vec{Y_j}) \leftarrow \text{not } out\_auxc\_\beta_j(\vec{Y_j}). \tag{3.18}$$

$$out\_auxc\_\beta_j(\vec{Y_j}) \leftarrow \text{not } auxc\_\beta_j(\vec{Y_j}). \tag{3.19}$$

(2) a rule which computes the set of consequences E:

$$\begin{aligned} aux\_\gamma(\vec{Z}) \leftarrow &\mathrm{DL}[\lambda; \alpha_1](\vec{X_1}), \ldots, \mathrm{DL}[\lambda; \alpha_k](\vec{X_k}), \\ &auxc\_\beta_1(\vec{Y_1}), \ldots, auxc\_\beta_m(\vec{Y_m}). \end{aligned} \tag{3.20}$$

where $\lambda = \bigcup_{\delta_i \in D}(\gamma_i \uplus in\_name(\gamma_i), \gamma_i \,\cup\!\!\!\!-\, in\_not\_name(\gamma_i))$

(3) rules that check the compliance of our guess with E:

$$fail \leftarrow \mathrm{DL}[\lambda; \neg\beta_j](\vec{Y_j}), auxc\_\beta_j(\vec{Y_j}), \text{ not } fail. \tag{3.21}$$

$$fail \leftarrow \text{not } \mathrm{DL}[\lambda; \neg\beta_j](\vec{Y_j}), out\_auxc\_\beta_j(\vec{Y_j}), \text{ not } fail. \tag{3.22}$$

Definition 3.7. Let $\Delta = \langle L, D\rangle$ be a default theory, then KB^{df}_{Υ} is the dl-program (L, P) where

$$P = \bigcup_{\delta \in D} \Upsilon(\delta)$$

Theorem 3.5. Let $\Delta = \langle L, D\rangle$ be a default theory. Then:

(1) For each extension E of Δ, there exists a (unique) strong answer set M of KB^{df}_{Υ} such that

$$E = Cn(L(M; \lambda))$$

(2) For each strong answer set M of KB^{df}_{Υ}, the set

$$E = Cn(L(M; \lambda))$$

is an extension of Δ.

The first two rules (3.18) and (3.19) make a choice for the consistency of each justification with the extension E. Then rule (3.20) applies the Γ_Δ operator. We can see that compared to rules (3.5) and (3.16), this rule poses less queries to the DL-KB; but we have to pay off by having more guesses inside the program part. Finally, rule (3.21) prevents interpretations being all models in which we guess a justification $\beta_i(\vec{c})$ to be consistent but can actually derive $\neg\beta_i(\vec{c})$; and rule (3.22) dismisses situations where a justification $\beta_i(\vec{c})$ is guessed to be inconsistent but we cannot derive $\neg\beta_i(\vec{c})$. A proof for theorem 3.5 is given in Appendix A.2.

Hereafter are some quick remarks for extensions to more complex defaults:

- defaults whose consequent is a conjunction: using the same strategy as in section 3.2, rule (3.20)'s head will be replaced by an auxiliary predicate *all_in_*, and then we add rules saying that each consequent's element will be concluded if the this auxiliary predicate is concluded.
- justifications that are conjunctions: the guessing rules in (3.18) will now mean that the whole conjunction is consistent with the extension; therefore, rules for checking the compliance in rules (3.21) and (3.22) must use UCQs instead. We also need to add rules saying that if the whole conjunction is consistent with E then each of its element is also consistent with E.

3.4 Pruning Rules for Optimization

This section will be based on investigating the relationship between pairs of defaults. In particular, consider the defaults

$$\delta_1 = \frac{\alpha_{1,1}(\vec{X}_{1,1}) \land \cdots \land \alpha_{1,k_1}(\vec{X}_{1,k_1}) : \beta_{1,1}(\vec{Y}_{1,1}), \ldots, \beta_{1,m_1}(\vec{Y}_{1,m_1})}{\gamma_{1,1}(\vec{Z}_{1,1}) \land \cdots \land \gamma_{1,n_1}(\vec{Z}_{1,n_1})},$$

and

$$\delta_2 = \frac{\alpha_{2,1}(\vec{X}_{2,1}) \land \cdots \land \alpha_{2,k_2}(\vec{X}_{2,k_2}) : \beta_{2,1}(\vec{Y}_{2,1}), \ldots, \beta_{2,m_2}(\vec{Y}_{2,m_2})}{\gamma_{2,1}(\vec{Z}_{2,1}) \land \cdots \land \gamma_{2,n_2}(\vec{Z}_{2,n_2})}$$

3.4.1 Forcing other defaults to be out

Based on consequent-consequent relationship

The "Nixon diamond" example motivated us to come up with an optimization technique for defaults with opposite consequents. In this example, we can see that the conclusion of each default blocks the other default. In order to prune such cases, we can pose a constraint saying that the two consequents can not be both in an extension. A more formal argument is as follows:

If there exists $\gamma_{1,i}(\vec{Z}_{1,i})$ and $\gamma_{2,j}(\vec{Z}_{2,j})$ such that

$$\gamma_{1,i}(\vec{Z}_{1,i}) \equiv \neg\gamma_{2,j}(\vec{Z}_{2,j}) \text{ where } (1 \leq i \leq n_1, 1 \leq j \leq n_2)$$

then $\mathit{all\_in\_def}_1(\vec{Z}_1)$ and $\mathit{all\_in\_def}_2(\vec{Z}_2)$ cannot stay together in an answer set. This can be done by adding the following constraint:

$$\mathit{fail} \leftarrow \mathit{all\_in\_def1}(\vec{Z}_1), \mathit{all\_in\_def2}(\vec{Z}_2), \text{not } \mathit{fail}. \tag{3.23}$$

where: $\vec{Z}_1 = \bigcup_{1 \leq i \leq n_1} \mathit{Terms}(\vec{Z}_{1,i})$ and $\vec{Z}_2 = \bigcup_{1 \leq i \leq n_2} \mathit{Terms}(\vec{Z}_{2,i})$.

Based on consequent-justification relationship

Not only can the relationships between consequents be exploited for our pruning purpose, the relations between consequents and justifications can also play a role. If there exist i and j $(1 \leq i \leq n_1, 1 \leq j \leq n_2)$ such that: $\gamma_{1,i}(\vec{Z}_{1,i}) \equiv \neg\beta_{2,j}(\vec{Y}_{2,j})$ then the consequent of the first default will block the second one, therefore we can pose the following constraint:

$$fail \leftarrow all\_in\_def_1(\vec{Z}_1), all\_in\_def_2(\vec{Z}_2), \text{not } fail. \tag{3.24}$$

$\vec{Z}_1$ and $\vec{Z}_2$ are defined as above.

3.4.2 Forcing other defaults to be in

Another optimizing technique is to consider defaults that can be forced to be *in* by other defaults. Consider two defaults, if the consequent of one default is contained in that of the other, then guessing the second default to be *in* an extension will force the first one to be *in* also. We can express this idea in a formal way as follows.

Let $\Gamma_1 = \{\gamma_{1,1}(\vec{Z}_{1,1}), \ldots, \gamma_{1,n_1}(\vec{Z}_{1,n_1})\}$ and $\Gamma_2 = \{\gamma_{2,1}(\vec{Z}_{2,1}), \ldots, \gamma_{2,n_1}(\vec{Z}_{2,n_2})\}$.
If $\Gamma_1 \subseteq \Gamma_2$, then guessing $all\_in\_def_2(\vec{Z}_2)$ to be true will forces $all\_in\_def_1(\vec{Z}_1)$ to be true. This can be done by adding the following rule:

$$all\_in\_def_1(\vec{Z}_1) \leftarrow all\_in\_def_2(\vec{Z}_2). \tag{3.25}$$

3.4.3 Defaults whose conclusions are already in the background theory

From the definition of extensions, we know that the background theory W is always contained in an extension E $(W \subseteq E)$. Exploiting this property can help us to make another shortcut in our transformation. We can state that some certain facts which are in the deductive closure of the background theory must be in the extension. There are two ways to do that: (i) either directly query to the knowledge base and put the facts $aux\_\gamma(\vec{c})$ whenever $\gamma(\vec{c})$ can be concluded from the knowledge base. γ here can be a literal for a concept or a role, $\vec{c}$ is a vector of constants which can have one or 2 members corresponding to γ; (ii) or we can add the following dl-rule to the transformation so that the querying will be done by the dl-plugin at run time:

$$aux\_\gamma(\vec{X}) \leftarrow \text{DL}[\gamma](\vec{X}). \tag{3.26}$$

4

Front-End

Having all the theoretical transformations from default theories to dl-rules in Chapter 3, our next step is to implement them in a framework which allows the use of dl- and cq-rules to provide a front-end supporting default reasoning over ontologies. Before going into the details of how the implementation was developed, we describe an overview of the front-end in the following section.

4.1 Front-End Overview

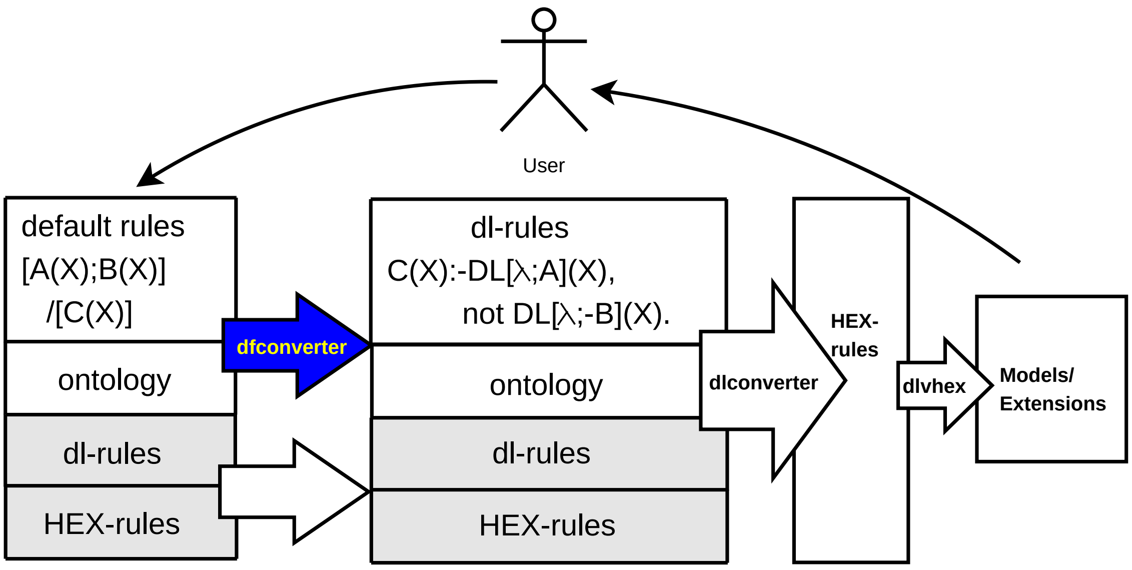


Figure 4.1: Strategy for implementing the df-converter

The front-end can be described as in Figure 4.1. Users provide input, including a set of default rules along with an ontology in an OWL file, and then get the extensions of the default theory in terms of answer sets. Other optional inputs can be either dl-rules or HEX-rules. Recall that dlvhex is a solver for HEX-programs under the answer-set semantics. It receives input in terms of HEX-rules and returns results as answer sets to users; hence, the level of HEX-programs can be considered transparent.[1] The reason why we do not need to know about HEX-rules is that the dl-plugin, which is a part of dlvhex's plugin environment, already supplies a dl-converter for converting dl-rules to HEX-rules. However, it does not prevent experts from providing more sophisticated input such as constraints in terms of dl-rules or HEX-rules to reduce the search space and speed up the evaluation. In fact, we do provide a technique called typing predicates in which users can specify more supportive information that helps our implementation gaining a significant improvement. Details on this technique will be presented in Section 4.3.

Our strategy is quite straightforward. What we need to implement is a df-converter whose input contains default rules accompanied with an ontology, optional dl-rules, or even HEX-rules. This converter then transforms all the default rules into dl-rules based on different transformations discussed in Chapter 3, with the help of the ontology serving as the sources of individuals for the domain predicates to guarantee the safety condition[2]. Then, the transformed dl-rules, along with other input dl- and HEX-rules, are transferred to the dl-converter. The rest of the evaluation will be done by the dl-plugin and dlvhex.

When reading this discussion, one can pose the question: *"What is the relationship between the df-converter and the dl-plugin?"* In fact, there are two possible answers: the df-converter can be either another plugin preceding the dl-plugin in solving default reasoning on top of ontologies, or it can be a component inside the dl-plugin which will be executed before the dl-converter. In a first attempt to implement this front-end, we were considering the first option. However, we realized that the df-converter and the dl-plugin share common functionality in communicating with DL reasoners, which is already available in the dl-plugin, hence we decided to take the second approach into account to make use of these functions. Section 4.4 will give more details on how the dl-plugin was changed to adopt the df-converter.

Next, we present the syntax of the input defaults and how typing predicates come into play, together with the *"Tweety bird"* example.

4.2 Syntax for Input Defaults

To make it easy to present the syntax for input defaults, we first describe the grammar in the simple case where namespace and typing predicates are not considered. Additional flavour will be discussed in an informal way by examples.

The syntax for input defaults in the basic case is as follows. We use square brackets to separate the prerequisites, justifications from the consequents. The prerequisites and justifications are split up by a semicolon; different justifications are split by commas, and the character "&" is used to represent the conjunction. The grammar for the input defaults is as follows; we assume that readers are familiar with the notion of literals (positive or negative predicates), which is denoted by *lit*:

$$just \equiv lit\,(\,\&\,lit)^*$$

$$[\underbrace{lit\,(\,\&\,lit)^*}_{prerequisites};\underbrace{just\,(\,,just)^*}_{justifications}]/[\underbrace{lit\,(\,\&\,lit)^*}_{consequents}]$$

The character '−' is used to represent strong negation.

