D3.3 Complexity and Optimization of Combinations of Rules and Ontologies

Cristina Feier (TUWIEN)

with contributions from:

Hassan Aït-Kaci (IBM), Jürgen Angele (Ontoprise), Jos de Bruijn (FUB), Hugues Citeau (IBM), Thomas Eiter (TUWIEN), Adil El Ghali (IBM), Volha Kerhet (FUB), Eva Kiss (Ontoprise), Roman Korf (Ontoprise), Thomas Krekeler (Ontoprise), Thomas Krennwallner (TUWIEN), Stijn Heymans (TUWIEN), Alessandro Mosca (FUB), Martín Rezk (FUB), Guohui Xiao (TUWIEN)

Abstract.
Deliverable D3.3 Complexity and optimization of combinations of rules and ontologies. Will include the selection of promising combinations, and the complexity analysis of these selected combinations, as well as theoretical optimizations for processing.
Keyword list: combinations of rules and ontologies, Datalog rewritable DL, $\mathcal{LDL}^+$, Forest Logic Programs, Production Rules, fixpoint logics, OWL, XPR⟨OWL⟩

Work package number and name	WP3 Inference and Execution
Deliverable nature	Report
Dissemination level	Public/PU
Contractual date of delivery	M24
Actual date of delivery	January 27, 2011

ONTORULE Consortium

This document is part of a research project partially funded by the IST Programme of the Commission of the European Communities as project number 2009-231875 .

ILOG, an IBM Company
9, rue de Verdun
94253 Gentilly Cedex
France
Tel: +33 1 49 08 29 81
Fax: +33 1 49 08 35 10
Contact person: Mr. Christian de Sainte Marie
E-mail: csma@fr.ibm.com

Ontoprise GmbH
An der RaumFabrik 29
76227 Karlsruhe
Germany
Tel: +49 721 50980910
Fax: +49 721 50980911
Contact person: Mr. Jürgen Angele
E-mail: angele@ontoprise.de

Free University of Bozen-Bolzano
Faculty of Computer Science
Piazza Domenicani 3
I-39100 Bozen-Bolzano BZ, Italy
Italy
Tel: +39 0471 016 127
Fax: +39 0471 016 009
Contact person: Mr. Enrico Franconi
E-mail: franconi@inf.unibz.it

Technische Universität Wien
Institut für Informationssysteme
AB Wissensbasierte Systeme (184/3)
Favoritenstrasse 9-11
A-1040 Vienna
Austria
Tel: +43 1 58801 18460
Fax: +43 1 58801 18493
Contact person: Prof. Thomas Eiter
E-mail: eiter@kr.tuwien.ac.at

PNA Training B.V.
Geerstraat 105
6411 NP Heerlen
The Netherlands
Tel: +31 455600 222
Fax: +31 455600 062
Contact person: Prof. Sjir Nijssen
E-mail: sjir.nijssen@pna-training.nl

Université de Paris 13
LIPN
99, avenue J.B. Clément
F-93430 Villetaneuse
France
Tel: +33 1 4940 4089
Fax: +33 1 4826 0712
Contact person: Prof. Adeline Narazenko
E-mail: adeline.narazenko@lipn.univ-paris13.fr

Fundación CTIC
Edificio Centros tecnológicos
33203 cabueñes - Gijón
Asturias
Spain
Tel: +34 984 29 12 12
Fax: +34 984 39 06 12
Contact person: Mr. Antonio Campos
E-mail: antonio.campos@fundacionctic.org

Audi
Auto Union Strasse
D-85045 Ingolstadt
Germany
Tel: +49 841 89 39765
Contact person: Mr. Thomas Syldatke
E-mail: thomas.syldatke@audi.de

ArcelorMittal
Marques de Suances
33400 - Avilés
Spain
Tel: +34 98 5126 404
Fax: +34 98 5126 375
Contact person: Mr. Nicolas de Abajo
E-mail: nicolas.abajo@arcelormittal.com

Executive Summary

This document presents further developments on the theoretical and practical approaches to combining logical rules and ontologies, as well as on the approaches to combining production rules and ontologies, introduced in deliverable D3.2 [35]. The work is structured along two orthogonal axes: the first axis is about investigating combinations of logical rules and Description Logics vs. investigating combinations of production rules and Description Logics, while the second axis is about loosely-coupled vs. tightly-coupled approaches. By loosely-coupled approaches we understand approaches where the interaction between the rules and ontology is based on a loose interface between the components (using, for example, the entailment procedure for a Description Logic) whereas by tightly-coupled approaches we understand approaches which treat the ontology and the rule component as one whole and define models that take into account both the ontology axioms as well as the rules.

As the name of the deliverable suggests, the focus is on optimizations and establishing complexity results. In the area of integrating logical rules and ontologies, we took over the two most relevant approaches identified in deliverable D3.2 and investigated them further (see Chapter 2). These two approaches are the loosely-coupled approach *dl-programs* and the tightly-coupled approach *f-hybrid knowledge bases*. In the case of dl-programs, we define a class of so-called DATALOG-rewritable Description Logics and show how reasoning with dl-programs over such DLs under well-founded semantics can be reduced to DATALOG$^{\neg}$ by means of an efficient transformation. Thus, we obtained a tractable method to deal with such dl-programs. We also define constructively such a logic, $\mathcal{LDL}^{+}$, and provide an implementation for reasoning with dl-programs over $\mathcal{LDL}^{+}$ ontologies, which uses the previously mentioned transformation.

f-hybrid knowledge bases achieve a tight combination of Forest Logic Progams (FoLPs) and $\mathcal{SHOQ}$ ontologies, by translation of the $\mathcal{SHOQ}$ ontologies to FoLPs. As such, we devised an algorithm which employs a knowledge pre-compilation technique for reasoning with the "unifying" formalism Forest Logic Programs: the algorithm has the same worst-case running time as the original algorithm, but based on empirical results concerning the usage of such techniques for other non-tractable formalisms we have reasons to believe that it would bring a great improvement when reasoning with such programs. It can also be seen as a starting point for heuristic reasoning with FoLPs/f-hybrid knowledge bases.

The work done by ontoprise in the period M13-M24 is closely related with the theoretical work on DATALOG-rewritable DLs: ontoprise has implemented the OWL2 RL profile (see chapter 3). The OWL 2 RL profile defines a syntactic subset of OWL 2 which is aimed at applications that require scalable reasoning without sacrificing too much expressive power. The RL acronym stands for the fact that reasoning in this profile can be implemented using a standard Rule Language. Ontoprise has used for this implementation the ontoprise product suite, specifically OntoBroker 6.1 with ObjectLogic as ontology and rule language. The axiomatization of OWL 2 RL in ObjectLogic is presented in this deliverable, while deliverable D3.6 *Efficient processing of expressive combinations* contains a demonstrator and a tutorial to illustrate the usage of the OWL in ObjectLogic constructs and the reasoning with the RL profile.

In deliverable D3.2 first steps for embedding production systems in logics have been made: in particular, it has been shown how to embed propositional and first-order production systems in μ-calculus and fixpoint logic, respectively. In this deliverable we took that work further by formalizing combinations of production rules and ontologies (see chapter 4). Both a loosely-coupled approach, and a tightly-coupled approach are considered. The loosely-coupled approach is based on first-order logic (both conditions and actions are defined using capabilities of FOL), with a special regard for DL ontologies. The tightly-coupled approach extends the FPL embedding presented in deliverable D3.2 to cover the semantics of the combination.

Chapter 5 describes the issues encountered by IBM when implementing XPR⟨OWL⟩, an execution engine for production rules over ontologies. The implemented approach is a loosely-coupling one that consists of implementing a new PR engine that delegates all ontological processing to an OWL engine. The prototype implementation, based on Jena [20] (see Deliverable D3.6 [40] for the details of the implementation), was driven by two aims: (a) bring the theoretical framework and the implementation in line as much as possible, and (b) make sure that the implementation respects the OWL semantics, while preserving the peculiarities of the PR engine.

Table of Contents

Chapter 1

Introduction

In this deliverable we present the progress on the theoretical work going on in the Ontorule project in the last year concerning combinations of logical/production rules and ontologies, as well as theoretical aspects concerning the implementations of such combinations. The first two chapters of the deliverable deal with combinations of logical rules and ontologies: Chapter 2, for the theoretical part, and Chapter 3, for the practical part. The next two chapters deal with combinations of production rules and ontologies: Chapter 4, for the theoretical part, and Chapter 5, for the practical part.

In Chapter 2, we selected two of the approaches introduced in deliverable D3.2 [35] and developed them further. The first approach is the loosely-coupled formalism of dl-programs under well-founded semantics (see Section 2.2). dl-programs support a loosely-coupled integration of rules and ontologies, and provide an expressive combination framework based on the interaction of rules with a DL knowledge base (KB) via so-called *dl-atoms*. In order to achieve tractability when reasoning with dl-programs, a class of tractable Description Logics has been identified: the so-called DATALOG-rewritable DLs. It has also been shown how reasoning with dl-programs over such DLs under well-founded semantics can be reduced to DATALOG$^{\neg}$ by means of an efficient transformation. This class is defined via a semantic property: there has to exist a certain transformation that translates DL KBs to DATALOG programs, such that ground entailment from a DL KB carries over to calculating the unique minimal model of the DATALOG program. Besides this non-constructive class, we also present a (syntactically defined) DL which has this property: $\mathcal{LDL}^{+}$. $\mathcal{LDL}^{+}$has no negation (hence the $+$) and distinguishes between expressions on the left- and right-hand side of axioms. $\mathcal{LDL}^{+}$ offers expressive concept-and role expressions on the left-hand side of axioms (hence the $\mathcal{L}$ in $\mathcal{LDL}^{+}$), e.g., qualified number restrictions and transitive closure of roles. Finally, a reasoner for dl-programs, DReW, has been developed.

The second approach is the tightly-coupled formalism of f-hybrid knowledge bases. F-hybrid knowledge bases are combinations of $\mathcal{SHOQ}$ ontologies and Forest Logic Programs. The latter are a decidable subset of Open Answer Set Programming which makes

use only of unary and binary predicates and has the *forest-model property*: if a unary predicate is satisfiable then there is a model that can be seen as a labeled forest, every node of the forest being an element of the domain, and every predicate p in the label of a node x suggesting the presence of $p(x)$ in the model. Satisfiability checking of unary predicates/concepts in f-hybrid knowledge bases is reduced to satisfiability checking of unary predicates w.r.t. FoLPs by a translation of $\mathcal{SHOQ}$ ontologies to FoLPs.

In Section 2.3 we use the tableau algorithm for reasoning with FoLPs described in deliverable D3.2 [35] for computing tree-shaped structures of depth 1 which can be possible building blocks of any forest model for a particular FoLP. We call this building blocks *unit completion structures*. A new tableau algorithm is described which tries to build up a (forest) model by simply matching and appending such unit completion structures. Some structures are strict "supersets" of others, making it harder to be matched with other structures. We identify these redundant structures and discard them. The new algorithm runs as the original one in the worst case in double exponential time. The high complexity is determined by the depth of the explored forests. However it can serve as the basis for an heuristic-based implementation: in practice, it is highly improbable that a unary predicate is satisfied only by a forest model of considerable size. As such, we can set a limit on the depth of the explored forest: if no model is found within that limit, one can conclude with a high probability that the predicate is unsatisfiable.

Chapter 3 describes the ontoprise contribution to the OntoRule Deliverable 3.3. This consists in providing an implementation of OWL 2 RL in ObjectLogic. ObjectLogic rules are conjunctions of literals in the rule head and arbitrary predicate logic formulas in the rule body. In OntoBroker the rules are transformed in an extended form of datalog programs using the Lloyd-Topor [42] transformation and evaluated during query time. Functions and negations are supported as well. OntoBroker supports multiple query languages. The primary query language and the native format of OntoBroker is ObjectLogic. Other supported languages are a subset of SPARQL and SQL. Disjunctive queries or queries which contain builtins, temporary facts, etc. are only supported by ObjectLogic.

The next chapter of the deliverable, chapter 4 moves to the area of combining production rules and ontologies. While deliverable D3.2 [35] presented an embedding of propositional and FOL based production rules in μ-calculus and fixpoint logic, this chapter takes this work further by presenting different embeddings of production rules and ontologies. Section 4.1 describes a *loosely-coupled* approach, where the semantics of the production rules and the ontologies are decoupled: the interaction between the two semantics is based on entailment, while Section 4.2 describes a *tightly-coupled* approach based on fixpoint logic. A new semantics for production systems augmented with DL ontologies is provided and several issues that arise when combining production rules (PR) with description logics (DL) ontologies are addressed.

Based on the theoretical framework for satisfaction of conditions and execution of actions introduced in [35], some issues resulting from the combination of PRs and ontologies have been identified and studied. The deep analysis of these issues allowed us to enhance the

implementation of XPR⟨OWL⟩ an execution engine for production rule over ontologies, and gave us a better comprehension of the impact of replacing object model with OWL in a PR engine, as well as the limitations of the loosely-coupled approach. Chapter 5 describes these issues encountered by IBM when implementing XPR⟨OWL⟩.

Finally, Appendix B reports on the analysis of the use cases performed in the context of Ontorule Task 3.2. Task 3.2 involves monitoring of the issues arising in combinations of rules and ontologies in the case studies. Additionally, the survey will be updated if necessary in subsequent deliverables in WP3.

Chapter 2

Theoretical Foundations for Complexity and Optimizations for Combining Logic Programs and Ontologies

2.1 Introduction

In this chapter, we present developments on two approaches for combining rules and ontologies introduced in Deliverable D3.2: *dl-programs* and *f-hybrid knowledge bases*.

dl-programs are a loosely-coupled formalism for integrating Logic Programming rules and DL ontologies: the main mechanism of interaction between the two components is a special type of atoms which appear in the LP rules called *dl-atoms*. Such dl-atoms query the DL KB by checking for entailment of ground atoms or axioms w.r.t. the KB; as knowledge deduced by the rules can be streamed up to the DL KB in turn, a bi-directional flow of information is possible. The *answer set semantics* of dl-programs in [15], based on [24], is highly expressive, but on the other hand already intractable on the rule side; hence, towards scalable reasoning with negation, D3.2 presented a *well-founded semantics* for dl-programs, based on [23]. Given that the queries in dl-atoms are tractable, such programs can be evaluated in polynomial time (as usual, under data complexity).

Tractability of queries in dl-atoms is in line with recent tractable DLs such as the $DL\text{-}Lite$ families [11], $\mathcal{EL}^{++}$ [5, 6], and *Description Logic Programs* [27], that strive for scalability. In fact, they gave rise to three families of languages that resulted in the OWL 2 Profiles of the emerging Web Ontology Language OWL 2 [44]. However, even when loosely coupling such a tractable DL with rules via dl-programs under well-founded semantics, one still needs a dedicated algorithm that uses native DL reasoners to perform the external queries, thus causing a significant overhead. We overcome this, by identifying a class of Description Logics, so-called DATALOG-*rewritable DLs*, for which reasoning with dl-programs can be reduced to pure Logic Programming, i.e., to DATALOG$^{\neg}$ (DATALOG with

negation under well-founded semantics). This class is defined via a semantic property: there has to be a certain transformation that translates DL KBs to DATALOG programs, such that ground entailment from a DL KB can be reduced to calculating the unique minimal model of the DATALOG program. Besides this non-constructive class, we also present a (syntactically defined) DL which has this property: the novel DL $\mathcal{LDL}^+$.

We also developed a reasoner, DReW, which uses the DATALOG-rewriting technique. DReW can answer conjunctive queries over $\mathcal{LDL}^+$ ontologies, as well as reason on dl-programs over $\mathcal{LDL}^+$ ontologies under well-founded semantics. The preliminary but encouraging experimental results show that DReW can efficiently handle large knowledge bases.

One of the missing features in $\mathcal{LDL}^+$ is the *exists restriction* on axiom right-hand sides. However, DLs allowing this are not straight DATALOG-rewritable, as this feature can enforce the introduction of new domain elements (beyond the Herbrand domain). One may handle this using function symbols in the logic program or so-called *open domains* [34]. The latter is the principle behind Open Answer Set Programming (OASP) . In deliverable D3.2 we provided a tableau-based algorithm for satisfiability checking w.r.t. forest logic programs (FoLP), a decidable fragment of OASP, which has the forest model property. The fragment underpins a tightly-coupled approach for combining rules and ontologies, the so-called *f-hybrid knowledge bases*: an f-hybrid knowledge base consists in a $\mathcal{SHOQ}$ knowledge base together with a colection of FoLP rules. Reasoning with respect to f-hybrid knowledge bases can be reduced to reasoning with FoLPs as by translation of $\mathcal{SHOQ}$ KBs to FoLPs.

In this chapter we describe a new algorithm for reasoning with FoLPs which is based on a knowledge pre-compilation technique. So-called unit completion structures, which are possible building blocks of a forest model, in the form of trees of depth 1, are computed in an initial step of the algorithm. This is done by using the original algorithm. Repeated computations are avoided by using these structures in a pattern-matching style when constructing a model. From the pool of unit completion structures we also identify structures which can be seen as redundant and discard them. The new algorithm does not improve on the worst-case running time of the algorithm, as this is determined by the depth of the forests we have to explore. However as discussed in section 2.4.2, we expect it will perform considerably better than the original algorithm in returning positive answers to satisfiability checking queries. This opens the way for using heuristics like establishing a limit on the depth of the explored structures: in practice it is highly improbable that if there is a solution, it can be found only in an open answer set of a considerable size (depth of the corresponding extended forest).