[1]For more details about HEX-programs, we refer readers to [Schindlauer, 2006]

[2]For more details about the safety condition, we refer the reader to [Schindlauer, 2006]

Example 4.1. The default for the *"Tweety bird"* example is as follows:

```
[Bird(X);Flier(X)]/[Flier(X)]
```

Now comes a more complicated case. It is common that concept, role and individual names in an ontology can come from different namespaces, hence a syntax declaring namespace is necessary. Fortunately, such a namespace definition syntax is already available in dlvhex. The following line

```
#namespace("foo","http://foo.bar.com")
```

defines a namespace `foo` recognized by the URI `http://foo.bar.com`, and now we can use a qualified name such as `foo:P` instead of `P` to explicitly specify `P`'s namespace.

Example 4.2. The following text represents the *"Tweety bird"* example with namespace:

```
#namespace("tweety", "http://example/tweety_bird")

[tweety:Bird(X);tweety:Flier(X)]/[tweety:Flier(X)]
```

4.3 Typing Predicates

Before going to discuss this technique, we would like to remark the safety condition in every transformation. Since we intended to have our theoretical transformations in Chapter 3 to be compact, we did not mention this requirement so far. Basically, the safety condition says that every variable appearing in a rule must appear in a positive non-DL atom, so that every variable is identified explicitly. In the implementation in dlvhex, what we need to do is to add a domain predicate (*dom* for short) to the body of the rule for each variable appearing in it, and generate a fact $dom(a)$ for each constant/individual appearing in the knowledge base. For more details on safety condition, we refer the reader to [Schindlauer, 2006].

After doing some experiments in which roles from the DL-KB appear in default rules, we recognized that many unnecessary guesses were made during dlvhex' guessing phase. For example, suppose that in our DL-KB, we have property *hasScholarship* whose domain is *Student* and range is *Scholarship*. Moreover, *em* and *oad* are two scholarships, *min* and *tkren* are two students. If *hasScholarship* appears somewhere in a default, there will be a corresponding dl-atom $\mathrm{DL}[\lambda; hasScholarship](X, Y)$ in a dl-rule. In the guessing phase of dlvhex it tries all combinations of two individuals for this dl-atom, hence a ground dl-atom such as

$$\mathrm{DL}[\lambda; hasScholarship](em, oad)$$

will be considered, which is unnecessary in this situation. To reduce the number of individuals, which have to be taken into account for the reasoning process, we provide an advanced technique called typing predicate. Basically, we allow users to attach each default with a predicate whose name is freely determined and terms already appeared in that default's components. In order to restrict the search space for dlvhex, users then can specify all facts, or even a program, which computes the models for such a typing predicate. This method can be seen as providing a database for the reasoner to work more effectively. One effect of typing predicates is that they make our transformation theoretically incomplete, but still acceptable from a practical point of view, especially when users only care about individuals specified in those facts.

Formally speaking, a default now can be represented in the following form

$$\delta = \left\langle \frac{\alpha_1(\vec{X}_1) \wedge \cdots \wedge \alpha_k(\vec{X}_k) : \beta_1(\vec{Y}_1), \ldots, \beta_m(\vec{Y}_m)}{\gamma_1(\vec{Z}_1) \wedge \cdots \wedge \gamma_n(\vec{Z}_n)}\, ; \theta(\vec{W}) \right\rangle$$

where

$$\vec{W} \subseteq \bigcup_{i=1}^{k} \mathit{Vars}(\vec{X}_i) \cup \bigcup_{i=1}^{m} \mathit{Vars}(\vec{Y}_i) \cup \bigcup_{i=1}^{n} \mathit{Vars}(\vec{Z}_i)$$

and θ is the name of the typing predicate.

In all transformations, if a rule r satisfies the condition $\vec{W} \subseteq \mathit{Vars}(r)$ then all the predicates $dom(X)$ such that $X \in \vec{W}$ will be removed and be replaced by $\theta(\vec{W})$.

The last update for our syntax to support typing predicates is as follows. We allow $\langle lit \rangle$ to occur next to a default where *lit* is a literal whose variables already appeared in the default's prerequisites, justifications or consequents. To make this typing predicate work, users must specify facts or rules which compute models for *lit* in a hex file as the optional input in Figure 4.1 which will be called with a default file and an OWL file by dlvhex.

Example 4.3. If we are interested in a small selection of birds from the ontology, what we can do is to provide two files as follows:

The file `tweety.df` representing a default rule:

```
#namespace("tweety", "http://example/tweety_bird")

[tweety:Bird(X);tweety:Flier(X)]/[tweety:Flier(X)]<mybird(X)>
```

The file `tweety.hex` representing the facts:

```
mybird("<http://example/tweety_bird#tweety>").

mybird("<http://example/tweety_bird#joe>").
```

Assume that the ontology of this example is stored in the file `bird.owl`. We can run the following command from the console to start the computation of the answer sets.

```
$dlvhex --default=bird.df --ontology=bird.owl bird.hex
```

In this simple case, only one extension exists in which *tweety* flies and *joe* does not. In the syntax of the transformed dl-program, the results are differently represented in each transformation. Table 4.1 shows the differences.

The name `all_in_def_1`, `out_def_1`, and `all_p_def_1` need to be clarified here. Basically, we assign to each default an integer identifier starting from 1 and continuing increasing as we get new input defaults. In Section 3.2, we mentioned using *all_in_*γ as an auxiliary predicate name to deal with conjunctive consequents of a default. In the implementation, we use `all_in_def_`i where i is the identifier for the corresponding default for this purpose.

We can see from this table that the transformations Π and Υ conclude more facts about *joe* than transformation Ω, which can be directly explained from the transformations since Ω does not have explicit guesses for negative information, while Π and Υ do; however, Π creates guesses for consequents which are `in` an extension, while Υ guesses if a justification is *consistent* with an extension.

Transformation	Expected results
Π	`all_in_def_1(tweety)`, `in_Flier(tweety)`, `out_def_1(joe)` `all_p_def_1(tweety)`, `p_Flier(tweety)`
Ω	`all_in_def_1(tweety)`, `in_Flier(tweety)`
Υ	`all_in_def_1(tweety)`, `in_Flier(tweety)` `cons_Flier(tweety)`, `out_cons_Flier(joe)`

Table 4.1: Expected results in the Tweety bird example for different transformations

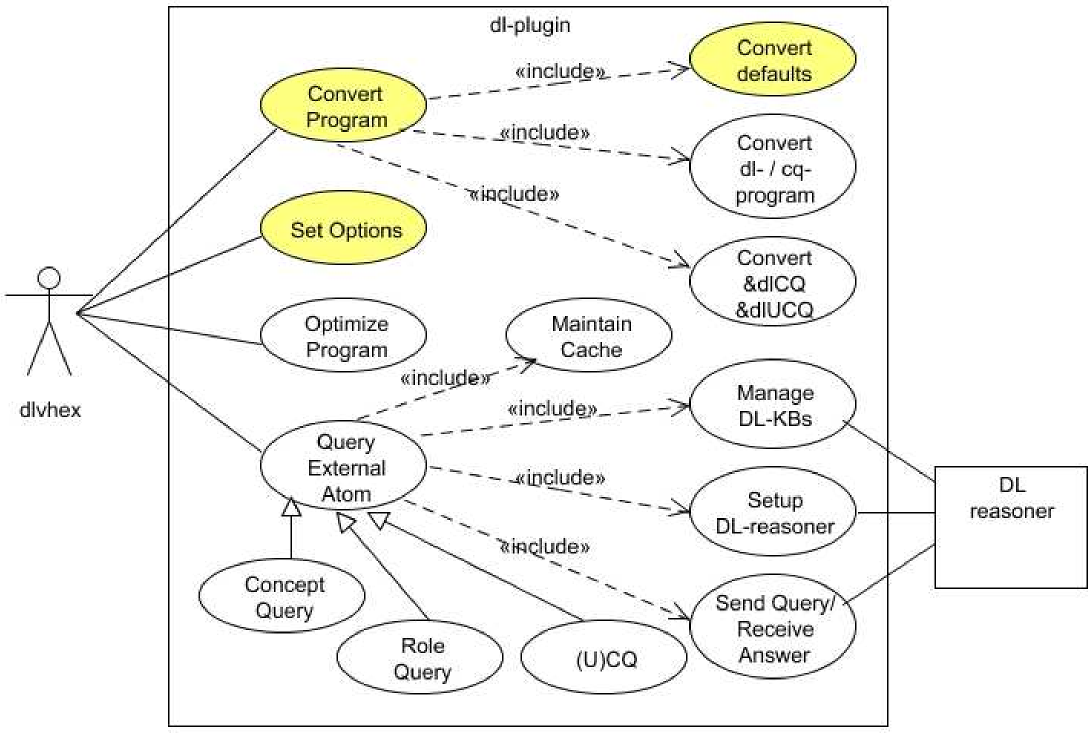


Figure 4.2: Use Case Diagram dl-plugin

Notice that in this table, for simplicity, we omitted all the domain facts, and the namespace of each individual. To be exact, the constant `tweety` is represented as `<http://example/tweety_bird#tweety>`. Therefore, the complete results from dlvhex will be more verbose than what is shown here.

4.4 Update of the dl-Plugin to Adopt the df-Converter

As mentioned in Section 4.1, our strategy is to build the df-converter as a new component in the dl-plugin, preceding the dl-converter. To fulfill this purpose, the following changes need to be made in the dl-plugin.

4.4.1 Update the dl-Plugin Use Cases

In [Krennwallner, 2007], the author presented the use case diagram of the dl-plugin with four use cases. The primary actor of this diagram is dlvhex while the support actor is

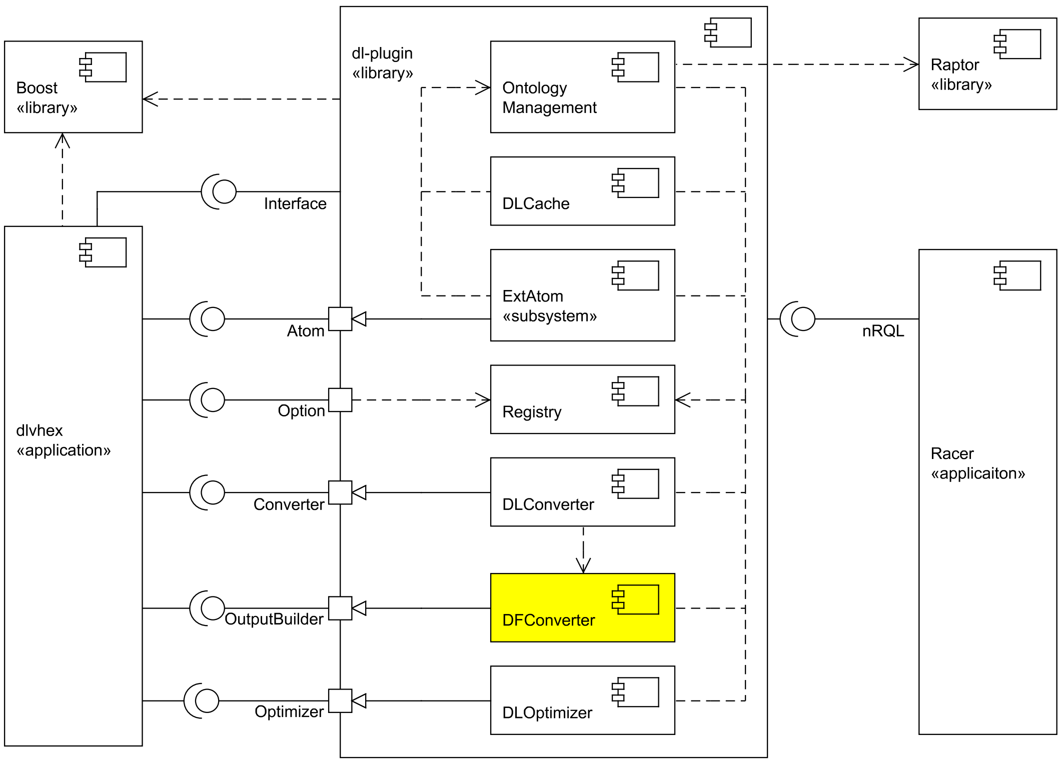


Figure 4.3: Component Diagram dl-plugin

the DL-reasoner. Under the addition of the df-converter, this diagram is updated as in Figure 4.2 in which all the updated/new use cases are shown in yellow. The changes compared to the original one are:

(i) Use case *"Set Options"*: we allow more options to be recognized in the dl-plugin; see Section 4.6 for details on these new options.

(ii) Use case *"Convert Program"*: this use case includes one new use case, the *"Convert defaults"* use case, which converts default rules to dl-rules.

4.4.2 Update the dl-plugin Components

The component diagram of the dl-plugin has been updated as in Figure 4.3. We have a new component named *DFConverter*. This component uses the component *Registry* to get the options and is used by *DLConverter* as the first step of converting default rules to HEX-rules. Moreover, it is also a good idea to implement the communication between the *DFConverter* and dlvhex via the *OutputBuilder* interface to give the result according to the option of brave or cautious reasoning from users.