2.2 Tractable Reasoning with DL-Programs over Datalog-rewritable Description Logics

2.2.1 Introduction

In this section we define a class of DATALOG-rewritable DLs (Section 2.2.3), and show how reasoning with dl-programs over such DLs under well-founded semantics can be reduced to DATALOG$^\neg$ by means of an efficient transformation. Noticeably, for dl-programs without negation, the result is a standard DATALOG program; moreover, the transformation preserves stratified negation.

$\mathcal{LDL}^+$ is defined as a particular DATALOG-rewritable DL (Section 2.2.4). This DL has no negation (hence the $+$) and distinguishes between expressions on the left- and right-hand side of axioms. $\mathcal{LDL}^+$ offers expressive concept-and role expressions on the left-hand side of axioms (hence the $\mathcal{L}$ in $\mathcal{LDL}^+$), e.g., qualified number restrictions and transitive closure of roles. The DATALOG-rewritability of $\mathcal{LDL}^+$ (Section 2.2.5) is interesting in itself, showing how to do reasoning in DLs with expressive constructs efficiently via Logic Programming. As a side result, we obtain that reasoning in $\mathcal{LDL}^+$ is tractable, considering both data and combined complexity; more precisely, we show that it is PTIME-complete in both settings. Despite its low complexity, $\mathcal{LDL}^+$ is still expressive enough to represent many constructs useful in ontology applications [6] such as role equivalences and transitive roles.

In Section 2.2.6 we review the different OWL 2 Profiles and relate them to $\mathcal{LDL}^+$. While $\mathcal{LDL}^+$ misses some constructs, e.g., the *exists restriction* on axiom right-hand sides as in $\mathcal{EL}^{++}$ and $DL\text{-}Lite$, or negation as in the $DL\text{-}Lite$ families, it adds others, e.g., expressive role constructs and *transitive closure* (which is not expressible in first-order logic). Furthermore, we show that $\mathcal{LDL}^+$ encompasses Description Logic Programs without a complexity increase.

Section 2.2.7 describes our new reasoner DReW (DATALOG ReWriter)[1], which rewrites $\mathcal{LDL}^+$ ontologies (dl-programs over $\mathcal{LDL}^+$ ontologies) to DATALOG (DATALOG$^\neg$) programs, and calls an underlying rule-based reasoner, currently DLV, to perform the actual reasoning. For $\mathcal{LDL}^+$ ontologies, DReW does instance checking as well as answering of conjunctive queries (CQs). For dl-programs over $\mathcal{LDL}^+$ ontologies, DReW computes the well-founded model.

DReW has been evaluated along two axes: as a pure DL reasoner and as a reasoner for dl-programs. Several real-word ontologies fall to a large extent in the $\mathcal{LDL}^+$ fragment. This enables us to compare CQs over the LUBM [28] benchmark with Pellet, KAON2 and RacerPro. For dl-programs, we compare DReW with DLVHEX over LUBM ontologies with dl-rules. The preliminary but encouraging experimental results show that DReW can efficiently handle large knowledge bases.

[1] `http://www.kr.tuwien.ac.at/research/systems/drew`

2.2.2 Preliminaries

2.2.2.1 DATALOG and DATALOG$^\neg$

Constants, variables, terms, and atoms are defined as usual. We assume that a binary inequality predicate $\neq$ is available; atoms not using $\neq$ are *normal*. A DATALOG$^\neg$ rule r has the form

$$h \leftarrow b_1, \ldots, b_k, not\ c_1, \ldots, not\ c_m \tag{2.1}$$

where the *body* $b_1, \ldots, b_k, c_1, \ldots, c_m$ are atoms and h is a normal atom.

We call $H(r) = \{h\}$ the *head*, $B^+(r) = \{b_1, \ldots, b_k\}$ the positive body, $B^-(r) = \{c_1, \ldots, c_m\}$ the *negative body*, and then $B(r) = \{b_1, \ldots, b_k, not\ c_1, \ldots, not\ c_m\} = B^+(r) \cup\ not\ B^-(r)$ the body. If $B^-(r) = \emptyset$, then r is a DATALOG rule. A finite set of DATALOG$^\neg$ (DATALOG) rules is a DATALOG$^\neg$ (DATALOG) *program*. *Ground* terms, atoms, and programs are defined as usual. A *fact* is a rule (2.1) with $k = m = 0$.

The *Herbrand Domain* $\mathcal{H}_P$ of a program P is the set of constants from P. The *Herbrand Base* $\mathcal{B}_P$ of P is the set of normal ground atoms with predicates and constants from P. An *interpretation* of P is any set $I \subseteq \mathcal{B}_P$. For a ground normal atom a, we write $I \models a$ if $a \in I$; for a ground atom $c_1 \neq c_2$, we write $I \models c_1 \neq c_2$ if c_1 and c_2 are different; for a ground *negation as failure atom* $l = not\ a$, we write $I \models l$ if $I \not\models a$. For a set of ground (negation as failure) atoms α, $I \models \alpha$ if $I \models l$ for all $l \in \alpha$. A ground rule $r : h \leftarrow \alpha$ is *satisfied* w.r.t. I, denoted $I \models r$, if $I \models h$ whenever $I \models \alpha$.

An interpretation I of a ground program P is a *model* of P, if $I \models r$ for every $r \in P$; in addition, I is *minimal*, if P has no model $J \subset I$. For a non-ground P, I is a (minimal) model of P iff it is a (minimal) model of $gr(P)$, the *grounding* of P with the constants of P defined as usual. Each DATALOG program P has some minimal model, which in fact is unique; we denote it with $MM(P)$. We write $P \models a$ if $MM(P) \models a$.

We recall the *well-founded semantics* [23] for DATALOG$^\neg$. Let I be an interpretation for a DATALOG$^\neg$ program P. The *GL-reduct* [24] P^I of a program P is the set of DATALOG rules $h \leftarrow b_1, \ldots, b_k$ such that $r : h \leftarrow b_1, \ldots, b_k, not\ c_1, \ldots, not\ c_m \in gr(P)$ and $I \not\models c_i$, for all $i, 1 \leqslant i \leqslant m$.

Using the γ operator [10], one can define the well-founded semantics as follows. Let $\gamma_P(I) = MM(P^I)$ and $\gamma_P^2(I) = \gamma_P(\gamma_P(I))$, i.e., applying the γ operator twice; as γ_P is anti-monotone, γ_P^2 is monotone. The set of *well-founded* atoms of P, denoted $WFS(P)$, is exactly the least fixed point of γ_P^2. We denote with $P \models^{wf} a$ that $a \in WFS(P)$.

For a DATALOG (DATALOG$^\neg$) program P and an atom a, deciding $P \models a$ ($P \models^{wf} a$) is *data complete* (P is fixed except for facts) for PTIME and *(combined) complete* (P is arbitrary) for EXPTIME [12].

Example 2.2.1. *Let P be the program with rules*

$$
\begin{aligned}
r_1 :&\quad good(X) \leftarrow super(X,Y), not\ over(Y)\\
r_2 :&\quad over(X) \leftarrow not\ good(X)\\
r_3 :&\ super(a,b) \leftarrow\\
r_4 :&\ super(b,c) \leftarrow\\
r_5 :&\quad over(b) \leftarrow\\
r_6 :&\quad over(c) \leftarrow
\end{aligned}
$$

where r_1 indicates that if X is supervising Y and Y is not overloaded, then X is a good manager and r_2 indicates that if X is not a good manager, then X is overloaded. The facts r_3-r_6 define a supervising hierarchy and indicate that b and c are overloaded. We then have that $P \models^{wf} over(a)$.

2.2.2.2 Description Logics

For space constraints, we assume the reader is familiar with DLs and adopt the usual conventions, see [8]. We highlight some points below.

A *DL knowledge base (KB)* $\Sigma = \langle \mathcal{T}, \mathcal{A} \rangle$ consists of a finite set $\mathcal{T}$ (called *TBox*) of *terminological* and *role axioms* $\alpha \sqsubseteq \beta$, where α and β are concept (respectively role) expressions, and a finite set $\mathcal{A}$ (called *ABox*) of assertions $A(o_1)$ and $R(o_1, o_2)$ where A is a concept name, R is a role name, and o_1, o_2 are individuals (i.e., constants). We also view Σ as the set $\mathcal{T} \cup \mathcal{A}$.

For particular classes of DL KBs Σ, we assume that (1) Σ is defined over a (finite) set $\mathcal{P}_o$ of concept and role names; we call the constants appearing in Σ the *Herbrand domain of* Σ, denoted with $\Delta_{\mathcal{H}(\Sigma)}$; (2) Σ can be extended with arbitrary assertions, i.e., for any ABox $\mathcal{A}'$ (over $\mathcal{P}_o$), $\Sigma \cup \mathcal{A}'$ is an admissible DL KB, and (3) Σ defines a ground entailment relation $\models$ such that $\Sigma \models Q(\mathbf{e})$ is defined for *dl-queries* $Q(\mathbf{e})$, $\mathbf{e}$ ground terms, which indicates that all models of Σ satisfy $Q(\mathbf{e})$. Here, a *dl-query* $Q(\mathbf{t})$ is either of the form (a) $C(t)$, where C is a concept and t is a term; or (b) $R(t_1, t_2)$, where R is a role and t_1, t_2 are terms.

DLs are build up using *concept- and role expressions* — the semantics of which is given as usual by an interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ with a non-empty domain $\Delta^{\mathcal{I}}$ and an interpretation function $\cdot^{\mathcal{I}}$ for concept- and role names as well as individuals. We assume that the unique names assumption holds such that $o^{\mathcal{I}} = o$ for individuals and in particular $\{o\}^{\mathcal{I}} = \{o^{\mathcal{I}}\}$ for *nominals* appearing in the KB. Note that OWL does not have the unique names assumption [52], and thus different individuals can point to the same resource. However, Logic Programming semantics gives a Herbrand interpretation to constants, i.e., constants are interpreted as themselves, and for consistency we assume that also DL nominals are interpreted this way. The semantics of concept and role expressions is defined as usual, see, e.g., [8], in particular $\top^{\mathcal{I}} = \Delta^{\mathcal{I}}$ and $(\top^2)^{\mathcal{I}} = \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ for the dedicate symbols $\top$

and $\top^2$. DL KBs usually consist of a *TBox* with *terminological and role axioms* $C \sqsubseteq D$ where C and D are either concept- or role expressions and an *ABox* consisting of assertions $A(o)$ or $R(o_1, o_2)$ where A is a concept name and R a role name. In the presence of nominals, we can restrict ourselves to TBoxes, as ABox assertions $A(o)$ and $R(o_1, o_2)$ can be rewritten as axioms $\{o\} \sqsubseteq A$ and $\{(o_1, o_2)\} \sqsubseteq R$. In the following, we will assume the DL at hand contains nominals such that we do not have to consider the ABox.

An interpretation $\mathcal{I}$ satisfies an axiom $C \sqsubseteq D$ if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ and it is a *model* of a KB Σ, denoted $\mathcal{I} \models \Sigma$, if it satisfies every axiom in Σ. A concept C is *satisfiable* w.r.t. Σ if there is a model $\mathcal{I}$ of Σ such that $C^{\mathcal{I}} \neq \emptyset$.

For a domain Δ and a KB Σ, we will w.l.o.g denote interpretations $(\Delta, \cdot^{\mathcal{I}})$ as sets consisting of $\{A(x)|x \in A^{\mathcal{I}}\} \cup \{P(x, y)|(x, y) \in P^{\mathcal{I}}\} \cup \{\{o\}(o)\}$ for concepts names A, role names P, and individuals o in Σ (and thus no $\{o\}(i)$ is present for $o \neq i$).

Instead of $x \in C^{\mathcal{I}}$, we will write $\mathcal{I} \models C(x)$ and instead of $(x, y) \in E^{\mathcal{I}}$, we will write $\mathcal{I} \models E(x, y)$ for such sets $\mathcal{I}$ and concept (role) expressions C (E). The latter can be easily defined according to the usual DLs definition of interpretations of concept (role) expressions. For example, $\{A(x), B(x)\} \models (A \sqcap B)(x)$. Note that $\{o\}(o)$ is present in each such interpretation for each individual o appearing in Σ. We furthermore assume that each such interpretation contains $\top(x)$ for every $x \in \Delta$ as well as $\top^2(x, y)$ for all $x, y \in \Delta$. If an interpretation $\mathcal{I}$ is assumed to be of the above form we will call it a *set-interpretation*. Recall, however, that they are just notational variants of interpretations and are thus in one-to-one correspondence.

Finally, the main reasoning service we are interested in this report is a particular type of subsumption checking: *ground atom entailment*, *entailment* for short. For a ground atom $C(a)$ where a is an individual, C a concept expression, and Σ a KB, we are interested whether for all models $\mathcal{I}$ of Σ, $a^{\mathcal{I}} = a \in C^{\mathcal{I}}$, denoted $\Sigma \models C(a)$.[2]

Exemplary DLs are $\mathcal{SHOIN}(\mathbf{D})$ and $\mathcal{SROIQ}(\mathbf{D})$ which provide the logical underpinnings of the Web ontology languages OWL DL and OWL 2 (see [37, 38, 44] for further background). In what follows we assume that the reader is familiar with standard DL syntax and semantics.

Example 2.2.2. *Take the DL KB* Σ*:*

$$\begin{array}{rcl} (\geqslant 2\ PaptoRev.\top) & \sqsubseteq & Over \\ Over & \sqsubseteq & \forall Super^{+}.Over \\ \{(a, b)\} \sqcup \{(b, c)\} & \sqsubseteq & Super \end{array}$$

where $Super^{+}$ *is the transitive closure of the role* $Super$*. The first two axioms indicate that someone who has more than two papers to review is overloaded, and that an overloaded person causes all the supervised persons to be overloaded as well (otherwise the manager delegates badly). The final axiom — equivalent to the assertions* $Super(a, b)$ *and* $Super(b, c)$ *— defines the supervision hierarchy.*

[2] When C is a concept name, ground entailment is also known as *instance checking*.

2.2.2.3 DL-Programs under Well-Founded Semantics

We introduce dl-programs under well-founded semantics [16].

Informally, a dl-program consists of a DL KB Σ over $\mathcal{P}_o$ and a DATALOG$^\neg$ program P over a set of predicates $\mathcal{P}_p$ distinct from $\mathcal{P}_o$, which may contain queries to Σ. Roughly, such queries ask whether a certain ground atom logically follows from Σ. Note that the Herbrand domains of Σ and P are not necessarily distinct.

Syntax. A *dl-atom* $a(\mathbf{t})$ has the form

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

where each S_i is either a concept or a role name from $\mathcal{P}_o$, p_i is a unary, resp. binary, predicate symbol from $\mathcal{P}_p$, and $Q(\mathbf{t})$ is a dl-query. We call the list $S_1 \uplus p_1, \ldots, S_m \uplus p_m$ the *input signature* and $p_1, \ldots, p_m$ the *input predicate symbols*. Intuitively, $\uplus$ increases S_i by the extension of p_i prior to the evaluation of query $Q(\mathbf{t})$.[3]

A *dl-rule* r has the form (2.1), where any atom b_i, c_j may be a dl-atom. A *dl-program* $KB = (\Sigma, P)$ consists of a DL KB Σ and a finite set of dl-rules P. We say KB *is over* $\mathcal{DL}$, if Σ is in the DL $\mathcal{DL}$.

Semantics. We define the *Herbrand base* $\mathcal{B}_{KB}$ of a dl-program (Σ, P) as the set of ground atoms with predicate symbols from P (i.e., from $\mathcal{P}_p$) and constants from the Herbrand domains of Σ and P. The ground program $gr(P)$ is the ground of P over $\mathcal{B}_{KB}$. An *interpretation* of KB is any subset $I \subseteq \mathcal{B}_{KB}$. It satisfies a ground atom a under Σ, denoted $I \models_\Sigma a$,

– in case a is a non-dl-atom, iff $I \models a$, and
– in case a is a dl-atom of form (2.2), iff $\Sigma \cup \tau^I(a) \models Q(\mathbf{c})$,

where $\tau^I(a)$, the *extension of* a *under* I, is $\tau^I(a) = \bigcup_{i=1}^m A_i(I)$ with $A_i(I) = \{S_i(\mathbf{e}) \mid p_i(\mathbf{e}) \in I\}$. Satisfaction of ground dl-rules r under Σ is then as usual (see DATALOG$^\neg$) and denoted with $I \models_\Sigma r$. I is a model of KB, denoted $I \models KB$, iff $I \models_\Sigma r$ for all $r \in gr(P)$.

We define the well-founded semantics for dl-programs as in [16] using the γ^2 operator. For I and (Σ, P), let $KB^I = (\Sigma, sP^I_\Sigma)$, the *reduct of* KB *wrt.* I, be the dl-program where sP^I_Σ results from $gr(P)$ by deleting (1) every dl-rule r where $I \models_\Sigma a$ for some $a \in B^-(r)$, and (2) from the remaining dl-rules r the negative body $B^-(r)$. Note that sP^I_Σ may still contain positive dl-atoms. As shown in [16], KB^I has a single minimal model, denoted $MM(KB^I)$.

Now the operator γ_{KB} on interpretations I of KB is defined by $\gamma_{KB}(I) = MM(KB^I)$. As γ_{KB} is anti-monotone, $\gamma^2_{KB}(I) = \gamma_{KB}(\gamma_{KB}(I))$ is monotone and has a least fixpoint. This fixpoint is the set of *well-founded* atoms of KB, denoted $WFS(KB)$; we denote with $KB \models^{wf} a$ that $a \in WFS(KB)$.

[3] Other modifiers, like $\uplus, \Cap$, may be expressed by $\uplus$ in strong enough DLs and similarly for subsumption expressions $C \sqsubseteq D$. However, DATALOG-rewritability precludes such constructs.

Example 2.2.3. *Take* (Σ, P) *where* Σ *as in Example 2.2.2 and* P*:*

$$\begin{array}{lrcl} r_1: & good(X) & \leftarrow & \mathrm{DL}[;\, Super](X,\, Y), \\ & & & not\ \mathrm{DL}[PaptoRev \uplus paper;\, Over](Y); \\ r_2: & over(X) & \leftarrow & not\ good(X); \\ r_3: & paper(b, p_1) & \leftarrow & ; \\ r_4: & paper(b, p_2) & \leftarrow & . \end{array}$$

Note that the first dl-atom has no input signature. Intuitively, r_1 *indicates that if* X *is supervising* Y *and* Y *is not overloaded, then* X *is a good manager and* r_2 *indicates that if* X *is a not a good manager then* X *is overloaded. Then,* $KB \models^{wf} over(a)$*.*

Deciding $(\Sigma, P) \models^{wf} a$ is combined complete for EXPTIME ($\text{PTIME}^{\text{NEXP}}$) for Σ in $\mathcal{SHIF}(\mathbf{D})$ ($\mathcal{SHOIN}(\mathbf{D})$) and data complete for PTIME^{NP} for Σ in $\mathcal{SHIF}(\mathbf{D})$ and $\mathcal{SHOIN}(\mathbf{D})$[16]; here data complete means that only the constants in Σ and P, the ABox $\mathcal{A}$, and the facts in P may vary.

2.2.3 Reducing DL-Programs to DATALOG$^\neg$

Let $KB = (\Sigma, P)$ be a dl-program and let a be a ground atom from $\mathcal{B}_{KB}$. We define a class of DLs, so-called DATALOG-*rewritable* DLs, such that reasoning w.r.t. dl-programs over such DLs becomes reducible to DATALOG$^\neg$. In particular, we show that for such DATALOG-rewritable DLs, we can reduce a dl-program $KB = (\Sigma, P)$ to a DATALOG$^\neg$ program $\Psi(KB)$ such that $KB \models^{wf} a$ iff $\Psi(KB) \models^{wf} a$.

We abstractly define which DLs we consider DATALOG-rewritable.

Definition 2.2.4. *A DL* $\mathcal{DL}$ *is* DATALOG-rewritable *if there exists a transformation* $\Phi_{\mathcal{DL}}$ *from* $\mathcal{DL}$ *KBs to* DATALOG *programs such that, for any* $\mathcal{DL}$ *KB* Σ*,*

(i) $\Sigma \models Q(\mathbf{o})$ *iff* $\Phi_{\mathcal{DL}}(\Sigma) \models Q(\mathbf{o})$ *for any concept or role name* Q *from* Σ*, and individuals* $\mathbf{o}$ *from* Σ*;*

(ii) $\Phi_{\mathcal{DL}}$ *is* modular*, i.e., for* $\Sigma = \langle \mathcal{T}, \mathcal{A} \rangle$ *where* $\mathcal{T}$ *is a TBox and* $\mathcal{A}$ *an ABox,* $\Phi_{\mathcal{DL}}(\Sigma) = \Phi_{\mathcal{DL}}(\mathcal{T}) \cup \mathcal{A}$*;*

In other words, a ground atom a is entailed by the $\mathcal{DL}$ KB L iff $a \in MM(\Phi_{\mathcal{DL}}(\Sigma))$, the unique minimal model of the DATALOG program $\Phi_{\mathcal{DL}}(\Sigma)$. Furthermore, we refer to a *polynomial* DATALOG-rewritable DL $\mathcal{DL}$, if $\Phi_{\mathcal{DL}}(\Sigma)$ for a $\mathcal{DL}$ KB Σ is computable in polynomial time.

We assume w.l.o.g. that both P and $\Phi_{\mathcal{DL}}(\Sigma)$ are *safe* — each variable appears in a positive normal atom in the body — for (Σ, P).

Let $\Lambda_P \triangleq \{\lambda \mid DL[\lambda; Q] \text{ occurs in } P\}$, i.e., the input signatures appearing in P. The translation of (Σ, P) to a DATALOG$^\neg$ program is then built up of the following four components:

- $\Sigma_{\Lambda_P} \triangleq \cup_{\lambda \in \Lambda_P} \Sigma_\lambda$ where Σ_λ is Σ with all concept and role names subscripted with λ. Intuitively, each input signature of a dl-atom in P will influence Σ differently. As we want to cater for these influences in one program, we have to differentiate between the KBs with different inputs.
- A DATALOG program $\rho(\Lambda_P)$ containing for each $\lambda = S_1 \uplus p_1, \ldots, S_m \uplus p_m \in \Lambda_P$ the rules $S_{i\lambda}(\mathbf{X_i}) \leftarrow p_i(\mathbf{X_i})$, $1 \leqslant i \leqslant m$, where the arity of $\mathbf{X_i}$ matches the one of S_i. Intuitively, we add the extension of p_i to the appropriate concept or role.
- A set T_P of DATALOG rules $\top(a) \leftarrow$ and $\top^2(a, b) \leftarrow$ for all a, b in the Herbrand domain of P to ensure their introduction in Σ.
- Finally, P^{ord} results from replacing each dl-atom $DL[\lambda; Q](\mathbf{t})$ in P with a new atom $Q_\lambda(\mathbf{t})$.

The transformation of the dl-program KB is then defined as

$$\Psi(KB) \triangleq \Phi_{\mathcal{DL}}(\Sigma_{\Lambda_P}) \cup P^{ord} \cup \rho(\Lambda_P) \cup T_P \tag{2.3}$$

Example 2.2.5. *Let* $KB = (\Sigma, P)$ *where* $\Sigma = \{\ C \sqsubseteq D\ \}$ *and*

$$P \triangleq \{\ p(a) \leftarrow ; \quad s(a) \leftarrow ; \quad s(b) \leftarrow ;$$
$$q \leftarrow DL[C \uplus s; D](a), not\ DL[C \uplus p; D](b)\ \}.$$

Then $\Lambda_P = \{\lambda_1 \triangleq C \uplus s, \lambda_2 \triangleq C \uplus p\}$*, such that* $\rho(\Lambda_P)$ *consists of* $C_{\lambda_1}(X) \leftarrow s(X)$ *and* $C_{\lambda_2}(X) \leftarrow p(X)$*. The component* P^{ord} *consists of* $q \leftarrow D_{\lambda_1}(a), not\ D_{\lambda_2}(b)$ *and the original facts.*

Note that $\Psi(KB)$ is a DATALOG program, if KB is negation-free, and a stratified DATALOG$^\neg$ program, if KB is stratified (cf. [15]); thus, beneficial for evaluation, acyclic negation is fully preserved.

Proposition 2.2.6. *Let* KB *be a dl-program over a polynomial* DATALOG*-rewritable DL. Then,* $\Psi(KB)$ *is constructible in polynomial time.*

Proof. This immediately follows from the defition of $\Psi(KB)$. □

In order to show that $KB \models^{wf} a$ iff $\Psi(KB) \models^{wf} a$ for an atom $a \in \mathcal{B}_{KB}$, we use the following intermediate lemmas, similarly as for the proof of Theorem 5.12 in [16].

For a dl-program $KB = (\Sigma, P)$ over a DATALOG-rewritable DL and an interpretation I over $\mathcal{B}_{KB}$, we define an interpretation for $\Psi(KB)$:

$$I^\Psi \triangleq I \cup \bigcup_{\lambda \in \Lambda_P} MM(\Phi(\Sigma_\lambda \cup S(\lambda, I)))$$

where

$$S(\lambda, I) \triangleq \{\{\mathbf{c}\} \sqsubseteq S_\lambda \mid S \uplus p \in \lambda, p(\mathbf{c}) \in I\}$$

In other words for an interpretation I of the KB KB, we define an interpretation I^Ψ of $\Psi(KB)$ that corresponds to it, i.e., it contains I and the minimal models of the positive programs consisting of the translation of the KB as well as the facts that follow from the particular extensions of the input predicates w.r.t. I.

We further define some shortcuts: $G(I) \triangleq \gamma_{KB}(I)$ and $G^\Psi(I) = \gamma_{\Psi(KB)}(I)$.

Lemma 2.2.7. *Let $KB = (\Sigma, P)$ be a dl-program over a* DATALOG*-rewritable DL, $DL[\lambda; Q](\mathbf{c})$ a ground dl-atom from $gr(P)$, and I an interpretation for KB. Then, $I \models_\Sigma DL[\lambda; Q](\mathbf{c})$ iff $I^\Psi \models Q_\lambda(\mathbf{c})$.*

Proof.

$I \models_\Sigma DL[\lambda; Q](\mathbf{c})$

iff *[Σ_λ is an equivalent rewriting of Σ; take $\lambda' = S_{1\lambda} \uplus p_1, \ldots, S_{m\lambda} \uplus p_m$ for $\lambda = S_1 \uplus p_1, \ldots, S_m \uplus p_m \in \Lambda_P$]*

$I \models_{\Sigma_\lambda} DL[\lambda'; Q_\lambda](\mathbf{c})$

iff *[Def. of $\models_{\Sigma_\lambda}$ and with $A_i(I) = \{S_{i\lambda}(\mathbf{c_i}) \mid p_i(\mathbf{c_i}) \in I\}$]*

$\Sigma_\lambda \cup \bigcup_i A_i(I) \models Q_\lambda(\mathbf{c})$

iff *[Rewriting ABox $\bigcup_i A_i(I)$ as axioms]*

$\Sigma_\lambda \cup \{\{\mathbf{c_i}\} \sqsubseteq S_{i\lambda} \mid p_i(\mathbf{c_i}) \in I\} \models Q_\lambda(\mathbf{c})$

iff *[Def. 2.2.4]*

$Q_\lambda(\mathbf{c}) \in MM(\Phi(\Sigma_\lambda \cup \{\{\mathbf{c_i}\} \sqsubseteq S_{i\lambda} \mid p_i(\mathbf{c_i}) \in I\}))$

iff *[using that the Σ_λ disjoint on the concept- and role names, that Φ is preserving, and that Q_λ is a concept- or role name]*

$Q_\lambda(\mathbf{c}) \in \bigcup_{\lambda \in \Lambda_P} MM(\Phi(\Sigma_\lambda \cup S(\lambda, I)))$

iff *[Q_λ is concept or role name]*

$Q_\lambda(\mathbf{c}) \in I \cup \bigcup_{\lambda \in \Lambda_P} MM(\Phi(\Sigma_\lambda \cup S(\lambda, I)))$

iff *[Def. of I^Ψ]*

$Q_\lambda(\mathbf{c}) \in I^\Psi$

Lemma 2.2.8. *Let $KB = (\Sigma, P)$ be a dl-program over a* DATALOG*-rewritable DL, and I an interpretation for KB. Then, $G(I)^\Psi = G^\Psi(I^\Psi)$.*

Proof. This follows from Lemma 2.2.7 and the observation that sP^I_Σ and $(P^{ord})^{I^\Psi}$ have the same rules where the (positive) dl-atoms $DL[\lambda; Q](\mathbf{c})$ in sP^I_Σ are replaced with $Q_\lambda(\mathbf{c})$ in $(P^{ord})^{I^\Psi}$.

Lemma 2.2.9. *Let $KB = (\Sigma, P)$ be a dl-program over a* DATALOG*-rewritable DL, and I an interpretation for KB. Then,* $\mathrm{LFP}(G^2)^\Psi = \mathrm{LFP}((G^\Psi)^2)$.

Proof. The proof technique is similar as the one used in Proposition B.3 in [16], using Lemmas 2.2.7 and 2.2.8.

The following result allows us to reduce reasoning with dl-programs to DATALOG$^\neg$ under well-founded semantics.

Theorem 2.2.1. *Let KB be a dl-program over a* DATALOG*-rewritable DL and a be a ground atom from $\mathcal{B}_{KB}$. Then,*

$KB \models^{wf} a$ *iff* $\Psi(KB) \models^{wf} a$.

Proof. We show both directions.
($\Rightarrow$). Assume $KB \models^{wf} a$. Then, by definition of the well-founded semantics for dl-programs, $a \in \mathrm{LFP}(\gamma_{KB}^2) = \mathrm{LFP}(G^2)$. Since for any interpretation $I \subseteq \mathcal{B}_{KB}$, $I \subseteq I^\Psi$, we have that $a \in \mathrm{LFP}(G^2)^\Psi$. By Lemma 2.2.9, we have that $a \in \mathrm{LFP}((G^\Psi)^2) = \mathrm{LFP}((\gamma_{\Psi(KB)})^2)$, and thus $\Psi(KB) \models^{wf} a$.
($\Leftarrow$). Since a is a ground atom from $\mathcal{B}_{KB}$ and thus constructed with a predicate from P, the reverse reasoning from ($\Leftarrow$) also holds. □

From Theorem 2.2.1 and the fact that any DATALOG$^\neg$ program P amounts to a dl-program $(\emptyset, P)$ [16], we obtain the following result.

Corollary 2.2.10. *For any dl-program KB over a DL $\mathcal{DL}$ and ground atom a from $\mathcal{B}_{KB}$, deciding $KB \models^{wf} a$ is (i) data complete for* PTIME*, if $\mathcal{DL}$ is* DATALOG*-rewritable and (ii) combined complete for* EXPTIME*, if $\mathcal{DL}$ is polynomial* DATALOG*-rewritable.*

Thus, over DATALOG-rewritable DLs, the data complexity of dl-programs decreases from PTIME$^{\mathrm{NP}}$ to PTIME compared to $\mathcal{SHIF}(\mathbf{D})$and $\mathcal{SHOIN}(\mathbf{D})$, and the combined complexity from PTIME$^{\mathrm{NExp}}$ to EXPTIME compared to $\mathcal{SHOIN}(\mathbf{D})$(and is the same as for $\mathcal{SHIF}(\mathbf{D})$) over polynomial DATALOG-rewritable DLs.

2.2.4 The Description Logic $\mathcal{LDL}^+$

In this section, we introduce the Description Logic $\mathcal{LDL}^+$ and derive some basic model-theoretic properties.

2.2.4.1 Basic Definitions

We design $\mathcal{LDL}^+$ by syntactic restrictions on the expressions that occur in axioms, distinguishing between occurrence in the "body" α and the "head" β of an axiom $\alpha \sqsubseteq \beta$. We define

- *h-roles* (h for *head*) E, F to be *role names* P, *role inverses* E^-, *role conjunctions* $E \sqcap F$, and *role top* $\top^2$;

- *b-roles* (b for *body*) E, F are the same as h-roles, plus *role disjunctions* $E \sqcup F$, *role sequences* $E \circ F$, *transitive closures* E^+, *role nominals* $\{(o_1, o_2)\}$, where o_1, o_2 are individuals.

Furthermore, let *basic concepts* C, D be concept names A, the top symbol $\top$, and *conjunctions* $C \sqcap D$; then we define

- *h-concepts* C, D are *basic concepts* B, and *value restrictions* $\forall E.B$ where E a **b-role**;
- *b-concepts* C, D are *basic concepts* B, *disjunctions* $C \sqcup D$, *exists restrictions* $\exists E.C$, *atleast restrictions* $\geqslant nE.C$, and *nominals* $\{o\}$, where E is a b-role, and o is an individual.

Note that all h-roles are also b-roles, but an analog relation does not hold for concepts: $\forall E.C$ is an h-concept but not a b-concept. When immaterial, we will refer to both b-concepts and h-concepts as ($\mathcal{LDL}^+$) concepts; we use an analog convention for roles.

Now an $\mathcal{LDL}^+$ *KB* is a pair $\Sigma = \langle \mathcal{T}, \mathcal{A} \rangle$ of a finite TBox $\mathcal{T}$ and a finite ABox $\mathcal{A}$, where

- $\mathcal{T}$ is a set of *terminological axioms* $B \sqsubseteq H$, where B is a b-concept and H is an h-concept, and *role axioms* $S \sqsubseteq T$, where S is a b-role and T is an h-role, and
- $\mathcal{A}$ is a set of assertions of the form $C(o)$ and $E(o_1, o_2)$ where C is an h-concept and E an h-role.

Example 2.2.11. *Reconsider the DL KB* Σ *from Example 2.2.2. It is easily checked that* Σ *amounts to an* $\mathcal{LDL}^+$ *KB.*

2.2.4.1.1 Normal Form. To simplify matters, we restrict to an expressive normal form of $\mathcal{LDL}^+$ knowledge bases Σ. First, an assertion $C(o)$ is equivalent to the axiom $\{o\} \sqsubseteq C$, and and similarly $E(o_1, o_2)$ is equivalent to $\{(o_1, o_2)\} \sqsubseteq E$; hence, we assume that the ABox is empty and identify Σ with its TBox. Second, every axiom $B \sqsubseteq H$ as above can be equivalently rewritten such that H is either a concept name A, the $\top$ symbol, or $\forall E.A$, where A is a concept name and E is a b-role. Indeed, $B \sqsubseteq C \sqcap D$ could be rewritten as $B \sqsubseteq C$ and $B \sqsubseteq D$, which in turn might have to be rewritten; $B \sqsubseteq \forall E.C$ could be rewritten as $B \sqsubseteq \forall E.A$ and $A \sqsubseteq C$ and $C \sqsubseteq A$ for some new concept name A where $A \sqsubseteq C$ might need to be rewritten further (note that C is a basic concept expression such that it is also a valid b-concept expression). Since the concepts become smaller this is clearly a finite process, which is moreover linear in the size of Σ. We can thus rewrite a $\mathcal{LDL}^+$ KB such that it does not contain conjunction in the head. We can similarly remove conjunction from the head T of role axioms $S \sqsubseteq T$, and restrict the h-role T to role names, inverse role names, and $\top^2$.