4.5 DFConverter Class Diagram

In this section, we look into the df-converter and see how its classes are organized. Figure 4.4 presents all the classes in this component, the relation between them, and the most important public methods that those classes provide. The basic classes are *Term*, *Terms*, *Unifier*, *Update*, and *Updates*. Then, at a one level higher, we have *Predicate*, *DLAtom*,

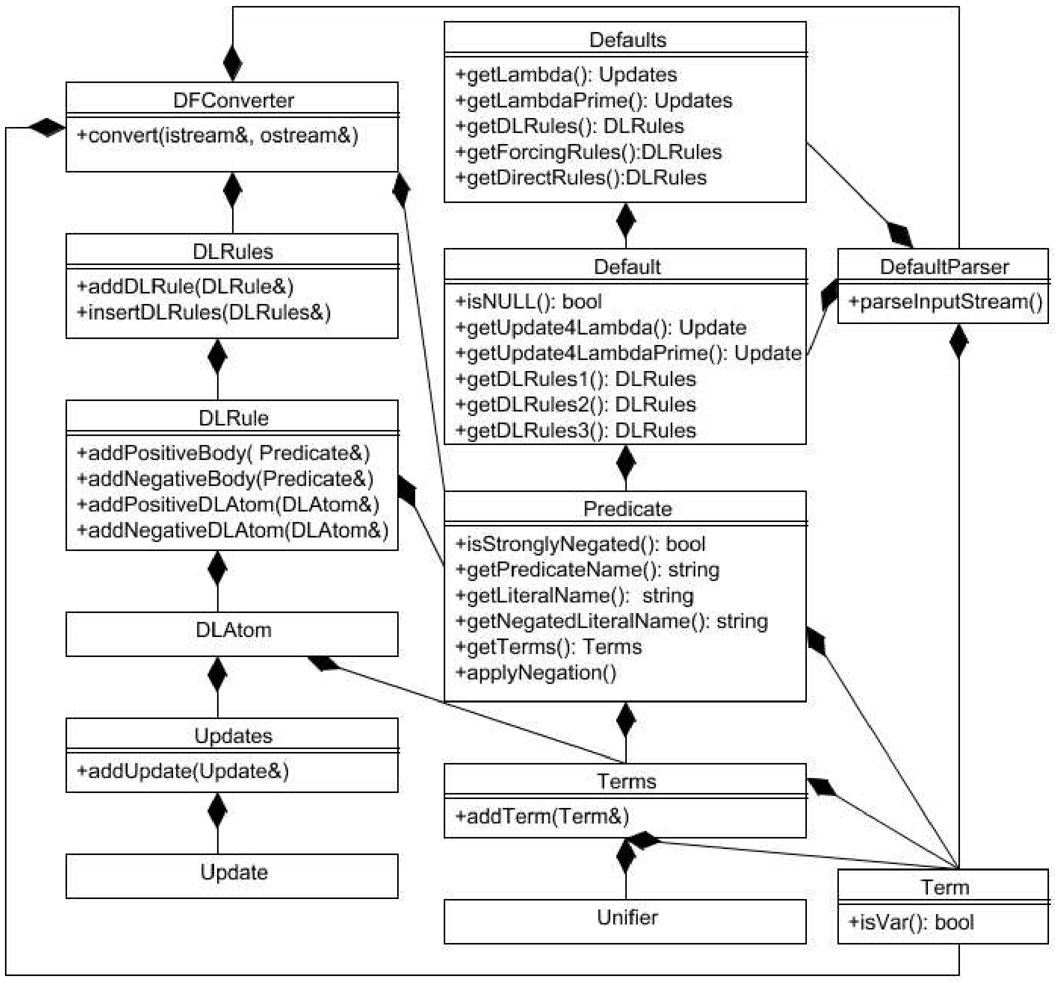

Defaults
+getLambda(): Updates
+getLambdaPrime(): Updates
+getDLRules(): DLRules
+getForcingRules():DLRules
+getDirectRules():DLRules
DFConverter
+convert(istream&, ostream&)
DLRules
+addDLRule(DLRule&)
+insertDLRules(DLRules&)
Default
+isNULL(): bool
+getUpdate4Lambda(): Update
+getUpdate4LambdaPrime(): Update
+getDLRules1(): DLRules
+getDLRules2(): DLRules
+getDLRules3(): DLRules
DefaultParser
+parseInputStream()
DLRule
+addPositiveBody(Predicate&)
+addNegativeBody(Predicate&)
+addPositiveDLAtom(DLAtom&)
+addNegativeDLAtom(DLAtom&)
Predicate
+isStronglyNegated(): bool
+getPredicateName(): string
+getLiteralName(): string
+getNegatedLiteralName(): string
+getTerms(): Terms
+applyNegation()
DLAtom
Updates
+addUpdate(Update&)
Terms
+addTerm(Term&)
Update
Unifier
Term
+isVar(): bool

Figure 4.4: Class Diagram df-converter

DLRule, *DLRules*, *Default*, and *Defaults*. Finally, we have *DefaultParser* and *DFConverter* on top.

The two most important classes are *DefaultParser* and *DFConverter*. A *DFConverter* object interacts with dl-plugin's interface to get all the related command line options, one of them is the name of the text file containing all the defaults. Then it calls a *DefaultParser* object to parse this set of defaults and store them in an internal structure of the df-converter, in objects of the type *Defaults* and *Default*. Those objects provide methods to return translated dl-rules to the *DFConverter* object. There are command line options to choose between different transformations, using pruning rules or not, etc. (see Section 4.6).

The *DFConverter* object finally queries the description logic knowledge base for all individuals in the ontology. Each individual will be wrapped in a *dom* predicate and these facts are added to the transformed dl-rules. At this point, we have a complete transformed program. This content is transferred to the dl-converter in an output stream afterwards.

4.6 Command line options

dlvhex supports command line options to the plugins. Each plugin can pick particular options for its own further processing purposes. The original set of command line options that the dl-plugin accepts was listed in [Krennwallner, 2007], Section 4.6.1. Hereafter we present the new options needed by the df-converter and one option from the original set that will be used in experiments (Section 5.3):

- `--default=FILENAME`: `FILENAME` is the name of the text file containing the defaults;
- `--dftrans=DFTRANS`: `DFTRANS` can be `1`, `2` or `3` to indicate the translation Π, Ω and Υ from defaults to dl-rules, respectively. By default, `DFTRANS` is set to `2`.
- `--dfpruning=DFP`: `DFP` can be either `on` or `off`, which identifies whether the user wishes to use pruning rules in the transformation or not. The pruning rules we are talking about were described in section 3.4. By default, `DFP` is set to `on`.
- `--dlopt=MOD[,MOD]*`: Setup particular optimization features according to the supplied list of modifiers `MOD`, which may be `-push` for disabling push optimizations and `-cache` for disabling the DL-Cache.

5

Applications

We start this chapter by describing all examples that have been tested, from the classical and simple ones in Section 5.1 to the more complex ones in Section 5.2. Each example will be described in the order of the DL-KB representation, the input defaults and typing predicates (if any), and the expected extensions. For complicated ontologies in Section 5.2, we will refer the reader to either an explanation in previous chapters or provide the ontology's original links available on the Internet.

In Section 5.3, we choose some examples to give experimental results on the sizes of the input, different transformations and different running modes, in particular whether using conjunctive queries or not. From these results, many interesting facts can be discovered to help us understand not only our transformations, but also the implementation of dlvhex and the dl-plugin. Furthermore, they suggest tasks for future work to improve the whole system.

5.1 Classical examples

5.1.1 Nixon Diamond

Intuitive meaning see Example 3.3

Description Logic Knowledge Base The DL-KB in this example has three concepts R, Q, and P as abbreviations for *Republican*, *Quaker* and *Pacifist*, respectively; and an individual *nixon* belonging to R and Q. In other words, $L = \{Q(nixon), R(nixon)\}$.

Defaults

```
[R(X);-P(X)]/[-P(X)]

[Q(X);P(X)]/[P(X)]
```

Expected result We have two extensions in this example, namely one in which *nixon* is a *Pacifist*, and the other in which *nixon* is not. The expected results are summarized in Table 5.1. Notice that the first default, which conclude `-P(X)`, has identifier 1, and the second default has identifier 2.

Table 5.1: Expected results in the Nixon Diamond example for different transformations

Transformation	**Expected results**	

Π	`all_in_def_1(nixon)`
	`all_p_def_1(nixon)`
	`in_not_P(nixon)`
	`out_def_2(nixon)`
		`all_in_def_2(nixon)`
		`all_p_def_2(nixon)`
		`in_P(nixon)`
		`out_def_1(nixon)`

Ω	`all_in_def_1(nixon)`
	`in_not_P(nixon)`
		`all_in_def_2(nixon)`
		`in_P(nixon)`

Υ	`all_in_def_1(nixon)`
	`in_not_P(nixon)`
	`cons_not_P(nixon)`
	`out_cons_P(nixon)`
		`all_in_def_2(nixon)`
		`in_P(nixon)`
		`cons_P(nixon)`
		`out_cons_not_P(nixon)`

5.1.2 Small Wine

Intuitive meaning In this example [Eiter et al., 2007b], we consider a small wine ontology L, which contains some knowledge about red and white wines, Lambrusco, as well as about Veuve Cliquot and Lambrusco di Modena; and we would like to say that:

- Normally, sparkling wines are white, and
- Normally, white wines are served cold.

Description Logic Knowledge Base

$$L = \left\{ \begin{array}{l} \mathit{RedWine} \sqsubseteq \neg \mathit{WhiteWine}, \mathit{Lambrusco} \sqsubseteq \mathit{SparklingWine} \sqcap \mathit{RedWine}, \\ \mathit{SparklingWine}(\mathit{veuveCliquot}), \quad \mathit{Lambrusco}(\mathit{lambrusco\_di\_Modena}) \end{array} \right\}$$

Notice that we can not conclude $\mathit{WhiteWine}(\mathit{veuveCliquot})$ from L alone. Adding the axiom $\mathit{SparklingWine} \sqsubseteq \mathit{WhiteWine}$ is impossible since it will make L inconsistent. However, the followings defaults specified on top of L can help.

Defaults

```
[SparklingWine(X);WhiteWine(X)]/[WhiteWine(X)]

[WhiteWine(X);ServedCold(X)]/[ServedCold(X)]
```

Expected result In this example, we would like to conclude that *veuveCliquot* is *WhiteWine* and hence it is *ServedCold*, both by default, as in Table 5.2.

5.2 Complex examples

5.2.1 Student

Intuitive meaning Through Sections 1 and 2, a student ontology has been mentioned in diagram (Example 1.2) and a description logic representation (Example 2.5). Due to its verbosity, we are not going to rewrite everything but refer the reader to previous examples. What we need to add now is the intuition of the defaults. There are some rational defaults which can be specified on top of this ontology:

Table 5.2: Expected results in the Nixon Diamond example for different transformations

Transformation	Expected results

Π	`all_in_def_1(veuveCliquot)`, `all_p_def_1(veuveCliquot)`
	`in_WhiteWine(veuveCliquot)`, `p_WhiteWine(veuveCliquot)`
	`all_in_def_2(veuveCliquot)`, `all_p_def_2(veuveCliquot)`
	`in_ServedCold(veuveCliquot)`, `p_ServedCold(veuveCliquot)`
	`out_def_1(lambrusco_di_Modena)`, `out_def_2(lambrusco_di_Modena)`

Ω	`all_in_def_1(veuveCliquot)`, `in_WhiteWine(veuveCliquot)`
	`all_in_def_2(veuveCliquot)`, `in_ServedCold(veuveCliquot)`

Υ	`all_in_def_1(veuveCliquot)`, `in_WhiteWine(veuveCliquot)`
	`all_in_def_2(veuveCliquot)`, `in_ServedCold(veuveCliquot)`
	`cons_WhiteWine(veuveCliquot)`, `cons_ServedCold(veuveCliquot)`
	`out_cons_WhiteWine(lambrusco_di_Modena)`
	`out_cons_ServedCold(lambrusco_di_Modena)`

- Normally, a graduate student is an assistant (to earn money for his/her study)
- Graduate students with scholarship(s) do not have to be assistants (unless he/she wants to earn some more money)
- Normally, an assistant should only be either a research assistant or a teaching assistant

Since the current version of dlvhex is not able to handle a large number of dl-atoms and individuals, we will test the example with the first two defaults only.

Defaults

```
[GraduateStudent(X);Assistant(X)]/[Assistant(X)]<arg1(X)>

[GraduateStudent(X) & hasScholarsip(X,S);-Assistant(X)]
    /[-Assistant(X)]<arg2(X,S)>
```

We can see the differences between this example and the previous ones, i.e., the use of typing predicates. Since we would like to avoid unnecessary guesses such as considering *hasScholarship* relationship between two *GraduateStudents*, supportive information can be provided via predicates `arg1` and `arg2` as follows:

Typing predicates

```
arg1(min).
arg1(tkren).

arg2(min, erasmus_mundus).
```

Expected result There are two extensions in this example, in which individual *tkren* is concluded to be an *Assistant* by default while individual *min* is derived to be an *Assistant* in one extension and not in the other, since it satisfies the preconditions of both defaults but the consequent of each default blocks the other as in the Nixon Diamond example, hence only one conclusion can be made at a time. Table 5.3 summarizes these results.