Proposition 2.2.12. *Every $\mathcal{LDL}^+$ KB Σ can be transformed into the form described in polynomial (in fact, in linear) time.*

In the sequel, we tacitly deal with such *normalized* $\mathcal{LDL}^+$ KBs.

2.2.4.2 Immediate Consequence Operator

In this section, we define an immediate consequence operator for $\mathcal{LDL}^+$ that allows us to calculate the ground entailment of atoms. Moreover, we show that ground entailment for $\mathcal{LDL}^+$ is domain independent, and thus can be confined to the constants in the KB.

We first show that b-concepts satisfy a *monotonicity* property. For a given KB Σ and interpretations $\mathcal{I} = (\Delta, \cdot^{\mathcal{I}})$ and $\mathcal{J} = (\Delta, \cdot^{\mathcal{J}})$ over the same domain Δ, we write $\mathcal{I} \subseteq \mathcal{J}$ if $A^{\mathcal{I}} \subseteq A^{\mathcal{J}}$ for concept names A in Σ and $P^{\mathcal{I}} \subseteq P^{\mathcal{J}}$ for role names P in Σ; note that $o^{\mathcal{I}} = o^{\mathcal{J}}$ for any individual o due to the unique names assumption. Then $\mathcal{I} \subset \mathcal{J}$ if $\mathcal{I} \subseteq \mathcal{J}$ but $\mathcal{I} \neq \mathcal{J}$. We say that a model $\mathcal{I} = (\Delta, \cdot^{\mathcal{I}})$ of Σ is *minimal*, if there is no model $\mathcal{J} = (\Delta, \cdot^{\mathcal{J}})$ of Σ such that $\mathcal{J} \subset \mathcal{I}$.

Definition 2.2.13. *An $\mathcal{LDL}^+$ concept (role) C (E) is* monotonic*, if for each pair of interpretations $\mathcal{I} = (\Delta, \cdot^{\mathcal{I}})$ and $\mathcal{J} = (\Delta, \cdot^{\mathcal{J}})$ of Σ, $\mathcal{I} \subseteq \mathcal{J}$ implies $C^{\mathcal{I}} \subseteq C^{\mathcal{J}}$ ($E^{\mathcal{I}} \subseteq E^{\mathcal{J}}$).*

Proposition 2.2.14. *All b-concepts and all $\mathcal{LDL}^+$ roles are monotonic.*

Proof. This can be easily inductively proved on the structure of the concepts and roles. □

Note that an h-concept $\forall E.B$ is not monotonic.

Recall that we can write interpretations $\mathcal{I} = (\Delta, \cdot^{\mathcal{I}})$ as *set-interpretations* $\{A(x) | x \in A^{\mathcal{I}}\} \cup \{P(x,y) | (x,y) \in P^{\mathcal{I}}\} \cup \{\{o\}(o)\}$ for concept (role) names A (P), and for individuals o. Instead of $x \in C^{\mathcal{I}}$ ($(x,y) \in E^{\mathcal{I}}$), we write $\mathcal{I} \models C(x)$ ($\mathcal{I} \models E(x,y)$) for concepts (roles) C (E). Note that each such $\mathcal{I}$ contains $\top(x)$ for every $x \in \Delta$ as well as $\top^2(x,y)$ for all $x, y \in \Delta$.

One can see that for a fixed Δ, the set $\mathbf{I}_{\mathbf{\Delta}}$ of all set-interpretations over Δ is under the usual subset relation $\subseteq$ a complete lattice as in [55].

Indeed, it is a non-empty set, $\subseteq$ is a partial order on $\mathbf{I}_{\mathbf{\Delta}}$, and for any two sets $\mathcal{I}$ and $\mathcal{J}$ in $\mathbf{I}_{\mathbf{\Delta}}$ there is a least upper bound $\mathcal{I} \cup \mathcal{J}$ and greatest lower bound $\mathcal{I} \cap \mathcal{J}$. Moreover, it is a complete lattice as every subset $\mathbf{S} \subseteq \mathbf{I}_{\mathbf{\Delta}}$ has a least upper bound $\bigcup \mathbf{S}$ and greatest lower bound $\bigcap \mathbf{S}$. In particular, there are elements $0_\Delta \triangleq \bigcap \mathbf{I}_{\mathbf{\Delta}}$ and $1_\Delta \triangleq \bigcup \mathbf{I}_{\mathbf{\Delta}}$. Note that 0_Δ is the set $\{\{o\}(o) \mid o \text{ ind. in } \Sigma\} \cup \{\top(x), \top^2(x,y) \mid x, y \in \Delta\}$.

For an $\mathcal{LDL}^+$ KB Σ and a domain Δ, we then define an *immediate consequence operator* T_Δ on $\mathbf{I}_\Delta$ as follows, where A ranges over the concept names, P over the role names, and

x, y over Δ:

$$\begin{aligned} T_\Delta(\mathcal{I}) = \mathcal{I} &\cup \{A(x) \mid B \sqsubseteq A \in \Sigma, \mathcal{I} \models B(x)\} \\ &\cup \{A(x) \mid B \sqsubseteq \forall E.A \in \Sigma, \mathcal{I} \models B(y), \mathcal{I} \models E(y,x)\} \\ &\cup \{P(x,y) \mid S \sqsubseteq P \in \Sigma, \mathcal{I} \models S(x,y)\} \\ &\cup \{P(y,x) \mid S \sqsubseteq P^- \in \Sigma, \mathcal{I} \models S(x,y)\} \ . \end{aligned}$$

For a set-interpretation $\mathcal{I}$ of Σ over Δ, $T_\Delta(\mathcal{I})$ is still a set-interpretation of Σ over Δ, such that T_Δ is well-defined.

As easily seen, T_Δ is *monotone*, i.e., $\mathcal{J} \subseteq \mathcal{I}$ implies $T_\Delta(\mathcal{J}) \subseteq T_\Delta(\mathcal{I})$, and thus has a least fixpoint $\mathrm{LFP}(T_\Delta)$, i.e., a unique minimal $\mathcal{I}$ such that $T_\Delta(\mathcal{I}) = \mathcal{I}$ [55].

Proposition 2.2.15. *Let Σ be an $\mathcal{LDL}^+$ KB, Δ a domain. Then, T_Δ is increasing and has a least fixed point, i.e., there is an $\mathcal{I} \in \mathbf{I}_\Delta$ such that $T_\Delta(\mathcal{I}) = \mathcal{I}$ and no $\mathcal{J} \in \mathbf{I}_\Delta$ with $\mathcal{J} \subset \mathcal{I}$ exists such that $T_\Delta(\mathcal{J}) = \mathcal{J}$.*

Proof. If T_Δ is increasing, the fixed point result follows directly from [55, Theorem 2]. We show that T_Δ is indeed increasing.

Assume $\mathcal{I} \subseteq \mathcal{J}$, set-interpretations of Σ over Δ; we show that $T_\Delta(\mathcal{I}) \subseteq T_\Delta(\mathcal{J})$. Take an $A(x) \in T_\Delta(\mathcal{I})$, then either (1) $A(x) \in \mathcal{I}$, (2) there is a terminological axiom $B \sqsubseteq A$ such that $\mathcal{I} \models B(x)$, or (3) there is a terminological axiom $B \sqsubseteq \forall E.A$ such that $\mathcal{I} \models B(y)$ and $\mathcal{I} \models E(y,x)$ for some $y \in \Delta$. Since $\mathcal{I} \subseteq \mathcal{J}$, (1) leads immediately to $A(x) \in \mathcal{J} \subseteq T_\Delta(\mathcal{J})$. For (2), we have, due to monotonicity of B that $\mathcal{J} \models B(x)$ such that again $A(x) \in T_\Delta(\mathcal{J})$. For (3), we have again due to monotonicity of E and B, that $\mathcal{J} \models B(y)$ and $\mathcal{J} \models E(y,x)$ such that $A(x) \in T_\Delta(\mathcal{J})$.

The case for a $P(x,y) \in T_\Delta(\mathcal{I})$ can be done similarly. □

This fixpoint corresponds to a model of Σ, which in fact is the single minimal model of Σ over Δ.

Proposition 2.2.16. *Let Σ be an $\mathcal{LDL}^+$ KB and let Δ be a domain over Σ. Then, $\mathcal{I} = (\Delta, \cdot^\mathcal{I})$ is a minimal model of Σ iff $\mathcal{I}$ corresponds to the set-interpretation $\mathrm{LFP}(T_\Delta)$.*

Proof. We abbreviate in the following T_Δ with T and $\mathrm{LFP}(T)$ with L.

($\Rightarrow$) Assume $\mathcal{I} = (\Delta, \cdot^\mathcal{I})$ is a minimal model of Σ. We show that $\mathcal{I}$ corresponds to the set-interpretation L. We assume $\mathcal{I}$ is written in its set-interpretation notation and prove that $\mathcal{I}$ is indeed the least fixed point of T.

1. $\mathcal{I}$ is a fixed point, i.e., $T(\mathcal{I}) = \mathcal{I}$. Clearly, $\mathcal{I} \subseteq T(\mathcal{I})$ by definition of T. Assume $T(\mathcal{I}) \not\subseteq \mathcal{I}$. Then, there is a $A(x)$ or a $R(x,y)$ in $T(\mathcal{I})$ that is not in $\mathcal{I}$. For the case $A(x)$, we have that there is then (1) a $B \sqsubseteq A \in \Sigma$ such that $\mathcal{I} \models B(x)$ or (2) a $B \sqsubseteq \forall E.A \in \Sigma$ such that $\mathcal{I} \models E(y,x)$ and $\mathcal{I} \models B(y)$ for some $y \in \Delta$. For (1),

clearly, then $x \in B^{\mathcal{I}}$ such that (since $\mathcal{I}$ is a model), $x \in A^{\mathcal{I}}$ and thus $A(x) \in \mathcal{I}$, a contradiction. For (2), $(y, x) \in E^{\mathcal{I}}$ and $y \in B^{\mathcal{I}}$ such that, since $\mathcal{I}$ is a model, $y \in (\forall R.A)^{\mathcal{I}}$ and thus, with $(y, x) \in E^{\mathcal{I}}$, that $x \in A^{\mathcal{I}}$, a contradiction. The case $R(x, y)$ can be done similarly. Thus, $T(\mathcal{I}) = \mathcal{I}$.

2. Assume it is not a least fixed point, then there is a $\mathcal{J}$ such that $\mathcal{J} \subset \mathcal{I}$ and $T(\mathcal{J}) = \mathcal{J}$. We show that $\mathcal{J} = (\Delta, \cdot^{\mathcal{J}})$ is then a model of Σ, violating the minimality of $\mathcal{I}$.

   Take a terminological axiom $B \sqsubseteq H \in \Sigma$ and $x \in B^{\mathcal{J}}$. Then $\mathcal{J} \models B(x)$. Assume $H = A$ for a concept name A, then $H(x) \in T(\mathcal{J})(= \mathcal{J})$ by definition of T such that $x \in H^{\mathcal{J}}$. Assume $H = \forall E.A$ for a concept name A. Then, $x \in (\forall E.A)^{\mathcal{J}}$. Indeed, if there is a $(x, y) \in E^{\mathcal{J}}$, then $\mathcal{J} \models E(x, y)$ such that, by definition of T, $A(y) \in T(\mathcal{J}) = \mathcal{J}$ such that $\mathcal{J} \models A(y)$. Thus, $\mathcal{J} \models (\forall E.A)(x)$ and $x \in (\forall E.A)^{\mathcal{J}}$. Assume $H = \top$, then the axiom is trivially satisfied.

   Role axioms can be treated similarly.

($\Leftarrow$) Assume $\mathcal{I} = L$. That $\mathcal{I}$ is a minimal model can be showed using similar techniques as in the other direction. □

We show that for a given domain Δ, the minimal model is unique, i.e., if both $(\Delta, \mathcal{I})$ and $(\Delta, \mathcal{J})$ are minimal models, then $\mathcal{I} = \mathcal{J}$.

Proposition 2.2.17. *Let Σ be an $\mathcal{LDL}^+$ KB and let Δ be a domain over Σ with minimal models $\mathcal{I}$ and $\mathcal{J}$ over Δ. Then, $\mathcal{I} = \mathcal{J}$.*

Proof. By Proposition 2.2.16, we have that $\mathcal{I}$ and $\mathcal{J}$ are least fixed points of T. Define $\mathcal{K} \triangleq \mathcal{I} \cap \mathcal{J}$. Then one can show that $T(\mathcal{K}) = \mathcal{K}$ using monotonicity and that $T(\mathcal{I}) = \mathcal{I}$ and $T(\mathcal{J}) = \mathcal{J}$. Hence, since $\mathcal{I}$ and $\mathcal{J}$ are least fixed points, $\mathcal{K} = \mathcal{I} = \mathcal{J}$. □

Corollary 2.2.18. *Let Σ be an $\mathcal{LDL}^+$ KB and let Δ be a domain over Σ. Then, there exists a unique minimal model $\mathcal{I} = (\Delta, \cdot^{\mathcal{I}})$, denoted $MM(\Delta, \Sigma)$, that equals* $\mathrm{LFP}(T_\Delta)$.

Proof. By Proposition 2.2.15, T_Δ has a least fixed point, which is, by Proposition 2.2.16, equal to the minimal model. The latter is unique by Proposition 2.2.17. □

Entailment checking of b-concepts can then in each domain be restricted to the unique minimal model for that domain.

Proposition 2.2.19. *Let Σ be an $\mathcal{LDL}^+$ KB, C a b-concept, and $o \in \Delta_{\mathcal{H}(\Sigma)}$. Then, $\Sigma \models C(o)$ iff for all Δ, $MM(\Delta, \Sigma) \models C(o)$.*

Proof. The ($\Rightarrow$) follows immediately.

For ($\Leftarrow$), we show that if minimal models entail $C(o)$, then all models do. Take a model $(\Delta, \cdot^{\mathcal{I}_0})$, then either the corresponding set interpretation $\mathcal{I}_0$ is minimal for the domain Δ

or not. If it is, we are done, otherwise, there is a model $(\Delta, \cdot^{\mathcal{I}_1})$ of Σ such that $\mathcal{I}_1 \subset \mathcal{I}_0$ for which one repeats the above reasoning, i.e., eventually, we will have a minimal model $\mathcal{I}_n$ such that $\mathcal{I}_n \subset \ldots \mathcal{I}_1 \subset \mathcal{I}_0$ for which $\mathcal{I}_n \models C(o)$. Since C is monotonic, we have that $\mathcal{I}_0 \models C(o)$. $\square$

Note that the proposition does not necessarily hold if C is an h-concept. For example, consider $\Sigma = \{\{a\} \sqsubseteq A\}$ and the h-concept $C = \forall R.A$, where A is a concept name and R is a role name. Clearly, $\Sigma \not\models \forall R.A(a)$. However, when we consider the domain $\Delta_{\mathcal{H}(\Sigma)} = \{a\}$, $MM(\Delta_{\mathcal{H}(\Sigma)}, \Sigma) = \{A(a)\}$ and $MM(\Delta_{\mathcal{H}(\Sigma)}, \Sigma) \models \forall R.A(a)$.

Lemma 2.2.20. *Let $\Delta_0 \subseteq \Delta$ be two domains, $C(E)$ a b-concept (role) expression, o, o_1, o_2 some individuals in Δ_0, $\mathcal{I}$ ($\mathcal{J}$) an interpretation on domain Δ_0 (Δ), and $\mathcal{J} = \mathcal{I} \cup \{\top(x), \top^2(x, y) \mid x, y \in \Delta\}$. Then $\mathcal{I} \models C(o)$ iff $\mathcal{J} \models C(o)$ and $\mathcal{I} \models E(o_1, o_2)$ iff $\mathcal{J} \models E(o_1, o_2)$.*

Proof. By induction on the structure of C (E). $\square$

Importantly, the only relevant interpretation domain is the Herbrand domain $\Delta_{\mathcal{H}(\Sigma)}$ of the KB Σ.

Proposition 2.2.21. *Let Σ be an $\mathcal{LDL}^+$ KB, C a b-concept, and $o \in \Delta_{\mathcal{H}(\Sigma)}$. Then, $\Sigma \models C(o)$ iff $MM(\Delta_{\mathcal{H}(\Sigma)}, \Sigma) \models C(o)$.*

Proof. Assume $\Delta_{\mathcal{H}(\Sigma)} = \{o_1, \ldots, o_n\}$, and take an arbitrary domain $\Delta = \{o_1, \ldots, o_n, e_1, \ldots, e_m\}$. Denote $\{\top(x), \top^2(x, y) \mid x, y \in \Delta\}$ by $\top_\Delta$. The zero elements for set-interpretations on $\Delta_{\mathcal{H}(\Sigma)}$ and Δ are $0_{\Delta_{\mathcal{H}(\Sigma)}} = \{\{o\}(o) \mid o \in \Delta_{\mathcal{H}(\Sigma)}\} \cup \top_{\Delta_{\mathcal{H}(\Sigma)}}$ and $0_\Delta = \{\{o\}(o) \mid o \in \Delta_{\mathcal{H}(\Sigma)}\} \cup \top_\Delta$.

We abbreviate in the following $T^k_{\Delta_{\mathcal{H}(\Sigma)}}(0_{\Delta_{\mathcal{H}(\Sigma)}})$ with $\mathcal{I}_k$, $T^k_\Delta(0_\Delta)$ with $\mathcal{J}_k$.

We inductively prove that $\mathcal{I}_k \cup \top_\Delta = \mathcal{J}_k$ for all $k \geqslant 0$.

For $k = 0$, it is easy to verify that $0_\Delta = 0_{\Delta_{\mathcal{H}(\Sigma)}} \cup \top_\Delta$.

For $k + 1$, take an $A(x) \in \mathcal{I}_{k+1}$, then either (1) $A(x) \in \mathcal{I}_k$, (2) there is a terminological axiom $B \sqsubseteq A$ such that $\mathcal{I}_k \models B(x)$, or (3) there is a terminological axiom $B \sqsubseteq \forall E.A$ such that $\mathcal{I}_k \models B(y)$ and $\mathcal{I}_k \models E(y, x)$ for some $y \in \Delta_{\mathcal{H}(\Sigma)}$. (1) leads immediately to $A(x) \in \mathcal{I}_k \subseteq \mathcal{J}_k \subseteq \mathcal{J}_{k+1}$. For (2), due to $\mathcal{I}_k \models B(x)$, by Lemma 2.2.20, we also have $\mathcal{J}_k \models B(x)$ so that $A(x) \in \mathcal{J}_{k+1}$. For (3), we again have that $\mathcal{J}_k \models B(y)$ and $\mathcal{J}_k \models E(y, x)$ so that $\mathcal{J}_{k+1} \models A(x)$. The case for a $P(x, y) \in T_\Delta(\mathcal{I})$ can be done similarly. So we have $\mathcal{I}_k \cup \top_\Delta \subseteq \mathcal{J}_k$.

Using the observation that for $e_i, e_j \in \{e_1, \ldots, e_m\}$, $\mathcal{J}_k \not\models A(e_i)$ for concept name A, and $\mathcal{J}_k \not\models P(e_i, e_j)$ for role name P, the other direction $\mathcal{I}_k \cup \top_\Delta \supseteq \mathcal{J}_k$ can be proved similarly,.

Now we have proved that $\mathcal{I}_k \cup \top_\Delta = \mathcal{J}_k$ for all $k \geqslant 0$. So we have that $\mathrm{LFP}(T_{\Delta_{\mathcal{H}(\Sigma)}}) \cup \top_\Delta = \mathrm{LFP}(T_\Delta)$. By Proposition 2.2.16, we have $MM(\Delta_{\mathcal{H}(\Sigma)}, \Sigma) \cup \top_\Delta = MM(\Delta, \Sigma)$.

Now by Lemma 2.2.20, it follows that for $o \in \Delta_{\mathcal{H}(\Sigma)}$, we have $MM(\Delta_{\mathcal{H}(\Sigma)}, \Sigma) \models C(o)$ iff $\forall \Delta,\ MM(\Delta, \Sigma) \models C(o)$. Finally, by Proposition 2.2.19, we have $\Sigma \models C(o)$ iff $MM(\Delta_{\mathcal{H}(\Sigma)}, \Sigma) \models C(o)$. $\square$

Note that $MM(\Delta_{\mathcal{H}(\Sigma)}, \Sigma) = \mathrm{LFP}(T_{\Delta_{\mathcal{H}(\Sigma)}})$ is effectively constructable by fixpoint iteration (for a finite KB in finite time).

Proposition 2.2.21 is at the core of the argument that $\mathcal{LDL}^+$ is a DATALOG-rewritable DL, which we show in the next section.

2.2.5 $\mathcal{LDL}^+$ is DATALOG-rewritable

To show that a (normalized) $\mathcal{LDL}^+$ KB Σ is DATALOG-rewritable, we construct a suitable DATALOG program $\Phi_{\mathcal{LDL}^+}(\Sigma)$ such that $\Sigma \models Q(\mathbf{o})$ iff $\Phi_{\mathcal{LDL}^+}(\Sigma) \models Q(\mathbf{o})$, whenever Q is a concept- or role name appearing in Σ and $\mathbf{o} \subseteq \Delta_{\mathcal{H}(\Sigma)}$.

Define the *closure* of Σ, $clos(\Sigma)$, as the smallest set containing *(i)* all subexpressions that occur in Σ (both roles and concepts) except value restrictions, and *(ii)* for each role name occurring in Σ, its inverse. The closure is in other words the smallest set satisfying the following conditions:

- $\top$ and $\top^2$ are in $clos(\Sigma)$,
- every concept name A, role name R and its inverted role name R^-, nominal $\{o\}$, and role nominal $\{(o_1, o_2)\}$ appearing in Σ is in $clos(\Sigma)$,
- for each $B \sqsubseteq H$ a terminological axiom in Σ with H a concept name or $H = \top$, $B \in clos(\Sigma)$,
- for each $B \sqsubseteq \forall E.A$ a terminological axiom in Σ, $\{B, E\} \subseteq clos(\Sigma)$,
- for each $S \sqsubseteq T$ a role axiom in Σ, $S \in clos(\Sigma)$,
- for every concept expression D in $clos(\Sigma)$, we have
  - if $D = D_1 \sqcap D_2$, then $\{D_1, D_2\} \subseteq clos(\Sigma)$,
  - if $D = D_1 \sqcup D_2$, then $\{D_1, D_2\} \subseteq clos(\Sigma)$,
  - if $D = \exists E.D_1$, then $\{E, D_1\} \subseteq clos(\Sigma)$,
  - if $D = \geqslant nE.D_1$, then $\{E, D_1\} \subseteq clos(\Sigma)$,
- for every role expression E in $clos(\Sigma)$, we have
  - if $E = F^-$, then $F \in clos(\Sigma)$,
  - if $E = E_1 \sqcap E_2$, then $\{E_1, E_2\} \subseteq clos(\Sigma)$,
  - if $E = E_1 \sqcup E_2$, then $\{E_1, E_2\} \subseteq clos(\Sigma)$,

- if $E = E_1 \circ E_2$, then $\{E_1, E_2\} \subseteq clos(\Sigma)$.
- if $E = E_1^+$, then $E_1 \subseteq clos(\Sigma)$.

Formally, $\Phi_{\mathcal{LDL}^+}(\Sigma)$ is the following program:

- For each axiom $B \sqsubseteq H \in \Sigma$ where H is a concept name, add the rule

$$H(X) \leftarrow B(X) \tag{2.4}$$

- For each axiom $B \sqsubseteq \forall E.A \in \Sigma$ where A is a concept name, add the rule

$$A(Y) \leftarrow B(X), E(X, Y) \tag{2.5}$$

- For each role axiom $S \sqsubseteq T \in \Sigma$, add

$$T(X, Y) \leftarrow S(X, Y) \tag{2.6}$$

  (Here $T = P^-$ may be an inverse for a role name P.)

- For each role name P that occurs in Σ, add the rule

$$P(X, Y) \leftarrow P^-(Y, X) \tag{2.7}$$

- For each concept (role) name or (role) nominal Q (Q') in $clos(\Sigma)$, add the rules

$$\begin{array}{lcl} \top(X) & \leftarrow & Q(X) \\ \top(X) & \leftarrow & Q'(X, Y) \\ \top(Y) & \leftarrow & Q'(X, Y) \end{array} \tag{2.8}$$

  This ensures that newly introduced constants, e.g., in the context of dl-programs, are also assigned to $\top$ — a relevant property for modularity.

- To deduce the top role, add

$$\top^2(X, Y) \leftarrow \top(X), \top(Y). \tag{2.9}$$

- Next, we distinguish between the types of concepts D in $clos(\Sigma)$:
  - if $D = \{o\}$, add

$$D(o) \leftarrow \tag{2.10}$$

  - if $D = D_1 \sqcap D_2$, add

$$D(X) \leftarrow D_1(X), D_2(X) \tag{2.11}$$

– if $D = D_1 \sqcup D_2$, add

$$\begin{array}{rcl} D(X) & \leftarrow & D_1(X) \\ D(X) & \leftarrow & D_2(X) \end{array} \tag{2.12}$$

– if $D = \exists E.D_1$, add the rule

$$D(X) \leftarrow E(X, Y), D_1(Y) \tag{2.13}$$

– if $D = \geqslant n\, E.D_1$, add

$$\begin{array}{rcl} D(X) & \leftarrow & E(X, Y_1), D(Y_1), \ldots, E(X, Y_n), D(Y_n), \\ & & Y_1 \neq Y_2, \ldots, Y_i \neq Y_j, \ldots, Y_{n-1} \neq Y_n \end{array} \tag{2.14}$$

(where $1 \leqslant i < j \leqslant n$).

- Finally, for each role $E \in clos(\Sigma)$:

– if $E = \{(o_1, o_2)\}$, add

$$E(o_1, o_2) \leftarrow \tag{2.15}$$

– if $E = F^-$, add

$$E(X, Y) \leftarrow F(Y, X) \tag{2.16}$$

– if $E = E_1 \sqcap E_2$, add

$$E(X, Y) \leftarrow E_1(X, Y), E_2(X, Y) \tag{2.17}$$

– if $E = E_1 \sqcup E_2$, add

$$\begin{array}{rcl} E(X, Y) & \leftarrow & E_1(X, Y) \\ E(X, Y) & \leftarrow & E_2(X, Y) \end{array} \tag{2.18}$$

– if $E = E_1 \circ E_2$, add

$$E(X, Y) \leftarrow E_1(X, Z), E_2(Z, Y) \tag{2.19}$$

– if $E = F^+$, add

$$\begin{array}{rcl} E(X, Y) & \leftarrow & F(X, Y) \\ E(X, Y) & \leftarrow & F(X, Z), E(Z, Y) \end{array} \tag{2.20}$$

The following property is immediate.

Proposition 2.2.22. *$\Phi_{\mathcal{LDL}^+}$ is polynomial rewritable. Furthermore, $\Phi_{\mathcal{LDL}^+}$ is modular.*

The next result is the main result of this section, and shows that $\Phi_{\mathcal{LDL}^+}(\Sigma)$ works as desired.

Proposition 2.2.23. *For every (normalized) $\mathcal{LDL}^+$ KB Σ, $Q \in clos(\Sigma)$, and $\mathbf{o} \subseteq \Delta_{\mathcal{H}(\Sigma)}$, it holds that $\Sigma \models Q(\mathbf{o})$ iff $\Phi_{\mathcal{LDL}^+}(\Sigma) \models Q(\mathbf{o})$.*

Proof. We only prove the case Q is a concept C, as the case Q is a role can be proved similarly. Due to Proposition 2.2.21, it suffices to show that $MM(\Delta_{\mathcal{H}(\Sigma)}, \Sigma) \models C(o)$ iff $\Phi(\Sigma) \models C(o)$. Define $MM \triangleq MM(\Delta_{\mathcal{H}(\Sigma)}, \Sigma)$.

($\Rightarrow$) Assume $MM \models C(o)$. We define an interpretation M for $\Phi(\Sigma)$ as follows:

$$M \triangleq \{D(a) \mid D \in clos(\Sigma), MM \models D(a)\} \cup \{E(a,b) \mid E \in clos(\Sigma), MM \models E(a,b)\}$$

Clearly, $M \models C(o)$, such that it remains to show that M is a minimal model of $gr(\Phi(\Sigma))$.

1. *M is a model of $gr(\Phi(\Sigma))$.* We check satisfiability of the rules in $gr(\Phi(\Sigma))$.
   - Take a rule $H(a) \leftarrow B(a)$ originating from (2.4) such that $B(a) \in M$. Then, by definition of M, $MM \models B(a)$. Since the rule (2.4) was introduced by an axiom $B \sqsubseteq H \in \Sigma$ and MM is a model of Σ, we have that $MM \models H(a)$ and thus $H(a) \in M$.
   - Take a rule $A(b) \leftarrow B(a), E(a,b)$ originating from (2.5) such that $B(a), E(a,b) \in M$. Then, $MM \models B(a)$ and $MM \models E(a,b)$. Since the rule (2.5) originates from an axiom $B \sqsubseteq \forall E.A$, we have that $MM \models (\forall E.A)(a)$, and thus by definition of a value restriction, $MM \models A(b)$ such that $A(b) \in M$.
   - Rules originating from (2.6) can be done similarly.
   - For a rule $P(a,b) \leftarrow P^-(b,a)$ originating from (2.7) with $P^-(b,a) \in M$, we have that $MM \models P^-(b,a)$ such that $MM \models P(a,b)$ and thus $P(a,b) \in M$.
   - Since $MM \models \top(o_1)$ and $MM \models \top^2(o_1, o_2)$, rules (2.8) and (2.9) are satisfied as well.
   - For a rule (2.10), we have that $MM \models \{o\}(o)$ such that $\{o\}(o) \in M$.
   - For a rule $D(a) \leftarrow D_1(a), D_2(a)$ originating from (2.11) with $D_1(a), D_2(a) \in M$, we have $MM \models D_1(a)$ and $MM \models D_2(a)$ such that $MM \models (D_1 \sqcap D_2)(a)$ and thus $(D_1 \sqcap D_2)(a) \in M$ where $D = D_1 \sqcap D_2$.
   - All remaining rules in $gr(\Phi(\Sigma))$ can be verified similarly.
2. *M is a minimal model of $gr(\Phi(\Sigma))$.* Assume it is not, then there is a model $N \subset M$ of $gr(\Phi(\Sigma))$. Define a set-interpretation NN for Σ:

$$\begin{aligned} NN \triangleq \{&A(a) \mid A(a) \in N, A \text{ concept name}\} \\ &\cup \{P(a,b) \mid P(a,b) \in N, P \text{ role name}\} \\ &\cup \{\top(a_1), \top^2(a_1,a_2) \mid a_1, a_2 \in \Delta_{\mathcal{H}(\Sigma)}\} \\ &\cup \{\{o_1\}(o_1), \{(o_1,o_2)\}(o_1,o_2) \mid o_1, o_2 \in \Delta_{\mathcal{H}(\Sigma)}\} \end{aligned}$$

Clearly, NN is a set-interpretation for Σ over $\Delta_{\mathcal{H}(\Sigma)}$ the Herbrand domain of Σ (and of $\Phi(\Sigma)$). Moreover, one can show — using that N is a model of $gr(\Phi(\Sigma))$ and by induction — that for any b-concept expression D' and for any b-role expression E'

$$NN \models D'(a) \Rightarrow D'(a) \in N \tag{2.21}$$

and

$$NN \models E'(a,b) \Rightarrow E'(a,b) \in N \tag{2.22}$$

We show that $NN \subset MM$ and that NN is a model of Σ, contradicting the minimality of MM, such that M is indeed a minimal model of $gr(\Phi(\Sigma))$.

(a) $NN \subset MM$. Note that both have the same domain $\Delta_{\mathcal{H}(\Sigma)}$ and thus NN and MM are equal on

$$\{\top(a_1), \top^2(a_1,a_2) \mid a_1, a_2 \in \Delta_{\mathcal{H}(\Sigma)}\} \cup \{\{o_1\}(o_1), \{(o_1,o_2)\}(o_1,o_2) \mid o_1, o_2 \in \Delta_{\mathcal{H}(\Sigma)}\}$$

We show first that $NN \subseteq MM$. Take a $A(a) \in NN$, then $A(a) \in N \subset M$ such that, by definition of M, $MM \models A(a)$, i.e., $A(a) \in MM$ (MM seen as a set-interpretation); and similarly for a $P(a,b) \in NN$.

Since $N \subset M$ there is a $D(a) \in M - N$ or a $E(a,b) \in M - N$. Assume $D(a) \in M - N$. Then, $MM \models D(a)$ and by (2.21) $NN \not\models D(a)$. Since D is an b-concept expression (M only contains b-concept expressions) it is monotonic (Proposition 2.2.14) and thus $MM \not\subseteq NN$. Together with $NN \subseteq MM$, we have that $NN \subset MM$.

(b) NN is a model of Σ. One can prove this using that N is a model of $gr(\Phi(\Sigma))$ together with (2.21) and (2.22). For example, for an axiom $B \sqsubseteq A$ with concept name A and $NN \models B(a)$, we have that $(A(a) \leftarrow B(a)) \in gr(\Phi(\Sigma))$ and, via (2.21) that $B(a) \in N$. Such that, since N is a model of $gr(\Phi(\Sigma))$, $A(a) \in N$. For concept names A, we then have that $A(a) \in NN$ and thus $NN \models A(a)$.

($\Leftarrow$) The other direction can be done similarly. □

Corollary 2.2.24. *$\mathcal{LDL}^+$ is (polynomial)* DATALOG*-rewritable.*

Proof. Indeed, take $\Phi_{\mathcal{LDL}^+} = \Phi$ as in Proposition 2.2.23. □

Thus, using Theorem 2.2.1, reasoning with dl-programs over $\mathcal{LDL}^+$ reduces to reasoning with DATALOG$^\neg$ under well-founded semantics.