Table 5.3: Expected results in the Student example for different transformations

Transformation	**Expected results (two extensions)**	
Π	all_in_def_1(tkren)	all_in_def_1(tkren)
	in_Assistant(tkren)	in_Assistant(tkren)
	all_in_def_1(min)	all_in_def_2(min)
	in_Assistant(min)	in_not_Assistant(min)
	all_p_def_1(tkren)	all_p_def_1(tkren)
	p_Assistant(tkren)	p_Assistant(tkren)
	all_p_def_1(min)	all_p_def_2(min)
	p_Assistant(min)	p_not_Assistant(min)
	out_def_2(min)	out_def_1(min)
Ω	all_in_def_1(tkren)	all_in_def_1(tkren)
	in_Assistant(tkren)	in_Assistant(tkren)
	all_in_def_1(min)	all_in_def_2(min)
	in_Assistant(min)	in_not_Assistant(min)
Υ	all_in_def_1(tkren)	all_in_def_1(tkren)
	in_Assistant(tkren)	in_Assistant(tkren)
	all_in_def_1(min)	all_in_def_2(min)
	in_Assistant(min)	in_not_Assistant(min)
	cons_Assistant(tkren)	cons_Assistant(tkren)
	cons_Assistant(min)	cons_not_Assistant(min)
	out_cons_not_Assistant(tkren)	out_cons_not_Assistant(tkren)
	out_cons_not_Assistant(min)	out_cons_Assistant(min)

5.2.2 Web Services Property Reasoning

Intuitive meaning OWL-S[1] is well known to be an ontology for describing web services, including the profile, process and grounding of a web service. Handling all of these components of OWL-S is too heavy for the current combination of dlvhex, RacerPro and the dl-plugin. Therefore, we chose to experiment our example with profile and process and put a very simple default specifying that a web service is normally described by an atomic process, unless it is explicitly described as a composite one. Based on this example, we can put more defaults describing other different properties of a web service and allow reasoning about those properties in a non-monotonic way.

Description Logic Knowledge Base The documents, which encode OWL-S as OWL ontologies, are available on the Internet. However, different components are represented in different namespaces and OWL files; and the current dl-plugin does not support OWL import directives, which is used to partition an ontology to serveral files. Therefore, to test this example, we had to merge the content of three owl files just to have a partly definition of web services, namely *Service.owl*, *Profile.owl*, and *Process.owl* into one file;[2] but we still keep different namespaces to prevent name clash. For an example web service, we chose the BookFinderService.[3] Its contents also need to be combined into the web service definition above, but only the profile and process parts. Due to the merging, many unnecessary individuals appear in the ontology, hence typing predicates plays a very important role

[1] http://www.w3.org/Submission/OWL-S/

[2] files can be downloaded from http://www.ai.sri.com/daml/services/owl-s/1.2/

[3] http://www.mindswap.org/2004/owl-s/services.shtml

Table 5.4: Expected results in the Web Service example for different transformations

Transformation	**Expected results**

Π	`all_in_def_1("mindswap:BookFinderService")`
	`in_AtomicProcess("mindswap:BookFinderService")`
	`all_p_def_1("mindswap:BookFinderService")`
	`p_Assistant("mindswap:BookFinderService")`

Ω	`all_in_def_1("mindswap:BookFinderService")`
	`in_AtomicProcess("mindswap:BookFinderService")`

Υ	`all_in_def_1("mindswap:BookFinderService")`
	`in_AtomicProcess("mindswap:BookFinderService")`
	`cons_AtomicProcess("mindswap:BookFinderService")`

here.

Defaults Since we have to merge different sources into one ontology, we have to specify the namespaces:

```
#namespace("service",
           "http://www.daml.org/services/owl-s/1.1/Service.owl#")

#namespace("process",
           "http://www.daml.org/services/owl-s/1.1/Process.owl#")

#namespace("mindswap",
           "http://www.mindswap.org/2004/owl-s/1.1/BookFinder.owl#")

[service:Service(X);process:AtomicProcess(X)]
   /
[process:AtomicProcess(X)]<myService(X)>
```

Typing Predicates We consider only one individual here:

```
myService("mindswap:BookFinderService").
```

Expected Results Without information about what kind of process by which *BookFinder-Service* is described, we would like to conclude, by default, that it is described by an atomic process. The results according to different transformation are represented in Table 5.4.

5.3 Experimental Results

In this section, we will present our experimental results on the examples described in Section 4.3 and 5.1 and give a discussion. For each example, we tested the performance of each transformation based on the number of individuals concerned, denoted by n. For each test case, we compare two running modes: (i) caching query calls to RacerPro (`--dlopt=-push`), and (ii) not using caching (`--dlopt=-cache`). And we compare our three different transformations Π, Ω, and Υ. This means that we have six runs for each test case. To get objective experimental results, for each run, we restarted RacerPro and ran the test four times in a row to get the evaluation times. Among these running times, we

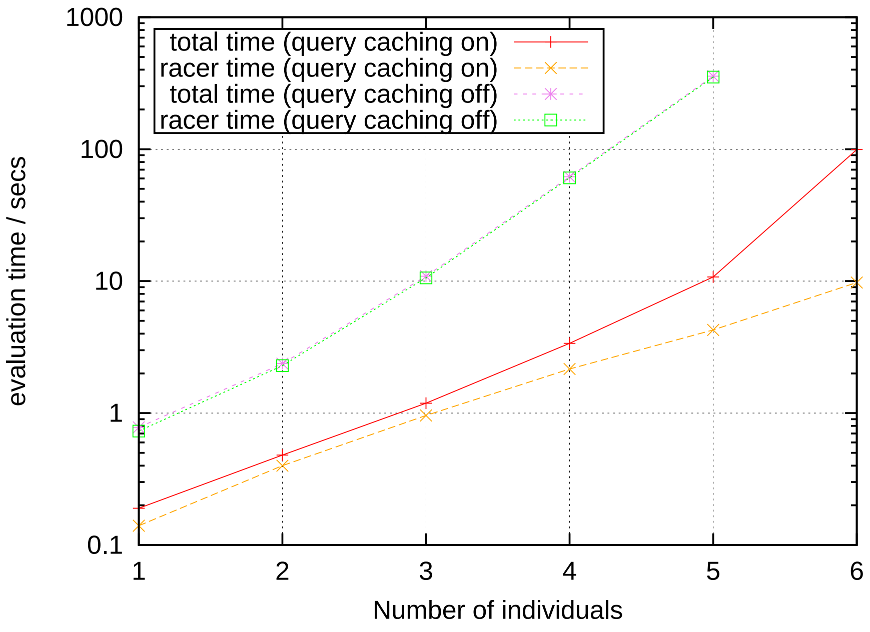


Figure 5.1: Tweety bird example's running time - transformation Π

omitted the fastest and slowest ones, took the other two and calculated the average as the final running time. We used time format `mm:ss:pp`, i.e., minutes, seconds and percentage of a second. For test cases that take too much time, (more than 30 minutes), we use "—" to represent the results. We are also interested the partial time that RacerPro and dlvhex use. The tests were done on a P4 1.8GHz PC with 1GB RAM under Ubuntu 6. We used RacerPro 1.9.2Beta, dlvhex and the dl-plugin source code updated on May 30th 2008.

5.3.1 Tweety bird example

First of all, we will have a look at the *"Tweety bird"* example. The results are shown in Table 5.5 and 5.6.

Figures 5.1 to 5.4 provides a graphical representation of the results. In all diagrams, the horizontal axis shows the number of individuals while the vertical axis serves to display the time in seconds. Missing entries in the diagrams indicate that the evaluation takes too much time to finish. Each figure from 5.1 to 5.3 compares total running times and RacerPro running time in two modes, namely `--dlopt=-push` and `--dlopt=-cache`, for each transformation from Π to Υ, respectively. Recall that `--dlopt=-push` is used for disabling push optimizations and `--dlopt=-cache` is used for disabling the DL-cache. Since in the dl-plugin, DL-cache has just been implemented for ordinary dl-atoms, `--dlopt=-push` has the same meaning as running under DL-Cache mode. Figure 5.4 compares three transformations in their fast mode, i.e., query caching is on.

Based on these tables and diagrams, we came up with the following analyses. The very first, and most obvious fact that we can see is that transformations Ω and Υ are much faster than Π. This is not surprising if we look closer at the transformations. While Ω tries to improve by not posing any guess for the consequents and Υ tries to minimize the number of dl-atoms to reduce the communication between dlvhex and RacerPro, Π has both of them, namely guessing rules for consequents and dl-atoms for every justifications. Moreover, it has two different input lists λ and λ', and uses two kinds of auxiliary predicates (*"in_"* and *"p_"*). Thus, Π not only has communicating overhead with RacerPro compared to Ω and

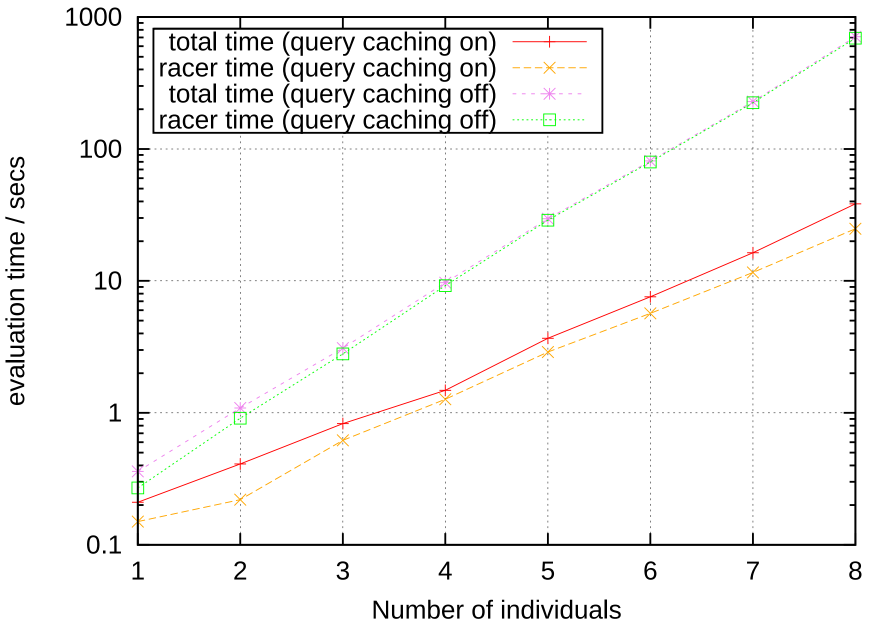


Figure 5.2: Tweety bird example's running time - transformation Ω

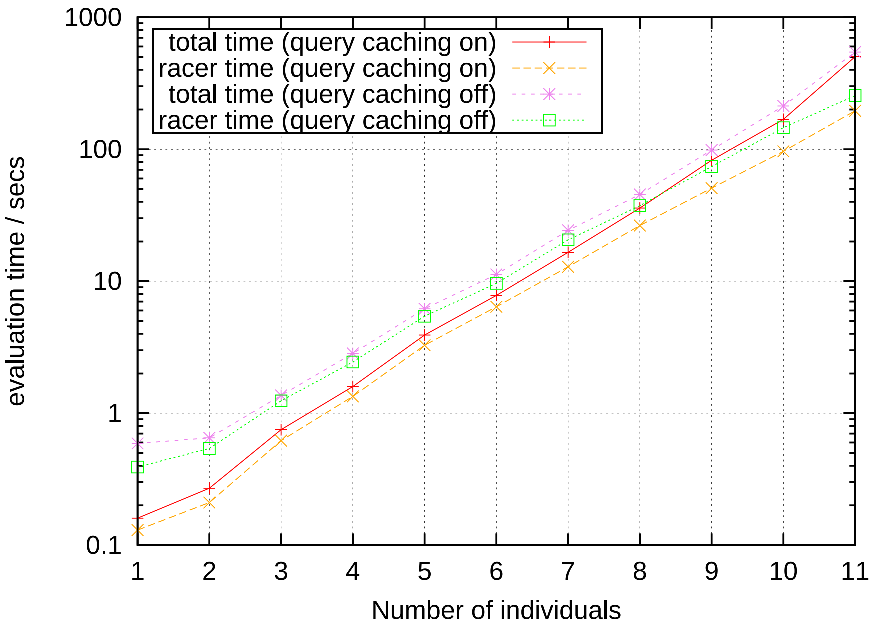


Figure 5.3: Tweety bird example's running time - transformation Υ

n	Transformation Π			Transformation Ω			Transformation Υ		
	total	racer	dlvhex	total	racer	dlvhex	total	racer	dlvhex
1	00:00:78	00:00:73	00:00:05	00:00:36	00:00:27	00:00:09	00:00:59	00:00:39	00:00:20
2	00:02:37	00:02:29	00:00:09	00:01:09	00:00:91	00:00:18	00:00:65	00:00:54	00:00:10
3	00:10:90	00:10:59	00:00:31	00:03:10	00:02:80	00:00:30	00:01:36	00:01:24	00:00:12
4	01:01:90	01:00:72	00:01:18	00:09:70	00:09:20	00:00:50	00:02:84	00:02:44	00:00:40
5	05:57:79	05:51:27	00:06:53	00:29:57	00:28:80	00:00:77	00:06:17	00:05:43	00:00:74
6	—	—	—	01:21:25	01:19:63	00:01:61	00:11:27	00:09:64	00:01:64
7	—	—	—	03:48:59	03:44:26	00:04:33	00:24:28	00:20:55	00:03:73
8	—	—	—	11:49:72	11:31:51	00:18:21	00:45:47	00:37:38	00:08:09
9	—	—	—	—	—	—	01:38:59	01:14:01	00:24:59
10	—	—	—	—	—	—	03:32:59	02:25:72	01:06:88
11	—	—	—	—	—	—	09:04:69	04:15:31	04:49:38