Example 2.2.25. *Take the $\mathcal{LDL}^+$ KB Σ from Example 2.2.2. Then, the reduction yields the* DATALOG *program $\Phi_{\mathcal{LDL}^+}(\Sigma)$:*

$$
\begin{aligned}
Over(X) &\leftarrow (\geqslant 2PaptoRev.\top)(X)\\
(\geqslant 2PaptoRev.\top)(X) &\leftarrow PaptoRev(X, Y_1), \top(Y_1)\\
&\quad\ PaptoRev(X, Y_2), \top(Y_2), Y_1 \neq Y_2\\
Over(Y) &\leftarrow Super^+(X, Y), Over(X)\\
Super^+(X, Y) &\leftarrow Super(X, Y)\\
Super^+(X, Y) &\leftarrow Super(X, Z), Super^+(Z, Y)
\end{aligned}
$$

$$
\begin{aligned}
Super(X, Y) &\leftarrow \{(a, b)\}(X, Y)\\
Super(X, Y) &\leftarrow \{(b, c)\}(X, Y)\\
\{(a, b)\}(a, b) &\leftarrow\\
\{(b, c)\}(b, c) &\leftarrow\\
Super(X, Y) &\leftarrow Super^-(Y, X)\\
Pap2Rev(X, Y) &\leftarrow Pap2Rev^-(Y, X)\\
\top^2(X, Y) &\leftarrow \top(X), \top(Y)
\end{aligned}
$$

and in addition the rules for $\top$. For $KB = (\Phi, P)$ in Example 2.2.3, we then can easily construct $\Psi(KB)$.

Reductions of DLs to LP have been considered before, e.g., in [39, 54]. Swift [54] reduces reasoning in the DL $\mathcal{ALCQI}$ (in fact, consistency checking of concept expressions) to DATALOG$^\neg$ under answer set semantics (employing a guess and check methodology), while Hustadt et al. [39] reduce reasoning in the DL $\mathcal{SHIQ}^-$ to disjunctive DATALOG in a non-modular way, i.e., the translation as such is not usable in the context of dl-programs; both DLs considered in [39] and [54] do not feature transitive closure.

From the complexity of DATALOG, we obtain by DATALOG-rewritability of $\mathcal{LDL}^+$ immediately that it is tractable under data complexity. Moreover, due to the structure of $\Phi_{\mathcal{LDL}^+}(\Sigma)$, the same holds under combined complexity.

Corollary 2.2.26. *For every $\mathcal{LDL}^+$ KB Σ, concept name A, and $o \in \Delta_{\mathcal{H}(\Sigma)}$, deciding $\Sigma \models A(o)$ is in* PTIME *under both data and combined complexity.*

Indeed, all rules in $\Phi_{\mathcal{LDL}^+}(\Sigma)$ except (2.14) can be grounded in polynomial time (they use only constantly many variables). The rule (2.14) can be partially grounded for all values of X; whether the body of such a partially grounded rule can be satisfied in a given set of ground atoms is easily decided in polynomial time; hence, we can compute $MM(\Phi_{\mathcal{LDL}^+}(\Sigma))$ by simple fixpoint iteration in polynomial time.

We will establish matching lower complexity bounds below.

2.2.6 The OWL 2 Profiles

In this section, we review the OWL 2 Profiles [43], which are fragments of OWL 2 [44] that can be more efficiently evaluated than OWL 2, and discuss their relation with $\mathcal{LDL}^+$.

OWL 2 EL. OWL 2 EL corresponds to the DL $\mathcal{EL}^{++}$ [5, 6]. We consider the definition of $\mathcal{EL}^{++}$ in [6], which extends [5], in particular its *normal form for TBoxes*. The only concept expressions that $\mathcal{EL}^{++}$ allows that we did not introduce in $\mathcal{LDL}^{+}$ are $\bot$ (*bottom*) and so-called concrete domains. Given $(C_i)_{i\in N}, D$ either the top concept, concept names, nominals or concrete domain concepts, where D can additionally be $\bot$, an $\mathcal{EL}^{++}$ TBox contains the following axioms:

1. All terminological axioms have the following form: $C_1 \sqcap \ldots \sqcap C_n \sqsubseteq D$, $C_1 \sqsubseteq \exists R.C_2$, and $\exists R.C_1 \sqsubseteq D$, for role names R,
2. Role inclusions are of the form $R_1 \sqsubseteq R$ or $R_1 \circ R_2 \sqsubseteq R$;
3. Range restrictions are of the form $\mathtt{ran}(R) \sqsubseteq A$ for concept names A and role names R.

Both the range restrictions (in 3.) and the role inclusions (in 2.) correspond to valid $\mathcal{LDL}^{+}$ axioms. Indeed, a range restriction $\mathtt{ran}(R) \sqsubseteq A$ can be written as a terminological $\mathcal{LDL}^{+}$ axiom $\top \sqsubseteq \forall R.A$. The only constructs that prevent (1.) from also being valid $\mathcal{LDL}^{+}$ axioms are the possible presence of $\bot$, concrete domains, and the exists restriction in the head of the axiom. Denote the DL $\mathcal{EL}^{++}$ without $\bot$, concrete domains, and exists restrictions in the head of terminological axioms with $\mathcal{EL}^{++}_{-}$. We assume range restrictions in $\mathcal{EL}^{++}_{-}$ are written in $\mathcal{LDL}^{+}$ syntax.

Proposition 2.2.27. *$\mathcal{EL}^{++}_{-}$ is a fragment of $\mathcal{LDL}^{+}$, i.e., each $\mathcal{EL}^{++}_{-}$ KB is an $\mathcal{LDL}^{+}$ KB, and thus polynomially* DATALOG*-rewritable.*

Even though $\mathcal{EL}^{++}$ is not a fragment of $\mathcal{LDL}^{+}$, in turn $\mathcal{LDL}^{+}$ contains many constructs that $\mathcal{EL}^{++}$ does not allow, e.g., qualified number restrictions, inverses, general sequences of roles, role conjunction, role disjunction, concept disjunction in axiom bodies. For example, in $\mathcal{EL}^{++}$, one can only express so-called *enumerations involving a single individual* [43] while in $\mathcal{LDL}^{+}$ we can represent enumerations of several individuals using the nominal concept and concept disjunction.

OWL 2 QL. OWL 2 QL corresponds to the *DL-Lite* family $\mathit{DL\text{-}Lite}_{\mathit{core}}$, $\mathit{DL\text{-}Lite}_{\mathcal{R}}$, and $\mathit{DL\text{-}Lite}_{\mathcal{F}}$ [11]. Concept expressions in $\mathit{DL\text{-}Lite}_{\mathit{core}}$ are defined by:

$$\begin{array}{lcl|l|l} B & \longrightarrow & A & \exists R & \qquad \mathrm{R} \longrightarrow P \mid P^{-} \\ C & \longrightarrow & B & \neg B & \qquad \mathrm{E} \longrightarrow R \mid \neg R \end{array}$$

where A are concept names and P are role names. Terminological axioms are then of the form $B \sqsubseteq C$, with B and C defined as above. A *DL-Lite* ABox (also for the other variants than $\mathit{DL\text{-}Lite}_{\mathit{core}}$) is a set of assertions $A(a)$ and $R(a, b)$ for individuals a, b. The language $\mathit{DL\text{-}Lite}_{\mathcal{R}}$ adds role axioms $R \sqsubseteq E$ additionally, with R, E as above. Denote by $\mathit{DL\text{-}Lite}^{-}_{X}$ the DL $\mathit{DL\text{-}Lite}_{X}$ without negation and exists restrictions in axiom right-hand sides, $X \in \{\mathit{core}, \mathcal{R}, \mathcal{F}\}$. Then, both terminological and role axioms in $\mathit{DL\text{-}Lite}^{-}_{\mathcal{R}}$

are $\mathcal{LDL}^+$ axioms; and, any $DL\text{-}Lite_{\mathcal{R}}^-$ ABox can be rewritten using the nominals of $\mathcal{LDL}^+$ as usual.

Proposition 2.2.28. *The DLs $DL\text{-}Lite_{core}^-$ and $DL\text{-}Lite_{\mathcal{R}}^-$ are fragments of $\mathcal{LDL}^+$, and thus polynomially* DATALOG*-rewritable.*

Similar to $\mathcal{EL}^{++}$, full $DL\text{-}Lite_{core}$ and $DL\text{-}Lite_{\mathcal{R}}$ are not fragments of $\mathcal{LDL}^+$, but in turn $\mathcal{LDL}^+$ has constructs which none of the DLs $DL\text{-}Lite_X$ allows, e.g., role sequences.

The DL $DL\text{-}Lite_{\mathcal{F}}^-$, however, is not a fragment of $\mathcal{LDL}^+$. Indeed, like $DL\text{-}Lite_{\mathcal{F}}$ it allows for functional restrictions on roles, something that is not expressible in DATALOG as such.

OWL 2 RL. OWL 2 RL extends so-called *Description Logic Programs* [27]. The latter have a classical model semantics and correspond to the restriction of $\mathcal{LDL}^+$ to conjunction and disjunction of concepts, exists restrictions, and value restrictions. Thus, Description Logic Programs are a strict subset of $\mathcal{LDL}^+$, missing, e.g., nominals, qualified number restrictions, and role constructors.

Proposition 2.2.29. *Description Logic Programs are a fragment of $\mathcal{LDL}^+$, and thus polynomially* DATALOG*-rewritable.*

The full OWL 2 RL profile supports some features which are not in $\mathcal{LDL}^+$now, such as datatypes and negations in the head. Such constructors can be added to $\mathcal{LDL}^+$easily, thus $\mathcal{LDL}^+$will be a strict superset of OWL 2 RL.

Note that the translation of the transitive closure of a role expression E^+ results in the recursive rules (2.20) such that, in contrast with Description Logic Programs, the transformation $\Phi_{\mathcal{LDL}^+}$is not a first-order rewriting, justifying the term DATALOG*-rewritable*. Although DLs with expressive role constructs such as role sequence, role disjunction and transitive closure tend to become undecidable (e.g., $\mathcal{ALC}^+\mathcal{N}(\circ, \sqcup)$ [7]), $\mathcal{LDL}^+$ remains decidable. Moreover, it has an Herbrand domain model property (a finite model property where the domain is the Herbrand domain). Indeed, from [7] one can see that the undecidability proofs for expressive DLs extensively use functional restrictions on roles, a feature $\mathcal{LDL}^+$ cannot express.

Checking ground entailment in OWL 2 RL and Description Logic Programs is data and combined complete for PTIME [43]. As the latter are a fragment of $\mathcal{LDL}^+$ without number restrictions, combined with Corollary 2.2.26 we obtain the following result.

Proposition 2.2.30. *For any $\mathcal{LDL}^+$ KB Σ, concept name A, and $o \in \Delta_{\mathcal{H}(\Sigma)}$, deciding $\Sigma \models A(o)$ is data and combined complete for* PTIME*. The hardness holds in absence of number restrictions.*

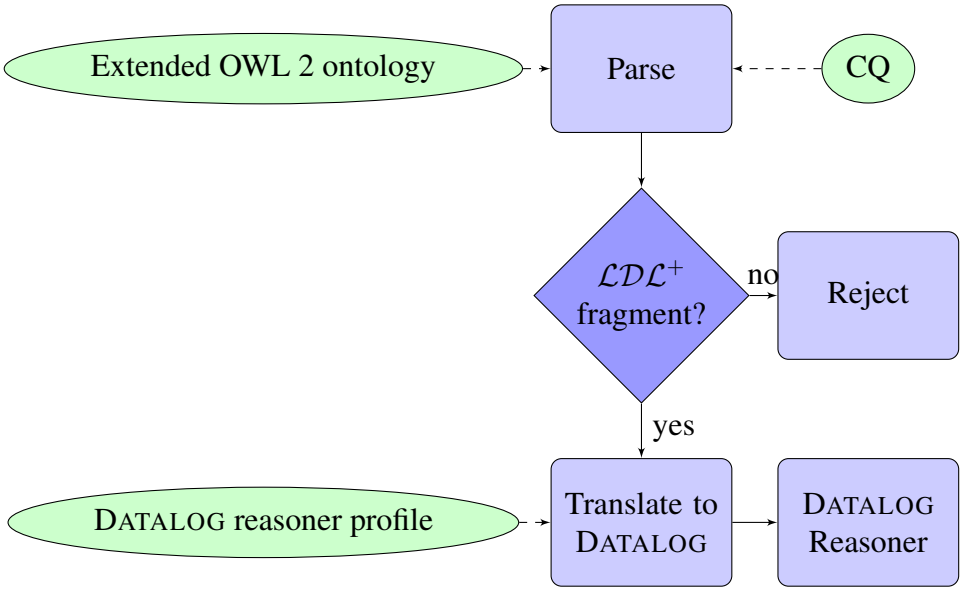


Figure 2.1: DReW Control Flow — DL Component

2.2.7 Implementation and Evaluation

2.2.7.1 Implementation

Based on the concept of DATALOG-rewriting, we developed a new reasoner DReW (DATALOG ReWriter)[4], which rewrites $\mathcal{LDL}^+$ ontologies (dl-programs over $\mathcal{LDL}^+$ ontologies) to DATALOG (DATALOG$^\neg$) programs, and calls an underlying rule-based reasoner, currently DLV, to perform the actual reasoning. For $\mathcal{LDL}^+$ ontologies, DReW does instance checking as well as answering of conjunctive queries (CQs). For dl-programs over $\mathcal{LDL}^+$ ontologies, DReW computes the well-founded model.

Figure 2.1 shows a schematic overview of the component of DReW responsible for reducing entailment from DLs to DATALOG. The extension for dl-programs is a straightforward elaboration of this. Taking as input a conjunctive query and an ontology in OWL 2 syntax extended for the complex role expressions of $\mathcal{LDL}^+$, DReW checks whether the ontology is in the $\mathcal{LDL}^+$ fragment. If it is, we translate the ontology according to the format suitable for the specified DATALOG reasoner (DLV in our case).

DReW is written in Java using an extension of the OWL API 3.0.0[5] for parsing $\mathcal{LDL}^+$ ontologies. The underlying DATALOG engine we used is the latest version of DLV (*dl-magic-snapshot-2009-11-26*) [6] which supports magic sets and well-founded semantics.

[4] http://www.kr.tuwien.ac.at/research/systems/drew
[5] http://owlapi.sourceforge.net/
[6] http://www.dbai.tuwien.ac.at/proj/dlv/magic/

Ontology	Axioms	Inds	Concepts	Object Props	$\mathcal{LDL}^+$?	Violated Axs	Violated %
Galen	4,356	0	2,747	261	no	1,881	0.43
Dolce	1,185	2	125	251	no	162	0.14
Wine	773	162	142	13	no	137	0.18
Vicodi	53,876	16,942	194	10	yes	0	0
Semintec	65,459	17,941	60	16	no	113	$1.73 \cdot 10^{-3}$
LUBM	8,612	1,555	43	25	no	8	$9.29 \cdot 10^{-4}$

Table 2.2: $\mathcal{LDL}^+$profile checking

2.2.7.2 Evaluation

In this section, we evaluate the DReW reasoner. We do so along two axes: as a pure Description Logic reasoner and as a reasoner for dl-programs.

All experiments were performed on a laptop running Ubuntu 10.04 with a 1.83G CPU and 2G of memory; the memory of the Java Virtual Machine was set to 1G.

2.2.7.2.1 Reasoning with Description Logics We first analyze to what extent common ontologies fall in the $\mathcal{LDL}^+$ fragment. Next, we analyze the performance of conjunctive query answering with DReW compared to standard DL reasoners on those ontologies.

Expressiveness of $\mathcal{LDL}^+$

To assess the expressiveness of $\mathcal{LDL}^+$, we select several ontologies and show that they fall to a large extent in the $\mathcal{LDL}^+$profile. We picked the ontologies that are used in Motik's thesis for testing the DL reasoner KAON2 [46]; they can be downloaded from the KAON 2 site[7].

The results of this experiment are listed in Table 2.2. Note that for Galen over 40% of the axioms are not in the $\mathcal{LDL}^+$ profile, but that for Dolce and Wine over 80% of axioms are in $\mathcal{LDL}^+$. Only Vicodi is fully in $\mathcal{LDL}^+$and over 99% of Semintec and LUBM axioms are in $\mathcal{LDL}^+$. Most of the violations are due to existential quantifiers occurring on the right side of axioms.

Conjunctive Query Evaluation

To evaluate the performance of CQs over ontologies using DReW, we compare it with 3 state-of-the-art DL reasoners: KAON2, RacerPro, and Pellet. We did not consider other

[7] `http://kaon2.semanticweb.org/download/test_ontologies.zip`

DL reasoners, such as HermiT or Fact++ as they cannot handle CQs.

Reasoning in KAON2[8] [46] is implemented using novel algorithms that reduce a $\mathcal{SHIQ}(\mathbf{D})$ knowledge base to a disjunctive DATALOG program based on resolution techniques. Pellet[9] [51] fully supports OWL 2[44]. In contrast with KAON2, it is a reasoner based on tableaux algorithms. RacerPro[10] [29] is a tableaux-based reasoner as well and implements the Description Logic $\mathcal{SHIQ}$. All 3 reasoners support conjunctive query answering.

We specifically tested CQs on The Lehigh University Benchmark (LUBM) [28]. LUBM is developed to facilitate the evaluation of Semantic Web repositories in a standard and systematic way. The benchmark is intended to evaluate the performance of those repositories with respect to extensional queries over a large data set that commits to a single realistic ontology. It consists of a university domain ontology, customizable and repeatable synthetic data, a set of test queries, and several performance metrics. The queries we evaluated are as in the LUBM query page [11], referring to numbers 1-14.

As we indicated in Table 2.2, LUBM is not fully in $\mathcal{LDL}^+$: there are 8 violated axioms, e.g.,

$$Person \sqcap \exists headOf.Department \equiv Chair \ .$$

For our experiments and to have an $\mathcal{LDL}^+$ conformant fragment of LUBM, we replace such equivalence axioms by subsumption axioms, e.g., by

$$Person \sqcap \exists headOf.Department \sqsubseteq Chair \ .$$

In general, such a conversion changes the semantics of the ontology. However, in our considered test of the benchmark queries, the query results are exactly the same as on the original LUBM. It is part of future research to investigate how DReW can deal with partial $\mathcal{LDL}^+$ ontologies in answering queries as faithfully as possible.

The results of evaluating the 14 CQs on LUBM are shown in Table 2.3. From the table, we see that DReW outperforms RacerPro and Pellet in all the queries and that it is slightly better than KAON2 for most of the queries. Note the out-of-the-normal times for RacerPro on query 4 and query 8; we assume they are caused by the use of data properties.

As DReW and KAON2 have evaluation times close to each other, we also evaluate CQs on LUBM ontologies with a different numbers of individuals. The result is summarized in Table 2.4.

LUBM1 is the original LUBM ontology. By removing and adding individuals, we get LUBM0 and LUBM2. The number under each reasoner is the average time for answering the 14 queries. In all the LUBMs, DReW is better than RacerPro and Pellet. However, compared with KAON2, we also see that DReW is not so good at dealing with large number of individuals. We assume that the reason is the use of DLV as the underlying

[8] http://kaon2.semanticweb.org/
[9] http://clarkparsia.com/pellet/
[10] http://www.racer-systems.com
[11] http://swat.cse.lehigh.edu/projects/lubm/query.htm

Query	DReW	KAON2	RacerPro	Pellet
1	3.13	2.84	3.78	4.55
2	2.23	2.39	4.24	4.54
3	2.29	2.35	3.68	4.54
4	2.25	2.61	26.05	4.63
5	2.29	2.60	5.12	4.52
6	2.24	2.56	5.05	4.51
7	2.21	2.63	3.39	4.44
8	2.28	2.65	27.13	4.62
9	2.22	2.67	4.80	4.54
10	2.22	2.42	3.85	4.53
11	2.23	2.31	4.39	4.49
12	2.27	2.55	4.08	4.63
13	2.31	2.58	4.44	4.42
14	2.26	2.35	5.30	4.52

Table 2.3: Conjunctive Queries on LUBM (in secs.)

Ontology	Inds	DReW	KAON2	RacerPro	Pellet
LUBM0	904	1.61	2.27	4.51	3.53
LUBM1	1,555	2.27	2.54	7.52	4.53
LUBM2	2,753	5.07	3.72	9.38	7.57

Table 2.4: Conjunctive Queries on LUBM with Different Number of Individuals

DATALOG engine. Since there is no public API for DLV, we have to use it as a standalone process. When the translated ontology is big, the communication of processes costs significant time.

2.2.7.2.2 Reasoning with DL-Programs DReW is designed for reasoning over dl-programs under well-founded semantics. The only reasoner available for comparison is DLVHEX [12] [17]. DLVHEX is a prototype implementation for computing the stable models of so-called HEX-programs – an extension of dl-programs for reasoning with external sources (not necessarily DL knowledge bases) under the answer set semantics. By using the *Description Logic Plugin* [13], which interfaces to OWL ontologies via a Description Logic reasoner (currently RacerPro), DLVHEX can reason on dl-programs under the answer set semantics.

Note that for DATALOG programs (i.e., without negation), the well-founded semantics

[12] http://www.kr.tuwien.ac.at/research/systems/dlvhex

[13] http://www.kr.tuwien.ac.at/research/systems/dlvhex/dlplugin.html

coincides with the answer set semantics. We thus evaluate both reasoners on LUBM, which is negation free. We wrote several dl-programs to evaluate CQs over dl-programs.

All the test results are shown in Table 2.5. We see that DReW outperforms DLVHEX for all the tests. As the number of dl-atoms increases, the advantage of DReW becomes more clear, confirming our hypothesis that translating dl-programs to DATALOG programs reduces the overload of calling external DL reasoners as is the case in DLVHEX.

Query	DReW	DLVHEX+DL-Plugin	dl-atoms	Factor
0	2.81	4.31	1	1.53
1	2.63	3.04	1	1.16
2	2.60	3.88	1	1.49
3	2.59	4.04	1	1.56
4	2.75	3.51	1	1.27
5	3.00	5.10	1	1.70
6	4.69	19.59	6	4.17
7	3.20	8.38	2	2.62

Table 2.5: Reasoning on dl-programs

2.2.8 Conclusion

We defined the class of DATALOG-rewritable DLs and showed that reasoning with dl-programs over such DLs can be reduced to DATALOG$^\neg$ under well-founded semantics. This reduction avoids the overhead that is normally associated with the querying of a native DL reasoner. The transformation is applicable to a range of different DLs, including $\mathcal{LDL}^+$, a novel rich DL, as well as to large fragments of the OWL 2 Profiles that have been designed for tractable DL reasoning. In particular, the transformation of a negation-free (stratified) dl-program results in a DATALOG (stratified DATALOG$^\neg$) program. In this way, we obtain tractable reasoning with recursion and negation, which thanks to the availability of efficient engines for well-founded semantics (e.g., DLV, and the XSB system) provides a basis for developing efficient and scalable applications that combine rules and ontologies.

Looking at the OWL 2 profile OWL 2 QL $\mathcal{LDL}^+$ misses *negation*. Negation is not realizable in DATALOG; it remains to be seen whether for DATALOG$^\neg$ under well-founded semantics, transformations similar to the one we presented (with possibly restricted negation in DLs) are feasible.

We developed the reasoner DReW that uses the DATALOG-rewriting technique. DReW can answer conjunctive queries over $\mathcal{LDL}^+$ ontologies, as well as reason on dl-programs over $\mathcal{LDL}^+$ ontologies under well-founded semantics. The preliminary but encouraging experimental results show that DReW can efficiently handle large knowledge bases. Our

results enable the use of mature LP technology, e.g., systems like XSB or DATALOG engines like DLV, and emerging implementations of recursive SQL, to reason efficiently with dl-programs involving recursion and negation, as a basis for advanced applications.

Finally, DATALOG-rewritability is not just useful for (1) DL reasoning via DATALOG engines or (2) loosely-coupled reasoning via dl-programs, but also for tight-coupling approaches such as *r-hybrid KBs* [49]. Intuitively, while rules in r-hybrid KBs must be *DL-safe* to ensure that only the Herbrand domain is relevant, our approach hints that it is also interesting to look at DLs that have this property. Note that they are of particular interest for data management, where often just the Herbrand domain matters.

2.3 Optimizations for Tableaux Algorithms for F-Hybrid Knowledge Bases

As already mentioned in the introduction of this section, Datalog-rewritable DLs typically do not allow for the *exists restriction* on the right-hand side of General Inclusion Axioms, as this feature can enforce the introduction of new domain elements (beyond the Herbrand universe). However, such a feature is sometimes needed: see, for example, the discussion concerning the Steel Industry Use Case in the Appendix. One way to deal with this feature is by dropping the restriction of the domain to the Herbrand universe in a logic programming language: this is the principle which underlies OASP, a formalism which extends the Answer Set Programming language by preserving the syntax of the original language, but introducing an open domain semantics, like it is common in the DL world. Programs are interpreted w.r.t. open domains, i.e., non-empty arbitrary domains which extend the Herbrand universe.

Example 2.3.1. *Consider the following program:*

$$\begin{aligned} fail(X) &\leftarrow not\ pass(X) \\ pass(john) &\leftarrow \end{aligned}$$

Although the predicate $fail$ is not satisfiable under the ordinary answer set semantics – the only answer set being $\{pass(john)\}$ – it is satisfiable under the open answer set semantics. If one considers, for example, the universe $\{john, x\}$, with x some individual which does not belong to the Herbrand universe, there is an open answer set $\{pass(john), fail(x)\}$ which satisfies $fail$.

OASP is generally undecidable. Several decidable fragments of OASP were identified by syntactically restricting the shape of logic programs, while carefully safe-guarding enough expressiveness for integrating rule- and ontology-based knowledge. A notable fragment is that of *Forest Logic Programs (FoLPs)* that are able to simulate reasoning in the DL $\mathcal{SHOQ}$. FoLPs allow for the presence of only unary and binary predicates in rules which have a tree-like structure. A sound and complete algorithm for satisfiability

checking of unary predicates w.r.t. FoLPs has been described in deliverable D3.2 [35]. The algorithm exploits the forest model property of the fragment: if a unary predicate is satisfiable, than it is satisfied by a forest-shaped model, with the predicate checked to be satisfiable being in the label of the root of one of the trees composing the forest. It is essentially a tableau-based procedure which builds such a forest model in a top-down fashion.

In this section we present an optimization for the tableau algorithm for reasoning with FoLPs achieved by means of a knowledge compilation technique. So-called unit completion structures, which are possible building blocks of a forest model, in the form of trees of depth 1, are computed in an initial step of the algorithm. This is done by using the original algorithm. Repeated computations are avoided by using these structures in a pattern-matching style when constructing a model. In general, not all unit completion structures have to be considered: inherent redundancy in a FoLP, like rules which are less general than others gives rise to redundancy among completion structures. A unit completion structure is redundant iff there is another simpler (less constrained) unit completion structure. The latter can replace the former in any forest model. We formalize this notion, making it possible to identify such redundant structures and discard them.

We will start with some preliminaries in Section 2.3.1, like the OASP semantics and some notation. Section 2.3.2 recalls the FoLP and the f-hybrid knowledge bases fragments, while Section 2.3.3 gives an overview of the original algorithm for reasoning with FoLPs. The new algorithm for reasoning with FoLPs is described in Section 2.4.1. Finally, Section 2.4.2 draws some conclusions.

2.3.1 Preliminaries

We recall the open answer set semantics [34]. *Constants* $a, b, c, \ldots$, *variables* $X, Y, \ldots$, *terms* $s, t, \ldots$, and *atoms* $p(t_1, \ldots, t_n)$ are as usual. A *literal* is an atom L or a negated atom $not\ L$. We allow for *inequality literals* of the form $s \neq t$, where s and t are terms. A literal that is not an inequality literal will be called a *regular literal*. For a set S of literals or (possibly negated) predicates, $S^+ = \{a \mid a \in S\}$ and $S^- = \{a \mid not\ a \in S\}$. For a set S of atoms, $not\ S = \{not\ a \mid a \in S\}$. For a set of (possibly negated) unary predicates S: $S(X) = \{a(X) \mid a \in S\}$, and for a set of (possibly negated) binary predicates S: $S(X, Y) = \{a(X, Y) \mid a \in S\}$. For a predicate p, $\pm p$ denotes p or $not\ p$, whereby multiple occurrences of $\pm p$ in the same context will refer to the same symbol (either p or $not\ p$).

A *program* is a countable set of rules $\alpha \leftarrow \beta$, where α is a finite set of regular literals and β is a finite set of literals. The set α is the *head* and represents a disjunction, while β is the *body* and represents a conjunction. If $\alpha = \emptyset$, the rule is called a *constraint*. A special type of rules with empty bodies, are so-called *free rules* which are rules of the form: $q(t_1, \ldots, t_n) \vee not\ q(t_1, \ldots, t_n) \leftarrow$, for terms $t_1, \ldots, t_n$; these kind of rules enable a choice for the inclusion of atoms in the open answer sets. We call a predicate

q *free* if there is a $q(X_1, \ldots, X_n) \vee not\ q(X_1, \ldots, X_n) \leftarrow$, with variables $X_1, \ldots, X_n$. Atoms, literals, rules, and programs that do not contain variables are *ground*. For a rule or a program R, let $cts(R)$ be the constants in R, $vars(R)$ its variables, and $preds(R)$ its predicates with $upreds(R)$ the unary and $bpreds(R)$ the binary predicates. For every non-free predicate q and a program P, P_q is the set of rules of P that have q as a head predicate. A *universe* U for P is a non-empty countable superset of the constants in P: $cts(P) \subseteq U$. We call P_U the ground program obtained from P by substituting every variable in P by every element in U. Let HBP ($\mathcal{L}_P$) be the set of regular atoms (literals) that can be formed from a ground program P.

An *interpretation* I of a ground P is a subset of HBP. We write $I \models p(t_1, \ldots, t_n)$ if $p(t_1, \ldots, t_n) \in I$ and $I \models not\ p(t_1, \ldots, t_n)$ if $I \not\models p(t_1, \ldots, t_n)$. Also, for ground terms s, t, we write $I \models s \neq t$ if $s \neq t$. For a set of ground literals L, $I \models L$ if $I \models l$ for every $l \in L$. A ground rule $r : \alpha \leftarrow \beta$ is *satisfied* w.r.t. I, denoted $I \models r$, if $I \models l$ for some $l \in \alpha$ whenever $I \models \beta$. A ground constraint $\leftarrow \beta$ is satisfied w.r.t. I if $I \not\models \beta$.

For a positive ground program P, i.e., a program without not, an interpretation I of P is a *model* of P if I satisfies every rule in P; it is an *answer set* of P if it is a subset minimal model of P. For ground programs P containing not, the *GL-reduct* [24] w.r.t. I is defined as P^I, where P^I contains $\alpha^+ \leftarrow \beta^+$ for $\alpha \leftarrow \beta$ in P, $I \models not\ \beta^-$ and $I \models \alpha^-$. I is an *answer set* of a ground P if I is an answer set of P^I.

A program is assumed to be a finite set of rules; infinite programs only appear as byproducts of grounding with an infinite universe. An *open interpretation* of a program P is a pair (U, M) where U is a universe for P and M is an interpretation of P_U. An *open answer set* of P is an open interpretation (U, M) of P with M an answer set of P_U. An n-ary predicate p in P is *satisfiable* if there is an open answer set (U, M) of P s. t. $p(x_1, \ldots, x_n) \in M$, for some $x_1, \ldots, x_n \in U$.

We introduce notation for trees which extend those in [57]. Let $\cdot$ be a concatenation operator between sequences of constants or natural numbers. A *tree* T with root c (T_c), where c is a specially designated constant, has as nodes sequences of the form $c \cdot s$, where s is a (possibly empty) sequence of positive integers formed with the concatenation operator; for $x \cdot d \in T$, $d \in \mathbb{N}^*$, we have that $x \in T$. The set $A_T = \{(x, y) \mid x, y \in T, \exists n \in \mathbb{N}^* : y = x \cdot n\}$ is the set of arcs of a tree T. For $x, y \in T$, we say that $x <_T y$ iff x is a prefix of y and $x \neq y$.

A *forest* F is a set of trees $\{T_c \mid c \in C\}$, where C is a set of distinguished constants. We denote with $N_F = \cup_{T \in F} T$ and $A_F = \cup_{T \in F} A_T$ the set of nodes and the set of arcs of a forest F, respectively. Let $<_F$ be a strict partial order relationship on the set of nodes N_F of a forest F where $x <_F y$ iff $x <_T y$ for some tree T in F. An extended forest EF is a tuple (F, ES) where $F = \{T_c \mid c \in C\}$ is a forest and $ES \subseteq N_F \times C$. We denote by $N_{EF} = N_F$ the nodes of EF and by $A_{EF} = A_F \cup ES$ its arcs. So unlike a normal forest, an extended forest can have arcs from any of its nodes to any root of some tree in the forest.

Finally, for a directed graph G, $paths_G$ is the set of pairs of nodes for which there exists

a path in G from the first node in the pair to the second one.

2.3.2 Forest Logic Programs

Forest Logic Programs (FoLPs) [33] are logic programs with tree-shaped rules which allow for constants and for which satisfiability checking under the open answer set semantics is decidable.

Definition 2.3.2. *A* forest logic program (FoLP) *is a program with only unary and binary predicates, and such that a rule is either:*

1. *a* free rule
$$a(s) \vee not\ a(s) \leftarrow$$
or
$$f(s,t) \vee not\ f(s,t) \leftarrow ,$$
where s and t are terms such that if s and t are both variables, they are different,

2. *a unary rule*
$$a(s) \leftarrow \beta(s), (\gamma_m(s,t_m), \delta_m(t_m))_{1 \leqslant m \leqslant k}, \psi$$
where s and t_m, $1 \leqslant m \leqslant k$, are terms (again, if both s and t_m are variables, they are different; similarly for t_i and t_j), where
   - $\psi \subseteq \bigcup_{1 \leqslant i \neq j \leqslant k} \{t_i \neq t_j\}$ *and* $\{\neq\} \cap \gamma_m = \emptyset$ *for* $1 \leqslant m \leqslant k$,
   - $\forall t_i \in vars(r) : \gamma_i^+ \neq \emptyset$, *i.e., for variables t_i there is a positive atom that connects s and t_i,*

3. *a* binary rule
$$f(s,t) \leftarrow \beta(s), \gamma(s,t), \delta(t)$$
with $\{\neq\} \cap \gamma = \emptyset$ and $\gamma^+ \neq \emptyset$ if t is a variable (s and t are different if both are variables), or

4. *a constraint*
$$\leftarrow a(s)$$
or
$$\leftarrow f(s,t),$$
where s and t are different if both are variables).