Table 5.5: Tweety bird example experiment results (query caching off)

n	Transformation Π			Transformation Ω			Transformation Υ		
	total	racer	dlvhex	total	racer	dlvhex	total	racer	dlvhex
1	00:00:19	00:00:14	00:00:05	00:00:21	00:00:15	00:00:06	00:00:16	00:00:13	00:00:03
2	00:00:48	00:00:40	00:00:09	00:00:41	00:00:22	00:00:19	00:00:27	00:00:21	00:00:06
3	00:01:19	00:00:96	00:00:23	00:00:83	00:00:62	00:00:21	00:00:75	00:00:62	00:00:13
4	00:03:38	00:02:16	00:01:22	00:01:48	00:01:27	00:00:21	00:01:59	00:01:34	00:00:26
5	00:10:76	00:04:27	00:06:48	00:03:67	00:02:89	00:00:79	00:03:91	00:03:27	00:00:64
6	01:39:12	00:09:75	01:29:37	00:07:58	00:05:67	00:01:92	00:07:79	00:06:41	00:01:38
7	—	—	—	00:16:36	00:11:58	00:04:78	00:16:55	00:12:84	00:03:71
8	—	—	—	00:38:39	00:24:83	00:13:57	00:35:83	00:26:44	00:09:38
9	—	—	—	—	—	—	01:22:46	00:50:75	00:31:71
10	—	—	—	—	—	—	02:48:36	01:36:29	01:12:08
11	—	—	—	—	—	—	08:23:02	03:15:35	05:07:66

Table 5.6: Tweety bird example experiment results (query caching on)

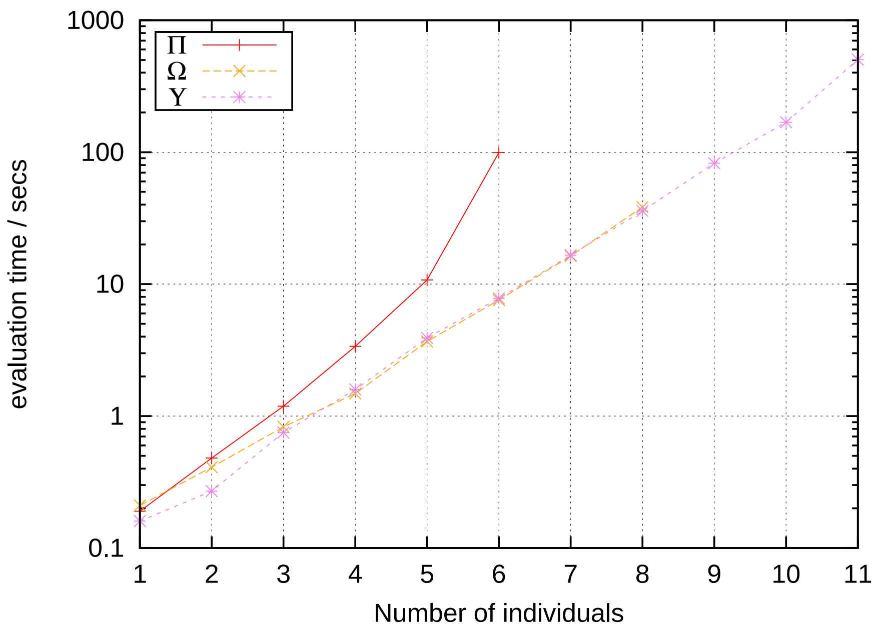


Figure 5.4: Tweety bird example - Comparing 3 transformations (query caching on)

Υ, but also has to deal with the exponentially increasing combinations when the number of input individuals increases. It cannot handle the cases in which we have more than 5 individuals and query caching is turned off. If query caching is turned on *Pi* can handle up to 6 individuals.

This shows our second expected result which is that using caching is much faster than not using caching.

Comparing Ω and Υ in this particular case, Υ has been able to handle the input size up to 11, while Ω spends a lot of time communicating with RacerPro if no caching is applied, because it has more dl-atoms. However, when caching is applied, Ω is able to save a lot of time and eventually it takes about the same amount of time as evaluating Υ, until giving up at input size $n \geq 9$.

Finally, in this particular example, we can see that the time consumed by RacerPro is longer than that used by dlvhex, except for the case in which $n = 11$. One of the reasons for this fact is that RacerPro currently has a bug when querying inconsistent updated KBs, that is, if we have an update to a KB that makes it inconsistent and query it, the first time RacerPro returns an inconsistent error message, which is correct. But if we do the update and query again, RacerPro returns a `NIL` result, which is incorrect. To circumvent this bug, one has to run `(full-reset)` before every query, which re-classifies the ontology. This is indeed a very expensive solution, especially in our case where querying the DL-KB is one of the main tasks. We believe that if RacerPro has this bug fixed, the performance of our transformations will improve significantly.

The case $n = 11$ seems not to follow the trends of the other previous ten test cases. However, it is explainable and is very close to what we can observe from the next two examples, namely *"Nixon Diamond"* and *"Small Wine."*

5.3.2 Nixon diamond and Small Wine

For the next two examples, we provide here the result in form of diagrams as shown in figure 5.5 and 5.6. However, since the size of the test cases are not large enough, the

	Transformation Π			Transformation Ω			Transformation ϒ		
n	total	racer	dlvhex	total	racer	dlvhex	total	racer	dlvhex
1	00:03:73	00:03:57	00:00:16	00:01:43	00:01:26	00:00:17	00:01:19	00:00:95	00:00:25
2	04:32:64	04:28:34	00:04:30	00:11:71	00:11:19	00:00:52	00:11:77	00:11:31	00:00:47
3	—	—	—	01:53:30	01:51:00	00:02:30	01:48:25	01:44:88	00:03:37
4	—	—	—	21:51:20	21:29:60	00:21:61	—	—	—

Table 5.7: Nixon Diamond example experiment results (query caching off)

	Transformation Π			Transformation Ω			Transformation ϒ		
n	total	racer	dlvhex	total	racer	dlvhex	total	racer	dlvhex
1	00:00:45	00:00:28	00:00:17	00:00:46	00:00:36	00:00:10	00:00:43	00:00:26	00:00:17
2	00:06:16	00:01:90	00:04:26	00:02:01	00:01:58	00:00:43	00:02:10	00:01:71	00:00:40
3	—	—	—	00:10:58	00:08:65	00:01:93	00:14:99	00:10:09	00:04:90
4	—	—	—	01:01:00	00:44:11	00:16:89	02:41:46	00:46:91	01:54:55

Table 5.8: Nixon diamond example experiment results (query caching on)

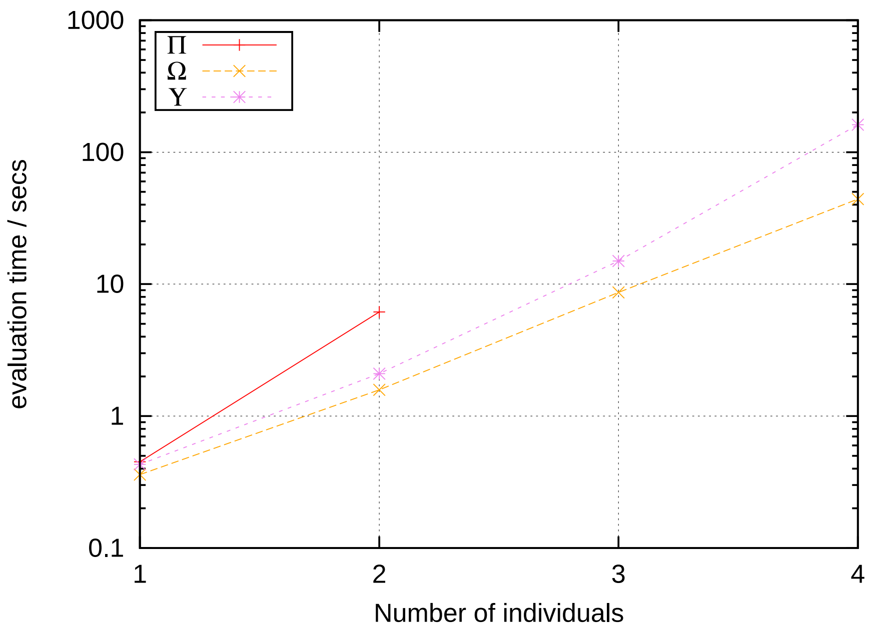


Figure 5.5: Nixon diamond example - Comparing 3 transformations (query caching on)

maximum number of individuals we can handle is 4 and 5 *"Nixon diamond"* and *"Small Wine"*, respectively. Therefore we will show only the comparisons between our three transformations, in their fast running mode when query caching is on.

We see that the performance of the system decreases drastically. The reason is that each of this examples has two defaults; one more default brings many more dl-atoms for Π and Ω, and more guesses on the justifications for ϒ; and the number of guesses in the ASP solver increases exponentially.

On the other side, the relationship between Π and both Ω and ϒ, whether using caching or not, are unchanged. The only difference compared to the result in the *"Tweety bird"* example is that Ω and ϒ share almost the same speed when query caching is off; and Ω is faster than ϒ when query caching is on. There is even a test case in *"Small wine"*, where $n = 5$, such that Ω finishes after nearly 6 minutes, while ϒ cannot terminate in a reasonable time frame. An explanation for this fact is that ϒ has to pay off with the guesses on the justifications. A careful comparison from columns *racer* and *dlvhex* for transformations Ω and ϒ in all tables reveals that the time spent in dlvhex increases by a much bigger factor in ϒ than compared to Ω. This is indeed the effect of exponential guesses increasing for the justifications in ϒ.

However, until now, we have not been able to confirm whether transformation Ω or ϒ is better in terms of performance. It depends on the performance of the DL-reasoner and the particular instance of the problem. A default theory with a large set of justifications is a big obstacle for ϒ while a slow DL-reasoner has a bigger effect on Ω than on ϒ.

The reader might wonder why there was no experiments regarding pruning rules described in Section 3.4 for, in particular, *"Nixon diamond"* example. Theoretically, adding constraints will reduce the search space, hence improve the performance of the transformation. However, in the current implementation of dlvhex, pruning rules are separated into a lower program component than our transformation. Each program component is evaluated consecutively, therefore pruning rules do not improve the overall performance of the system but make it a little bit slower, because we have to check the constraints for the

	Transformation Π			Transformation Ω			Transformation ϒ		
n	total	racer	dlvhex	total	racer	dlvhex	total	racer	dlvhex
1	00:06:75	00:06:45	00:00:30	00:01:29	00:01:20	00:00:08	00:01:14	00:01:06	00:00:08
2	20:58:57	20:37:30	00:21:27	00:11:96	00:11:73	00:00:23	00:07:73	00:07:30	00:00:43
3	—	—	—	01:53:43	01:51:29	00:02:14	01:43:64	01:40:84	00:02:80
4	—	—	—	—	—	—	—	—	—

Table 5.9: Small wine example experiment results (query caching off)

	Transformation Π			Transformation Ω			Transformation ϒ		
n	total	racer	dlvhex	total	racer	dlvhex	total	racer	dlvhex
1	00:00:57	00:00:33	00:00:24	00:00:43	00:00:38	00:00:05	00:00:42	00:00:32	00:00:10
2	00:12:35	00:02:33	00:10:02	00:01:61	00:01:30	00:00:31	00:01:76	00:01:32	00:00:44
3	—	—	—	00:10:65	00:08:59	00:02:06	00:10:15	00:06:80	00:03:35
4	—	—	—	01:01:98	00:44:42	00:17:56	02:15:98	00:36:93	01:39:05
5	—	—	—	05:51:96	03:30:62	02:21:34	—	—	—

Table 5.10: Small wine example experiment results (query caching on)

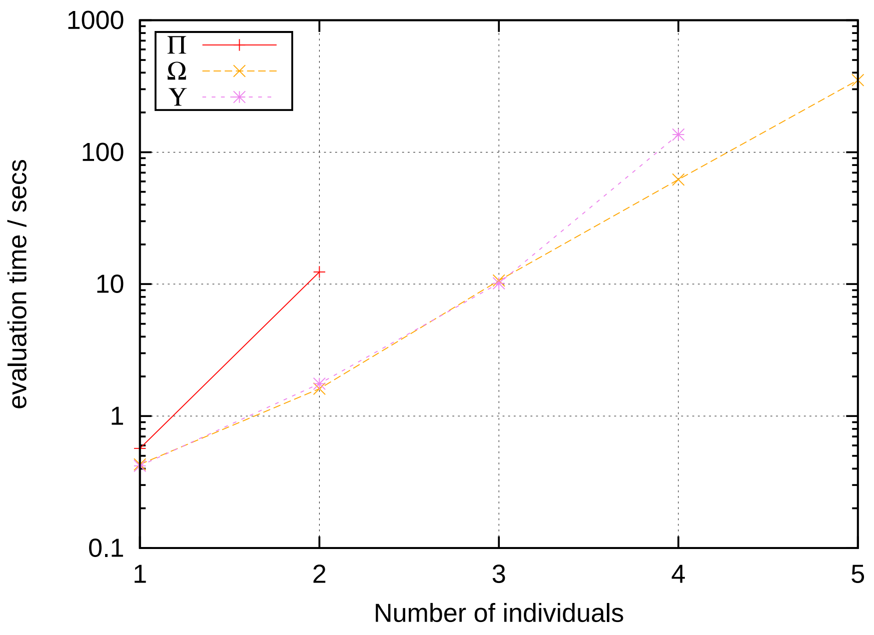


Figure 5.6: Small Wine example - Comparing 3 transformations (query caching on)

result of the transformation, which are satisfied anyway.

To prove the theoretical effect of pruning rules, we did a small hack into dlvhex by getting its intermediate rewritten rules, adding the pruning rules and evaluate this program under DLV. The result showed that fewer guesses were made, hence the running time should be faster. Nonetheless, the number of decreasing guesses in *"Nixon diamond"* example was not quite remarkable, as we had one guess interpretation pruned for each individual. Despite this fact, pruning rules are still promising to improve the performance. The issue here is that dlvhex needs to be upgraded so that constraints are shifted to an appropriate program component where they can fulfill their role to reduce the search space. This is one of the future works which will be mentioned again in Section 6.1.

5.3.3 Summary

To sum up, the experimental results showed that the two new transformations Ω and Υ are much faster compared to the original transformation Π from [Eiter et al., 2007b]. Another remarkable result is that caching techniques concerning calls to the ontology play an important role in improving the performance of the system while RacerPro is the biggest consumer of the total running time. Pruning rules, which have not been able to take effect, are promising to improve the system's performance. All of these results are valuable for the future work presented in Section 6.1.

6 Conclusion

In this work, we have studied different possibilities for transforming default theories over DL-KBs to dl-programs, respectively cq-programs, which are means of integrating logic programming and description-logic based ontologies. Based on these results, our approach for default reasoning over a DL-KB provides a simple and intuitive way for users to describe their defaults on top of an ontology without having to know about dl-programs or cq-programs, neither their syntax nor semantics. This is indeed a very convenient feature, especially for researchers working deeply in ontologies specified in an expert domain, for example medical or biological ontologies. Those researchers, on the one hand, have profound knowledge of the domain and have the need to reason with knowledge embedded in their ontologies, but on the other hand, they are not experts, or even not familiar with the notion of logic programming or answer-set semantics. Therefore, such a simple front-end hiding all these obstacles will bring logic programming, non-monotonic reasoning in general, and in particular dl-programs and cq-programs, closer to researchers in different fields so that they can take benefit from results which have been discovered by the logic community so far.

As we already mentioned, this is not the first attempt to combine default reasoning with description logics. The first one was proposed in [Baader and Hollunder, 1993]. However, from the theoretical point of view, our approach based on the strict semantic separation between rules and ontologies is guaranteed to be decidable as long as the DL-KB is decidable, while the approach in [Baader and Hollunder, 1993] is restricted to the description logics $\mathcal{ALC}$ and $\mathcal{ALCF}$. Moreover, from the practical point of view, we provide an implementation which was missing in [Baader and Hollunder, 1993]. The implementation is deployed as an additional component in the dl-plugin for the HEX-program solver dlvhex.

We have tested and compared three transformations from default theories to dl-programs, respectively cq-programs. Compared to transformation Π proposed in [Eiter et al., 2007b], the transformations Ω and Υ proposed here have significant performance gains. The experimental results revealed many future tasks that can help to improve the overall performance of our prototype implementation.

6.1 Future Work

Concerning our implementation, the following future work can be pointed out. Firstly, we would like to investigate more sophisticated pruning rules depending on the structure of the default theory. Secondly, a closer look into particular kinds of default theories such as normal default or semi-normal default theories should help to find more effective transformations.

Concerning dlvhex and the dl-plugin, in Section 5.3, we have seen the behaviour of our transformations in different running modes. The results suggested the following future tasks need for an increased system performance.

First of all, we have seen that the caching results from a DL-reasoner has a great impact on the evaluation time compared to not using caching. However, this technique is currently only available for ordinary dl-atoms. Thus, adding support for caching in cq-programs would give additional benefit.

We also witnessed that calling `(full-reset)` before every query to avoid a bug in RacerPro is very expensive for the overall speed of our system. Hence, it would be interesting to look at other possibilities interfacing dlvhex with different DL-reasoners such as KAON2 or Pellet, and then compare the results. This versatility will also be of interest for users, since they can freely choose between different DL-reasoners, especially between the commercial and the open-source ones.

After adding support for typing predicates, our system ran much faster since only a selected number of individuals are considered. Currently, the user has to specify the facts and rules for such typing predicates. To make it more user friendly, a challenging next step for dlvhex development would be to automatically classify the input, then do only necessary rules grounding to avoid unnecessary instances. Nonetheless, this is not an easy task and needs a lot of exploring deeply inside dlvhex.

It is well-known that deciding whether a propositional default theory has an extension is Σ_2^P-complete [Gottlob, 1992]. Therefore, we cannot expect a fast implementation in general. However, if users just need to check the consistency of their default theory on top of an ontology, we can provide an option that in which system halts whenever the first extension is found. Although this option does not solve the problem thoroughly, it still plays a role in practice.

As discussed in Section 5.3, an important step towards a faster implementation would be to upgrade dlvhex so that pruning rules can be effective. Specifically, what needs to be done is to shift a constraint to the upper program component whenever all the body atoms of the constraint appear in at least one head of a dl-rule of the program component.

A

Proofs

This section gives proofs for the correctness of the transformations Ω (Section 3.2) and Υ (Section 3.3). To ease the demonstration, we will consider defaults whose prerequisites, justifications and consequents consist only of a single literals. Formally speaking, let L be a DK-KB, and let $T = \langle W, D \rangle$ be a default theory, where W is the transformation of L to first-order logic, and let D consist of defaults of the form:

$$\delta = \frac{\alpha(\vec{X}) : \beta(\vec{Y})}{\gamma(\vec{Z})}$$

In each section below, we will first restate the dl-rules and then provide our claim followed by a proof for it.

A.1 Proof for Transformation Ω

The transformation Ω proposed in section 3.2 in this simple case is as follows. For each default $\delta \in D$, the transformation $\Omega(\delta)$ to dl-rules is:

$$aux\_\gamma(\vec{Z}) \leftarrow \mathrm{DL}[\lambda; \alpha](\vec{X}), \text{not } \mathrm{DL}[\lambda; \neg\beta](\vec{Y}).$$

The set of these dl-rules is called the program P.

Proof for theorem 3.4. We claim: **if there exists an answer set I_P of P then there exists a corresponding extension E_T of T and vice versa.** $(\star)$

We begin our proof for this property by showing how to construct E_T from I_P and vice verse:

(1) From I_P to E_T:

$E_T = W \cup \{\gamma(\vec{c}) | aux\_\gamma(\vec{c}) \in I_P\}$

(2) From E_T to I_P:

$I_P = \{aux\_\gamma(\vec{c}) | \gamma(\vec{c}) \in E_T\}$

In the rest of this section, P and T are clear of the context, hence we will omit subscripts P from I_P and T from E_T for simplicity.

The intuition of our transformation is that: the process of evaluating an extension in T will be simulated in the process of evaluating an answer set in P. Hence, every extension of T will have a corresponding answer set provided by P, which is equivalent to ($\star$). In the next steps, we show all the one-to-one correspondences between these two processes.

Evaluating interpretation I	Evaluating extension E
$in\_\gamma(\vec{c})$	$\gamma(\vec{c})$
$in\_not\_\gamma(\vec{c})$	$\neg\gamma(\vec{c})$
Description Logic KB L	Background theory W
A ground dl-rule r $aux\_\gamma(\vec{c}) \leftarrow$ $\mathrm{DL}[\lambda;\alpha](\vec{a}), \mathrm{not}\ \mathrm{DL}[\lambda;\neg\beta](\vec{b})$.	A ground default $\sigma = \frac{\alpha(\vec{a}):\beta(\vec{b})}{\gamma(\vec{c})}$
$L \cup \lambda(I) \not\models \neg\beta(\vec{b})$	$E \not\models \neg\beta(\vec{b})$ Notice that $W \subseteq E$ for every extension E
Evaluate the GL-reduct P_L^I Keep all rules r s.t. $L \cup \lambda(I) \not\models \neg\beta(\vec{b})$ Delete all NAF literals from these remaining rules.	Evaluate D_E $D_E = \left\{\frac{\alpha(\vec{a})}{\gamma(\vec{c})} \vert \frac{\alpha(\vec{a}):\beta(\vec{b})}{\gamma(\vec{c})} \in D \wedge E \not\models \neg\beta(\vec{b})\right\}$

To make the proof concise, we will use the symbol $\approx$ for the corresponding relation. Hereafter we provide an inductive proof of the following argument: "The least fix point (lfp) of P_L^I is corresponding to $\Gamma_T(E)$," denoted by:

$$\begin{array}{rcl} Th(T^{\omega}_{P_L^I}(\emptyset)) & \approx & Th^{D_E}(W) \\ \Leftrightarrow \quad Th(L \cup \bigcup_{k\geq 0} \lambda(I_k)) & \approx & Th(\bigcup_{k\geq 0} E_k), \end{array} \qquad (\star\star)$$

where ω is the ordinal for natural number, T_P is a continuous operator to compute the lfp of a logic program P, and

$$\begin{array}{rcl} I_0 & = & \emptyset; \\ I_{k+1} & = & T_{P_L^I}(I_{k-1}) = \{aux\_\gamma(\vec{c}) \vert r \in P_L^I \wedge L \cup \lambda(I_k) \models \alpha(\vec{a})\}; \\ E_0 & = & W; \\ E_{k+1} & = & \{\gamma(\vec{c}) \vert \frac{\alpha(\vec{a})}{\gamma(\vec{c})} \in D_E \wedge E_k \models \alpha(\vec{a})\}; \end{array}$$

Basic case: $I_0 = \emptyset$, $E_0 = W$. We have $Th(L) \approx Th(W)$ due to the fact that W is the transformation of L into fist order logic. ($\star\star$) holds with $w = 0$

Inductive case: assume that ($\star\star$) holds with k ($k \geq 0$), we will show that it also holds with $k+1$. Indeed:

Because ($\star\star$) holds with k, we have $[L \cup \lambda(I_k) \models \alpha(\vec{a})] \approx [E_k \models \alpha(\vec{a})]$. Moreover, we have $r \approx \sigma$, hence the conditions for constructing I_{k+1} and E_{k+1} are corresponding to each other, which yields that $Th(L \cup \bigcup_{i=0}^{k+1} I_i) \approx Th(\bigcup_{i=0}^{k+1} E_i)$. In other words, ($\star\star$) holds with $k+1$.

Finally, by the induction principle, we can conclude that ($\star\star$) holds for all $k \geq 0$.

From ($\star\star$) and all the correspondences above, we can conclude that ($\star$) holds.

$\square$

A.2 Proof for Transformation Υ

The transformation Υ proposed in Section 3.3 in this simple case is as follows. For each default $\delta \in D$, the transformation $\Upsilon(\delta)$ to dl-rules is the following set dl-rules:

$$\begin{array}{lrcl}
(r_1) & auxc\_\beta(\vec{Y}) & \leftarrow & \text{not } out\_auxc\_\beta(\vec{Y}). \\
(r_2) & out\_auxc\_\beta(\vec{Y}) & \leftarrow & \text{not } auxc\_\beta(\vec{Y}). \\
(r_3) & aux\_\gamma(\vec{Z}) & \leftarrow & \mathrm{DL}[\lambda;\alpha](\vec{X}), auxc\_\beta(\vec{Y}). \\
(r_4) & fail & \leftarrow & \mathrm{DL}[\lambda;\neg\beta](\vec{Y}), auxc\_\beta(\vec{Y}), \text{not } fail. \\
(r_5) & fail & \leftarrow & \text{not } \mathrm{DL}[\lambda;\neg\beta](\vec{Y}), out\_auxc\_\beta(\vec{Y}), \text{not } fail.
\end{array}$$

Proof for Theorem 3.5. Let P be the transformed dl-program from T. We claim: **if there exists a strong answer set I of P then there exists a corresponding extension E_T of T and vice versa.**

In the proof below, to make it simple, we omit subscripts T, P from E_T, I_P, respectively, for simplicity.

Firstly, we show how E is constructed from I and vice verse:

(1) From I to E:
$E = Cn(L \cup \lambda(I))$

(2) From E to I:

$$\begin{aligned} I \;=\; & \{auxc\_\beta(\vec{b}) | \neg\beta(\vec{b}) \notin E\} \cup \{out\_auxc\_\beta(\vec{b}) | \neg\beta(\vec{b}) \in E\} \\ & \cup \{aux\_\gamma(\vec{c}) | \tfrac{\alpha(\vec{a}):\beta(\vec{b})}{\gamma(\vec{c})} \in D \wedge \alpha(\vec{a}) \in E; \neg\beta(\vec{b}) \notin E\} \end{aligned}$$

We show: **if I is a strong answer set of P then E constructed by (1) is an extension of T.**
By (r_4): if $auxc\_\beta(\vec{b}) \in I$ then $\neg\beta(\vec{b}) \notin E$.

By (r_5): if $auxc\_\beta(\vec{b}) \notin I$, then $out\_auxc\_\beta(\vec{b}) \in I$, hence $\neg\beta(\vec{b}) \in E$.
Let $D^E = \{\frac{\alpha(\vec{a})}{\gamma(\vec{c})} | \frac{\alpha(\vec{a}):\beta(\vec{b})}{\gamma(\vec{c})} \in D \wedge auxc\_\beta \in I\}$

We claim: $Cn(L \cup \lambda(I))$ is closed under D^E. Indeed:

Consider $\frac{\alpha(\vec{a})}{\gamma(\vec{c})} \in D^E$.

Suppose that $\alpha(\vec{a}) \in Cn(L \cup \lambda(I))$. Then $\mathrm{DL}[\lambda;\alpha](\vec{a})$ is true in I. From the consistency condition, it is easy to see that $auxc\_\beta(\vec{b}) \in I$. Hence rule ($r_3$) fires, and $aux\_\gamma(\vec{c}) \in I$; therefore, $\gamma(\vec{c}) \in \lambda(I)$, as a result, $\gamma(\vec{c}) \in Cn(L \cup \lambda(I))$.
Since $E = Cn(L \cup \lambda(I))$ has the closeness property, we can conclude that $\Gamma_T(E) \subseteq E$, where $\Gamma_T(E)$ is as in Definition 2.3.
Now we will show that $E \subseteq \Gamma_T(E)$, in other words, there exists no E' such that $E' \subset E$ and E' is closed under D^E.

Suppose such an E' exists, then there exist some $\gamma(\vec{c})$ such that $\gamma(\vec{c}) \in E$ and $\gamma(\vec{c}) \notin E'$. We construct I' from E' as follows:

$$\begin{aligned} I' &= \{auxc\_\beta_i(\vec{b_i}) | \neg\beta_i(\vec{b_i}) \notin E\} \cup \{out\_auxc\_\beta_i(\vec{b_i}) | \neg\beta_i(\vec{b_i}) \in E\} \\ &\cup \{aux\_\gamma_i(\vec{c_i}) | \tfrac{\alpha_i(\vec{a_i}):\beta_i(\vec{b_i})}{\gamma_i(\vec{c_i})} \in D \wedge \alpha_i(\vec{a_i}) \in E'; \neg\beta_i(\vec{b_i}) \notin E\} \end{aligned}$$

This is similar to (2) above, but here we use E' instead of E.
Because E' is closed under D^E, there must exist some $\gamma(\vec{c})$ such that $\gamma(\vec{c}) \in I$ and $\gamma(\vec{c}) \notin I'$, in other words, $I' \subset I$.

Since $E' \subset E$, in P^I_L, the reduct of P w.r.t. I and L, we have:

- all rules (r_4) are satisfied by I';
- all rules (r_5) either disappear if $\beta(\vec{b})$ is not consistent with E or are satisfied by I' other wise;
- all rules (r_1) (r_2) and (r_3) are satisfied due to the construction of I'.

Therefore $I' \models P^I_L$, hence I is not a strong answer set, which is a contradiction.

So, our assumption of the existence of E' leads to a contradiction, therefore $E \subseteq \Gamma_T(E)$. Finally, we can conclude that $E = \Gamma_T(E)$, which means that E is an extension of T.

We show: **if E is an extension of T then I as constructed in (2) is a strong answer set of P.**
By the construction of I as in (2), it easy to see by the characterization of extension in terms of generating defaults, that in P^I_L:

- all rules (r_4) are satisfied by I,
- all rules (r_5) either disappear, if $\beta(\vec{b})$ is not consistent with E, or are satisfied by I other wise;
- all rules (r_1) and (r_2) are satisfied due to the construction of I
- for each rule (r_3), if the body of the rule is true, then the corresponding default δ must be applied in E; hence γ must be in E, and then $aux\_\gamma \in I$ by definition; hence the rule is satisfied.

Therefore, I is a model of P^I_L.

What we need to show now is that I is minimal. Suppose that there exists some $I' \subset I$ and $I' \models P^I_L$. Let $E' = Cn(L \cup \lambda(I'))$.

Since $I' \subset I$, there exist some A such that $A \in I$ and $A \notin I'$. Notice that in P^I_L, rule instances of (r_1) and (r_2) either disappear or become facts, hence I' and I must agree on the guessing of the consistency of the justifications to E. Hence, the different literal $A \in I'$ must be of the form $aux\_\gamma(\vec{c})$. This means the corresponding dl-rule does not fire, i.e., DL$[\lambda; \alpha](\vec{a})$ is evaluated to false, thus $aux\_\alpha_i(\vec{a}) \notin I'$. Therefore the corresponding monotonic rule $\frac{\alpha(\vec{a})}{\gamma(\vec{c})} \in D^E$ is satisfied in E'. In other words, E' is closed under D^E. Hence $\Gamma_T(E) \subseteq E'$.
Moreover, since $I' \subset I$, we have $E' \subset E$. Indeed, suppose $E' = E$. Then as I' is closed under the rule in P^I_L, we obtain that $I \subseteq I'$, which is a contradiction.

Hence $\Gamma_T(E) \subseteq E' \subset E$. This contradicts with the fact that E is an extension. Therefore, I is minimal and it is a strong answer set of P.

□

B

Transformed Programs

B.1 Tweety bird

B.1.1 Transformation Π

```
all_in_def_1(X) :- dom(X), not out_def_1(X).
out_def_1(X) :- dom(X), not all_in_def_1(X).
in_Flier(X) :- all_in_def_1(X).
fail :- out_def_1(X), DL[Flier+=in_Flier;Flier](X), not fail.
all_p_def_1(X) :- dom(X), DL[Flier+=in_Flier;Bird](X),
                  not DL[Flier+=in_Flier;-Flier](X).
p_Flier(X) :- all_p_def_1(X).
fail :- all_in_def_1(X), not DL[Flier+=p_Flier;Flier](X), not fail.
fail :- out_def_1(X), DL[Flier+=p_Flier;Flier](X), not fail.
```

B.1.2 Transformation Ω

```
all_in_def_1(X) :- dom(X), DL[Flier+=in_Flier;Bird](X),
                   not DL[Flier+=in_Flier;-Flier](X).
in_Flier(X) :- all_in_def_1(X).
```

B.1.3 Transformation Υ

```
cons_Flier(X) :- dom(X), not out_cons_Flier(X).
out_cons_Flier(X) :- dom(X), not cons_Flier(X).
all_in_def_1(X) :- cons_Flier(X), dom(X), DL[Flier+=in_Flier;Bird](X).
in_Flier(X) :- all_in_def_1(X).
fail :- cons_Flier(X), DL[Flier+=in_Flier;-Flier](X), not fail.
fail :- out_cons_Flier(X), not DL[Flier+=in_Flier;-Flier](X), not fail.
```

B.2 Nixon diamond

B.2.1 Transformation Π

```
all_in_def_2(X) :- dom(X), not out_def_2(X).
out_def_2(X) :- dom(X), not all_in_def_2(X).
```

```
in_P(X) :- all_in_def_2(X).
fail :- out_def_2(X), DL[P-=in_not_P, P+=in_P;P](X), not fail.
all_p_def_2(X) :- dom(X), DL[P-=in_not_P, P+=in_P;Q](X),
                  not DL[P-=in_not_P, P+=in_P;-P](X).
p_P(X) :- all_p_def_2(X).
fail :- all_in_def_2(X), not DL[P-=p_not_P, P+=p_P;P](X), not fail.
fail :- out_def_2(X), DL[P-=p_not_P, P+=p_P;P](X), not fail.
all_in_def_1(X) :- dom(X), not out_def_1(X).
out_def_1(X) :- dom(X), not all_in_def_1(X).
in_not_P(X) :- all_in_def_1(X).
fail :- out_def_1(X), DL[P-=in_not_P, P+=in_P;-P](X), not fail.
all_p_def_1(X) :- dom(X), DL[P-=in_not_P, P+=in_P;R](X),
                  not DL[P-=in_not_P, P+=in_P;P](X).
p_not_P(X) :- all_p_def_1(X).
fail :- all_in_def_1(X), not DL[P-=p_not_P, P+=p_P;-P](X), not fail.
fail :- out_def_1(X), DL[P-=p_not_P, P+=p_P;-P](X), not fail.
```

B.2.2 Transformation Ω

```
all_in_def_2(X) :- dom(X), DL[P-=in_not_P, P+=in_P;Q](X),
                   not DL[P-=in_not_P, P+=in_P;-P](X).
in_P(X) :- all_in_def_2(X).
all_in_def_1(X) :- dom(X), DL[P-=in_not_P, P+=in_P;R](X),
                   not DL[P-=in_not_P, P+=in_P;P](X).
in_not_P(X) :- all_in_def_1(X).
```

B.2.3 Transformation Υ

```
cons_P(X) :- dom(X), not out_cons_P(X).
out_cons_P(X) :- dom(X), not cons_P(X).
all_in_def_2(X) :- cons_P(X), dom(X), DL[P-=in_not_P, P+=in_P;Q](X).
in_P(X) :- all_in_def_2(X).
fail :- cons_P(X), DL[P-=in_not_P, P+=in_P;-P](X), not fail.
fail :- out_cons_P(X), not DL[P-=in_not_P, P+=in_P;-P](X), not fail.
cons_not_P(X) :- dom(X), not out_cons_not_P(X).
out_cons_not_P(X) :- dom(X), not cons_not_P(X).
all_in_def_1(X) :- cons_not_P(X), dom(X), DL[P-=in_not_P, P+=in_P;R](X).
in_not_P(X) :- all_in_def_1(X).
fail :- cons_not_P(X), DL[P-=in_not_P, P+=in_P;P](X), not fail.
fail :- out_cons_not_P(X), not DL[P-=in_not_P, P+=in_P;P](X), not fail.
```

B.3 Small Wine

B.3.1 Transformation Π

```
all_in_def_2(X) :- dom(X), not out_def_2(X).
out_def_2(X) :- dom(X), not all_in_def_2(X).
in_ServedCold(X) :- all_in_def_2(X).
fail :- out_def_2(X),
    DL[WhiteWine+=in_WhiteWine, ServedCold+=in_ServedCold;ServedCold](X),
    not fail.
all_p_def_2(X) :- dom(X),
    DL[WhiteWine+=in_WhiteWine, ServedCold+=in_ServedCold;WhiteWine](X),
    not DL[WhiteWine+=in_WhiteWine, ServedCold+=in_ServedCold;-ServedCold](X).
p_ServedCold(X) :- all_p_def_2(X).
fail :- all_in_def_2(X),
    not DL[WhiteWine+=p_WhiteWine, ServedCold+=p_ServedCold;ServedCold](X),
    not fail.
fail :- out_def_2(X),
    DL[WhiteWine+=p_WhiteWine, ServedCold+=p_ServedCold;ServedCold](X),
    not fail.
all_in_def_1(X) :- dom(X), not out_def_1(X).
out_def_1(X) :- dom(X), not all_in_def_1(X).
in_WhiteWine(X) :- all_in_def_1(X).
fail :- out_def_1(X),
    DL[WhiteWine+=in_WhiteWine, ServedCold+=in_ServedCold;WhiteWine](X),
    not fail.
all_p_def_1(X) :- dom(X),
    DL[WhiteWine+=in_WhiteWine, ServedCold+=in_ServedCold;SparklingWine](X),
    not DL[WhiteWine+=in_WhiteWine, ServedCold+=in_ServedCold;-WhiteWine](X).
p_WhiteWine(X) :- all_p_def_1(X).
fail :- all_in_def_1(X),
    not DL[WhiteWine+=p_WhiteWine, ServedCold+=p_ServedCold;WhiteWine](X),
    not fail.
fail :- out_def_1(X),
    DL[WhiteWine+=p_WhiteWine, ServedCold+=p_ServedCold;WhiteWine](X),
    not fail.
```

B.3.2 Transformation Ω

```
all_in_def_2(X) :- dom(X),
    DL[WhiteWine+=in_WhiteWine, ServedCold+=in_ServedCold;WhiteWine](X),
    not DL[WhiteWine+=in_WhiteWine, ServedCold+=in_ServedCold;-ServedCold](X).
in_ServedCold(X) :- all_in_def_2(X).
all_in_def_1(X) :- dom(X),
    DL[WhiteWine+=in_WhiteWine, ServedCold+=in_ServedCold;SparklingWine](X),
    not DL[WhiteWine+=in_WhiteWine, ServedCold+=in_ServedCold;-WhiteWine](X).
in_WhiteWine(X) :- all_in_def_1(X).
```

B.3.3 Transformation Υ

```
cons_ServedCold(X) :-
    dom(X), not out_cons_ServedCold(X).
out_cons_ServedCold(X) :-
    dom(X), not cons_ServedCold(X).
all_in_def_2(X) :-
     cons_ServedCold(X), dom(X),
    DL[WhiteWine+=in_WhiteWine, ServedCold+=in_ServedCold;WhiteWine](X).
in_ServedCold(X) :- all_in_def_2(X).
fail :- cons_ServedCold(X),
    DL[WhiteWine+=in_WhiteWine, ServedCold+=in_ServedCold;-ServedCold](X),
    not fail.
fail :- out_cons_ServedCold(X),
    not DL[WhiteWine+=in_WhiteWine, ServedCold+=in_ServedCold;-ServedCold](X),
    not fail.
cons_WhiteWine(X) :-
    dom(X), not out_cons_WhiteWine(X).
out_cons_WhiteWine(X) :-
    dom(X), not cons_WhiteWine(X).
all_in_def_1(X) :- cons_WhiteWine(X), dom(X),
    DL[WhiteWine+=in_WhiteWine, ServedCold+=in_ServedCold;SparklingWine](X).
in_WhiteWine(X) :- all_in_def_1(X).
fail :- cons_WhiteWine(X),
    DL[WhiteWine+=in_WhiteWine, ServedCold+=in_ServedCold;-WhiteWine](X),
    not fail.
fail :- out_cons_WhiteWine(X),
    not DL[WhiteWine+=in_WhiteWine, ServedCold+=in_ServedCold;-WhiteWine](X),
    not fail.
```

Bibliography

G. Antoniou. A tutorial on default logics. *ACM Comput. Surv.*, 31(4):337–359, 1999. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/344588.344602. 2.7

G. Antoniou, C. V. Damásio, B. Grosof, I. Horrocks, M. Kifer, J. Maluszynski, and P. 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. 1.4.2

F. Baader and P. Hanschke. A scheme for integrating concrete domains into concept languages. Technical Report RR-91-10, Deutsches Forschungszentrum für Künstliche Intelligenz, Germany (DFKI), 1991. URL `citeseer.ist.psu.edu/baader91scheme.html`. 2.7.2

F. Baader and B. Hollunder. Embedding defaults into terminological knowledge representation formalisms. Technical Report RR-93-20, Deutsches Forschungszentrum für Künstliche Intelligenz, Germany (DFKI), 1993. URL `citeseer.ist.psu.edu/baader95embedding.html`. 1, 2.7.2, 2.7.2, 6

F. Baader and C. Lutz. Description Logic. In P. Blackburn, J. van Benthem, and F. Wolter, editors, *The Handbook of Modal Logic*, pages 757–820. Elsevier, 2006. URL `http://lat.inf.tu-dresden.de/research/papers/2006/BaLu-ML-Handbook-06.ps.gz`. 2.3

F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications.* Cambridge University Press, 2003. 2.3

F. Baader, I. Horrocks, and U. Sattler. Description Logics. In F. van Harmelen, V. Lifschitz, and B. Porter, editors, *Handbook of Knowledge Representation.* Elsevier, 2007. URL `http://web.comlab.ox.ac.uk/oucl/work/ian.horrocks/Publications/download/2007/BaHS07a.pdf`. 2.3

P. Cholewinski and M. Truszczynski. Minimal number of permutations sufficient to compute all extensions a finite default theory. URL `citeseer.ist.psu.edu/250371.html`. unpublished note. 1, 2.7.2

K. L. Clark. Negation as failure. In *Logic and Data Bases*, pages 293–322, 1977. 1.1

J. de Bruijn, T. Eiter, A. Polleres, and H. Tompits. Embedding Non-Ground Logic Programs into Autoepistemic Logic for Knowledge-Base Combination. In *Proc. of the 20th Int. Joint Conf. on Art. Int. (IJCAI 2007)*, pages 304–309, Hyderabad, India, Jan. 2007a. Association for the Advancement of Artificial Intelligence (AAAI). URL `http://www.ijcai.org/papers07/Papers/IJCAI07-047.pdf`. 1.4.2

J. de Bruijn, D. Pearce, A. Polleres, and A. Valverde. Quantified Equilibrium Logic and Hybrid Rules. In *First Int. Conference on Web Reasoning and Rule Systems (RR2007)*, volume 4524 of *LNCS*, pages 58–72, Innsbruck, Austria, June 2007b. Springer. 1.4.2

J. Doyle. A truth maintenance system. *Artif. Intell.*, 12(3):231–272, 1979. 1.1

T. Eiter, T. Lukasiewicz, R. Schindlauer, and H. Tompits. Combining answer set programming with description logics for the Semantic Web. In *Proceedings KR-2004*, pages 141–151, 2004a. Extended Report RR-1843-03-13, Institut für Informationssysteme, TU Wien, 2003. 2.5, 2.5

T. Eiter, T. Lukasiewicz, R. Schindlauer, and H. Tompits. Well-founded Semantics for Description Logic Programs in the Semantic Web. In G. Antoniou and H. Boley, editors, *Proceedings RuleML 2004 Workshop, International Semantic Web Conference (ISWC), Hiroshima, Japan, November 2004*, number 3323 in LNCS, pages 81–97. Springer, 2004b. 2.5

T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits. A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In *International Joint Conferences on Artificial Intelligence (IJCAI)*, pages 90–96, 2005. 1.4.2

T. Eiter, G. Ianni, A. Polleres, R. Schindlauer, and H. Tompits. Reasoning with Rules and Ontologies. In P. Barahona, F. Bry, E. Franconi, N. Henze, and U. Sattler, editors, *Reasoning Web, Summer School 2006*, number 4126 in LNCS, pages 93–127. Springer, 2006. 1.4.2

T. Eiter, G. Ianni, T. Krennwallner, and R. Schindlauer. Exploiting Conjunctive Queries in Description Logic Programs. In D. Calvanese, E. Franconi, V. Haarslev, D. Lembo, B. Motik, A.-Y. Turhan, and S. Tessaris, editors, *Proceedings of the 20th International Workshop on Description Logics (DL2007)*, volume 250 of *CEUR Workshop Proceedings*, pages 259–266. CEUR-WS.org, June 2007a. URL `http://ceur-ws.org/Vol-250/paper_64.pdf`. 2.6, 2.8

T. Eiter, G. Ianni, T. Lukasiewicz, R. Schindlauer, and H. Tompits. Combining Answer Set Programming with Description Logics for the Semantic Web. Technical Report INFSYS RR-1843-07-04, Institut für Informationssysteme, TU Wien, Mar. 2007b. 1, 1, 1.4.2, 1.5, 2.3, 2.5, 3, 3.1, 3.2, 3.1, 5.1.2, 5.3.3, 6

T. Eiter, G. Ianni, T. Krennwallner, and R. Schindlauer. Exploiting Conjunctive Queries in Description Logic Programs. Technical Report INFSYS RR-1843-08-02, Institut für Informationssysteme, TU Wien, Favoritenstraße 9-11, A-1040 Vienna, Mar. 2008. URL `http://www.kr.tuwien.ac.at/research/reports/rr0802.pdf`. 1.4.2

M. Garey and D. Johnson. *Computers and Intractability. A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1979. 1.3

M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. A. Kowalski and K. Bowen, editors, *Proceedings of the Fifth International Conference on Logic Programming*, pages 1070–1080, Cambridge, Massachusetts, 1988. The MIT Press. URL `citeseer.ist.psu.edu/gelfond88stable.html`. 1.3, 2.1

M. R. Genesereth and N. J. Nilsson. *Logical Foundations of Artificial Intelligence.* San Mateo, CA: Morgan Kaufmann Publishers, 1987. 1.2

G. Gottlob. Complexity results for nonmonotonic logics. *J. Log. Comput.*, 2(3):397–425, 1992. 6.1

B. N. Grosof, I. Horrocks, R. Volz, and S. Decker. Description logic programs: Combining logic programs with description logic. In *Proc. of the Twelfth International World Wide Web Conference (WWW 2003)*, pages 48–57. ACM, 2003. ISBN 1-58113-680-3. URL `download/2003/p117-grosof.pdf`. 1.4.2

T. R. Gruber. Toward principles for the design of ontologies used for knowledge sharing. *Int. J. Hum.-Comput. Stud.*, 43(5-6):907–928, 1995. 1.2

I. Horrocks, U. Sattler, and S. Tobies. Practical Reasoning for Expressive Description Logics. In *Proceedings LPAR-1999*, volume 1705 of *LNCS*, pages 161–180. Springer, 1999. 2.3.1

I. Horrocks, P. F. Patel-Schneider, and F. 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. 2.4, 2.4

I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. SWRL: A semantic web rule language combining OWL and RuleML. W3C Member Submission, 21 May 2004. Available at `http://www.w3.org/Submission/SWRL/`. 1.4.2

U. Hufstadt, B. Motik, and U. Sattler. Reasoning for Description Logics around $\mathcal{SHIQ}$ in a Resolution Framework. Technical Report 3-8-04/04, Forschungszentrum Informatik (FZI), Karlsruhe, 76131 Karlsruhe, Germany, July 8, 2004. 1.4.2

U. Junker and K. Konolige. Computing the extensions of autoepistemic and default logics with a truth maintenance system. In *Association for the Advancement of Artificial Intelligence (AAAI)*, pages 278–283, 1990. 2.7.2

M. Knorr, J. J. Alferes, and P. Hitzler. A Well-founded Semantics for Hybrid MKNF Knowledge Bases. In *Proc. of 20th International Workshop on Description Logics DL'07*, volume 250 of *CEUR Workshop Proc.*, pages 347–354. CEUR-WS.org, 2007. URL `http://CEUR-WS.org/Vol-250/paper_54.pdf`. 2.5

T. Krennwallner. Integration of Conjunctive Queries over Description Logics into HEX-Programs. Master's thesis, Vienna University of Technology, Karlsplatz 13, A-1040 Wien, Oct. 2007. URL `http://www.postsubmeta.net/pub/2007/thesis.pdf`. 1.4, 2.6, 2.6, 4.4.1, 4.6

A. Y. Levy and M.-C. Rousset. CARIN: A representation language combining horn rules and description logics. In *European Conference on Artificial Intelligence*, pages 323–327, 1996. URL `citeseer.ist.psu.edu/levy96carin.html`. 1, 1.4.1, 1.4.2

T. Lukasiewicz. Stratified probabilistic description logic programs. In *International Semantic Web Conference-Uncertainty Reasoning for the Semantic Web (ISWC-URSW)*, pages 87–97, 2005. 1.4.2

W. Lukaszewicz. *Non-monotonic Reasoning, Formalization of Commonsense Reasoning.* Ellis Horwood, 1990. ISBN 0-13-624446-7. 1.1

J. McCarthy. Programs with common sense. In *Proceedings of the Teddington Conference on the Mechanization of Thought Processes*, pages 75–91, London, 1959. Her Majesty's Stationary Office. URL `citeseer.ist.psu.edu/mccarthy68programs.html`. 1.1

J. L. McCarthy. Epistemological problems of artificial intelligence. In *International Joint Conferences on Artificial Intelligence (IJCAI)*, pages 1038–1044, 1977. 1.1

D. V. McDermott and J. Doyle. Non-monotonic logic i. *Artif. Intell.*, 13(1-2):41–72, 1980. 1.1

J. Minker. An overview of nonmonotonic reasoning and logic programming. Technical Report UMIACS-TR-91-112, CS-TR-2736, University of Maryland, College Park, Maryland 20742, August 1991. URL `citeseer.ist.psu.edu/article/minker93overview.html`. 1.1

M. Minsky. A framework for representing knowledge. Technical Report AIM-306, 1974. 1.1

M. Minsky. Minsky's frame system theory. In *TINLAP '75: Proceedings of the 1975 workshop on Theoretical issues in natural language processing*, pages 104–116, Morristown, NJ, USA, 1975. Association for Computational Linguistics. doi: http://dx.doi.org/10.3115/980190.980222. 2.3

B. Motik and R. Rosati. A Faithful Integration of Description Logics with Logic Programming. In *Proc. of the 20th Int. Joint Conf. on Artificial Intelligence (IJCAI 2007)*, pages 477–482, Hyderabad, India, January 6–12 2007. Association for the Advancement of Artificial Intelligence (AAAI). URL `http://www.ijcai.org/papers07/Papers/IJCAI07-075.pdf`. 1.4.2

B. Motik, U. Sattler, and R. Studer. Query Answering for OWL-DL with Rules. *Journal of Web Semantics: Science, Services and Agents on the World Wide Web*, 3(1):41–60, 2005. 1.4.2

B. Motik, I. Horrocks, R. Rosati, and U. Sattler. Can OWL and Logic Programming Live Together Happily Ever After? In *Proceedings of the 2006 International Semantic Web Conference (ISWC 2006)*, volume 4273 of *Lecture Notes in Computer Science*, pages 501–514. Springer, 2006. URL `http://www.cs.man.ac.uk/~horrocks/Publications/download/2006/MHRS06.pdf`. 1, 1.4.2

R. Neches, R. Fikes, T. W. Finin, T. R. Gruber, R. S. Patil, T. E. Senator, and W. R. Swartout. Enabling technology for knowledge sharing. *AI Magazine*, 12(3):36–56, 1991. 1.2

J. Z. Pan, E. Franconi, S. Tessaris, G. Stamou, V. Tzouvaras, L. Serafini, I. R. Horrocks, and B. Glimm. Specification of Coordination of Rule and Ontology Languages. Project Deliverable D2.5.1, KnowledgeWeb NoE, June 2004. 1.4.2

D. Perlis. On the consistency of commonsense reasoning. *Computational Intelligence*, 2: 180–190, 1986. 1.1

R. Reiter. A logic for default reasoning. *Artif. Intell.*, 13(1-2):81–132, 1980. 1.1, 1.1, 2.7, 2.1, 2.2, 2.3, 2.5

R. Reiter. On Reasoning by Default. In *Proceedings of the 1978 workshop on Theoretical issues in natural language processing*, pages 210–218, Morristown, NJ, USA, 1978. Association for Computational Linguistics. URL `http://dx.doi.org/10.3115/980262.980297`. 1.1, 1.4.1

R. Rosati. On the decidability and complexity of integrating ontologies and rules. *Web Semantics*, 3(1):41–60, 2005a. ISSN 1570-8268. 1

R. Rosati. Semantic and Computational Advantages of the Safe Integration of Ontologies and Rules. In *Proceedings of the Third International Workshop on Principles and Practice of Semantic Web Reasoning (PPSWR 2005)*, volume 3703 of *Lecture Notes in Computer Science*, pages 50–64. Springer, 2005b. 1.4.2

R. Rosati. The limits of querying ontologies. In *Proceedings of the Eleventh International Conference on Database Theory (ICDT 2007)*, volume 4353 of *Lecture Notes in Computer Science*, pages 164–178. Springer, 2007. ISBN 3-540-69269-X. 3.2

R. Rosati. Towards expressive KR systems integrating datalog and description logics: preliminary report. In *Proceedings of the 1999 International Workshop on Description Logics (DL-1999)*, pages 160–164, 1999. 1.4.2

R. Rosati. On the Decidability and Complexity of Integrating Ontologies and Rules. *Journal of Web Semantics*, 3(1):61–73, 2005c. 1.4.2

R. Rosati. $\mathcal{DL}$+*log*: Tight Integration of Description Logics and Disjunctive Datalog. In *Proceedings of the Tenth International Conference on Principles of Knowledge Representation and Reasoning* (*KR 2006*), pages 68–78. Association for the Advancement of Artificial Intelligence (AAAI), 2006a. 1.4.2

R. Rosati. Integrating Ontologies and Rules: Semantic and Computational Issues. In *Reasoning Web, Summer School 2006*, number 4126 in LNCS, pages 128–151. Springer, 2006b. 1.4.2

R. Schindlauer. *Answer-Set Programming for the Semantic Web.* PhD thesis, Technische Universität Wien, 12 2006. 3, 2.7, 1, 2, 4.3

M. Sintek and S. Decker. Triple - a query, inference, and transformation language for the semantic web. In *ISWC '02: Proceedings of the First International Semantic Web Conference on The Semantic Web*, pages 364–378, London, UK, 2002. Springer-Verlag. ISBN 3-540-43760-6. 1.4.2

A. Van Gelder, K. A. Ross, and J. S. Schlipf. The Well-Founded Semantics for General Logic Programs. *Journal of the ACM*, 38(3):620–650, 1991. 2.5

T. Winograd. Extended inference modes in reasoning by computer systems. *Artif. Intell.*, 13(1-2):5–26, 1980. 1.1