The following program P is a FoLP which says that an individual is a special member of an organization (*smember*) if it has the support of another special member: *rule* r_1, or if it has the support of two regular members of the organization (*rmember*): *rule* r_2. The binary predicate *support* which describes the 'has support' relationship is free. No individual can be at the same time both a special member and a regular member: *constraint* r_4. Two particular regular members are a and b: *facts* r_5 and r_6.

$$\{rmember\}a \xleftarrow[\{support\}]{\{smember\}} x \xrightarrow[\{support\}]{} b\,\{rmember\}$$

Figure 2.2: A Forest Model for P

Example 2.3.3.

$$\begin{array}{lrcl}
r_1: & smember(X) & \leftarrow & support(X,Y), smember(Y) \\
r_2: & smember(X) & \leftarrow & support(X,Y), rmember(Y), \\
 & & & support(X,Z), rmember(Z), Y \neq Z \\
r_3: & support(X,Y) \vee not\ support(X,Y) & \leftarrow & \\
r_4: & & \leftarrow & smember(X), rmember(X) \\
r_5: & rmember(a) & \leftarrow & \\
r_6: & rmember(b) & \leftarrow &
\end{array}$$

As their name suggests FoLPs have the *forest model property*:

Definition 2.3.4. *Let P be a FoLP. A predicate $p \in upreds(P)$ is* forest satisfiable *w.r.t. P if there is an open answer set (U, M) of P and there is an extended forest $EF \equiv (\{T_\varepsilon\} \cup \{T_a \mid a \in cts(P)\}, ES)$, where ε is a constant, possibly one of the constants appearing in P , and a labeling function $\mathcal{L} : \{T_\varepsilon\} \cup \{T_a \mid a \in cts(P)\} \cup A_{EF} \longrightarrow 2^{preds(P)}$ s. t.*

- *$U = N_{EF}$, and*
- *$p \in \mathcal{L}(\varepsilon)$,*
- *for any $T \in EF$ and $i > 0$: $z \cdot i \in T$, iff there is some $f(z, z \cdot i) \in M$, $z \in T$, and*
- *for any $T \in EF$, $y \in T$, $q \in upreds(P)$, and $f \in bpreds(P)$, we have that*
  - *$q(y) \in M$ iff $q \in \mathcal{L}(y)$, and*
  - *$f(y, u) \in M$ iff $(u = y \cdot i \vee u \in cts(P)) \wedge f \in \mathcal{L}(y, u)$.*

We call such a (U, M) a forest model *and a program P has the* forest model property *if the following property holds: if $p \in upreds(P)$ is satisfiable w.r.t. P then p is forest satisfiable w.r.t. P.*

Consider the FoLP P introduced in Example 2.3.3. The unary predicate $smember$ is forest satisfiable w.r.t. P: $(\{a, b, x\}, \{rmember(a), rmember(b), support(x, a), support(x, b), smember(x)\})$ is a forest model in which $smember$ appears in the label of the (anonymous) root of one of the trees in the forest (see Figure 2.2). Note that in the ordinary LP setting, where one restricts the universe to the Herbrand universe, $smember$ is not satisfiable.

Finally, we also recall the definition and some complexity results concerning f-hybrid knowledge bases.

Definition 2.3.5. *An* f-hybrid knowledge base *is a pair* $\langle \Sigma, P \rangle$ *where* Σ *is a* $\mathcal{SHOQ}$ *knowledge base and* P *is a FoLP.*

Atoms and literals in P might have as the underlying predicate an atomic concept or role name from Σ, in which case they are called *DL atoms* and *DL literals* respectively. Additionally, there might be other predicate symbols available, but without loss of generality we assume they cannot coincide with complex concept or role descriptions.

Proposition 2.3.6. *Satisfiability checking w.r.t. f-hybrid knowledge bases is in* 2EXPTIME *in the size of the f-hybrid knowledge base.*

2.3.3 An Algorithm for Forest Logic Programs

In this section, we review the tableau algorithm for satisfiability checking for FoLPs introduced in deliverable D3.2. We use as a running example the FOLP from Example 2.3.3. Constraints are not treated explicitly in the algorithm as they can be simulated using unary rules. As such, the constraint

$$r_4 : \leftarrow smember(X), rmember(X)$$

in Example 2.3.3 is replaced with

$$r'_4 : co(X) \leftarrow not\ co(X), smember(X), rmember(X),$$

with co being a new predicate.

The basic data structure used by the algorithm to describe a forest model in construction is a so-called *completion structure*. Its main components are an extended forest EF, whose set of nodes constitutes the universe of the model, and a labeling function CT (*content*), which assigns to every node, resp. arc of EF, a set of possibly negated unary, resp. binary predicates. The presence of a predicate symbol $p/not\ p$ in the content of some node or arc x indicates the presence/absence of the atom $p(x)$ in the open answer set.

The presence (absence) of an atom in the open answer set is justified by imposing that the body of at least one ground rule which has the respective atom in the head is satisfied (no body of a rule which has the respective atom in the head is satisfied). In order to keep track which (possibly negated) predicate symbols in the content of some node or arc have already been expanded a so-called status function . is introduced. Furthermore, in order to ensure that no atom in the partially constructed open answer set is circularly motivated, i.e. the atoms are well-supported [18], a graph G which keeps track of dependencies between atoms in the (partial) model is maintained.

Definition 2.3.7. *An $\mathcal{A}_1$-completion structure for a FoLP P[14] is a tuple $\langle EF, \text{CT}, ., G\rangle$ where:*

- *$EF = \langle F, ES\rangle$ is an extended forest,*
- *$\text{CT} : N_{EF} \cup A_{EF} \longrightarrow 2^{preds(P) \cup not\ (preds(P))}$ is the 'content' function,*
- *$. : \{(x, \pm q) \mid \pm q \in \text{CT}(x), x \in N_{EF} \cup A_{EF}\} \longrightarrow \{exp, unexp\}$ is the 'status' function,*
- *$G = \langle V, A\rangle$ is a directed graph which has as vertices atoms in the answer set in construction: $V \subseteq HBP_{N_{EF}}$.*

An initial $\mathcal{A}_1$-completion structure *for checking satisfiability of a unary predicate p w.r.t. a FoLP P is a completion structure $\langle EF, \text{CT}, ., G\rangle$ with $EF = (F, \emptyset)$, $F = \{T_\varepsilon\} \cup \{T_a \mid a \in cts(P)\}$, where ε is a constant, possibly in $cts(P)$, $T_x = \{x\}$, for $x \in \{\varepsilon\} \cup cts(P)$, $G = \langle V, \emptyset\rangle$, $V = \{p(\varepsilon)\}$, and $\text{CT}(\varepsilon) = \{p\}$, $.(\varepsilon, p) = unexp$.*

Thus, an *initial $\mathcal{A}_1$-completion structure* contains an extended forest which is a set of single-node trees, one for every constants from P, having as root the respective constant, and, possibly, another single-node tree with anonymous root. The anonymous root, in case it exists, contains p, the predicate to satisfy. Otherwise the root of one of the other trees contains p. The intuition is that an *initial $\mathcal{A}_1$-completion structure* is a skeleton of a forest model which satisfies p, either by containing an atom of the form $p(a)$, where a is a constant appearing in the program, or an atom of the form $p(x)$, where x is an anonymous individual, in which $\varepsilon = x$. G is initialized with a single node graph – the atom $p(\varepsilon)$, which is already asserted as being part of the constructed model.

The forest model from Figure 2.2 has been evolved from the initial completion structure depicted in Figure 2.3 which has as ε, the root element where $smember$ has to be satisfied, the anonymous individual, x. There are two other single-node trees: T_a and T_b. The predicate $smember$ in the content of x is marked as unexpanded and G is a graph with a single vertex $smember(x)$.

An initial $\mathcal{A}_1$-completion structure for checking the satisfiability of a unary predicate p w.r.t. a FoLP P is evolved by means of *expansion rules* to a complete clash-free structure that corresponds to a finite representation of an open answer set in case p is satisfiable w.r.t. P. *Applicability rules* govern the application of the expansion rules.

[14] We use the prefix $\mathcal{A}_1$ to denote completion structures computed using this original algorithm as opposed to completion structures computed using the optimised algorithm described in the next section for which we will use the prefix $\mathcal{A}_2$.

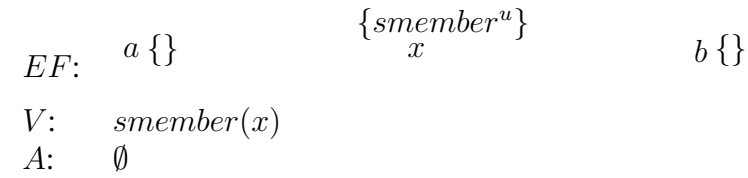


Figure 2.3: Initial completion structure for checking satisfiability of $smember$ w.r.t. P

2.3.4 Expansion Rules

Expansion rules justify why a certain atom is/is not part of an open answer set, or make guesses about the presence of certain atoms in the open answer set in order to have a complete model. Their scope of application is always a node or an arc in the completion structure. In the following, for a completion structure $\langle EF, \ ., \text{CT}, G\rangle$, let $x \in N_{EF}$ and $(x, y) \in A_{EF}$ be the node, resp. arc, under consideration.

(i) Expand unary positive. For a unary positive (non-free) $p \in \text{CT}(x)$ s. t. $.(x, p) = unexp$, choose a unary rule $r \in P_p$ for which s, the head term, matches x; ground this rule by substituting s with x, in case s is a variable, and each successor terms t_m which is a variable with a successor of x in EF s. t. the inequalities in ψ are satisfied (if needed one can introduce new successors of x in EF, either as successors of x in T, where $x \in T$, or in the form of constants from P). We motivate the presence of $p(x)$ in the open answer set by enforcing the body of this ground rule to be satisfied by inserting appropriate (possibly negated) predicate symbols in the contents of nodes/arcs of the structure. The newly inserted predicate symbols are marked as unexpanded and G is updated, by adding arcs from $p(x)$ to every atom in the body of the rule.

In our example, $smember$ is unexpanded in the initial completion structure. Rule r_2 is chosen to motivate the presence of $smember(x)$ in the open answer set. It is grounded by substituting X with x, and Y_1 and Y_2 with a and b, respectively: $smember(x) \leftarrow support(x, a), rmember(x, a), support(x, b),$
$rmember(x, b)$. We enforce the body of this ground rule to be true and update G accordingly (see Figure 2.4).

All currently unexpanded predicates, i.e., $support$ in the content of arcs (x, a) and (x, b), and $rmember$ in the content of nodes a and b, can be trivially expanded as $support$ is a free predicate and r_5 and r_6 are facts. However one still has to ensure that the structure constructed so far can be extended to an actual open answer set, i.e., it is consistent with the rest of the program. The following expansion rule takes care of this.

EF: $\{rmember^u\}a \xleftarrow[\{support^u\}]{\{smember^e\}} x \xrightarrow[\{support^u\}]{} b\,\{rmember^u\}$

V : $smember(x), support(x,a), support(x,b), rmember(x,a), rmember(x,b)$

A : $smember(x) \rightarrow support(x,a), smember(x) \rightarrow support(x,b),$
$smember(x) \rightarrow rmember(x,a), smember(x) \rightarrow rmember(x,b)$

Figure 2.4: The completion structure for checking satisfiability of $smember$ w.r.t. P after the expansion of $smember(x)$

EF: $a \xleftarrow[\{support\}]{\{sm, not\ rm, not\ co\}} x \xrightarrow[\{support\}]{} b$
$\{rm, not\ sm, not\ co\}$ $\{rm, not\ sm, not\ co\}$

V : $sm(x), support(x,a), support(x,b), rm(x,a), rm(x,b)$
A : $sm(x) \rightarrow support(x,a), sm(x) \rightarrow support(x,b),$
$sm(x) \rightarrow rm(x,a), sm(x) \rightarrow rm(x,b)$

Figure 2.5: Final completion structure for checking satisfiability of $smember$ w.r.t. P

(ii) Choose a unary predicate. If all predicates in $\text{CT}(x)$ and in the contents of x's outgoing edges are expanded and there are still unary predicates p which do not appear in $\text{CT}(x)$, pick such a p and inject either p or $not\ p$ in $\text{CT}(x)$. The intuition is that one has to explore all unary/binary predicates at every node/arc as some predicate which is not reachable by dependency-directed expansion can prohibit the extension of the partially constructed model to a full model. Consider the simple case where there is a predicate p defined only by the rule: $p \leftarrow not\ p$ and $\pm p$ does not appear in the body of any other rule. The program is obviously inconsistent, but this cannot be detected without trying to prove that p is or is not in the open answer set.

In our example, one does not know whether co or $not\ co$ belongs to $\text{CT}(x)$. We choose to inject $not\ co$ in $\text{CT}(x)$ and mark it as unexpanded.

(iii) Expand unary negative. Justifying a negative unary predicate $not\ p \in \text{CT}(x)$ means refuting the body of every ground rule which defines $p(x)$, or in other words refuting at least a literal from the body of every ground rule which defines $p(x)$. For more technical details concerning this rule we refer the reader to [35].

In our example, the unexpanded predicate in $\text{CT}(x)$, $not\ co$, is defined by one rule, r'_4, whose only possible grounding is:

$$co(x) \leftarrow not\ co(x), smember(x), rmember(x).$$

Refuting the body of this rule amounts to inserting $not\ rmember$ in $\text{CT}(x)$ ($smember$

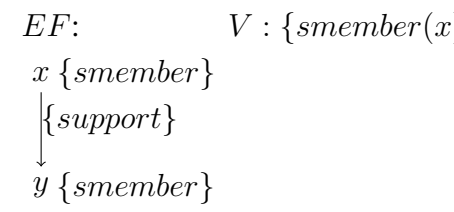


Figure 2.6: Completion structure for checking satisfiability of $smember$ w.r.t. P in which blocking is not applicable

and $not\ co$ are already part of the content of that node). At its turn, the presence of $not\ rmember$ in $\text{CT}(x)$ has to be motivated by using the expand unary negative rule, and the process goes on. Finally, we obtain a completion structure in which no expansion rule is further applicable and which represents exactly the forest model from Figure 2.2 (see Figure - $smember$ and $rmember$ are abbreviated with sm and rm, respectively):

Similarly to rules (i), (ii), and (iii) we define the expansion rules for binary predicates: (iv) *Expand binary positive*, (v) *Expand binary negative*, and (vi) *Choose binary*.

2.3.5 Applicability Rules

The applicability rules restrict the use of the expansion rules.

(vii) Saturation. A node $x \in N_{EF}$ is *saturated* if for all $p \in upreds(P)$, $p \in \text{CT}(x)$ or $not\ p \in \text{CT}(x)$, and no $\pm q \in \text{CT}(x)$ can be expanded with rules (i-iii), and for all $(x, y) \in A_{EF}$ and $p \in bpreds(P)$, $p \in \text{CT}(x, y)$ or $not\ p \in \text{CT}(x, y)$, and no $\pm f \in \text{CT}(x, y)$ can be expanded with (iv-vi). In other words, a node is saturated when every unary predicate symbol appears either in a positive form or negated in its label and every binary predicate symbol appears in a positive form or negated in the label of its outgoing arcs. We want to ensure saturations as nodes and arcs in order to guarantee that our structure will eventually stand for a complete model and not a partial one: partial stable models can not always be expanded to total models. No expansions should be performed on a node from N_{EF} which does not belong to $cts(P)$ until its predecessor is saturated.

(viii) Blocking. A node $x \in N_{EF}$ is *blocked* if there is an ancestor y of x in F, $y <_F x$, $y \notin cts(P)$, s. t. $\text{CT}(x) \subseteq \text{CT}(y)$ and the set $paths_G(y, x) = \{(p, q) \mid (p(y), q(x)) \in paths_G \wedge q \text{ is not free}\}$ is empty. We call (y, x) a *blocking pair*. No expansions can be performed on a blocked node. One can notice that subset blocking is not enough for pruning the tableau expansion. Every atom in the open answer set has to be finitely motivated [32, Theorem 2]: in order to ensure that, one has to check that there is no dependency in G between an atom formed with the blocking node and an atom formed with the blocked node.

Example 2.3.8. *Consider a restricted version of P from Example 2.3.3 which contains only rules r_1, and r_3. By checking satisfiability of $smember$ w.r.t. the new program one obtains the completion structure depicted in Figure 2.6.*

While the contents of nodes x and y are identical, they do not form a blocking pair as there is an arc in G between $smember(x)$ and $smember(y)$: unfolding the structure (justifying y similarly as x) would lead to an infinite chain of dependencies: $smember(x)$, $smember(y)$, $smember(z), \ldots$,.

The extra condition concerning paths in the dependency graph makes the blocking rule insufficient to ensure the termination of the algorithm. The following applicability rule ensures termination.

(ix) Redundancy. A node $x \in N_{EF}$ is *redundant* if it is saturated, it is not blocked, and there are k ancestors of x in F, $(y_i)_{1 \leqslant i \leqslant k}$, with $k = 2^p(2^{p^2} - 1) + 3$, and $p = |upreds(P)|$, s. t. $\text{CT}(x) = \text{CT}(y_i)$. In other words, a node is redundant if it is not blocked and it has k ancestors with content equal to its content. Any forest model of a FoLP P which satisfies p can be reduced to another forest model which satisfies p and has at most $k + 1$ nodes with equal content on any branch of a tree from the forest model, and furthermore the $(k + 1)st$ node, in case it exists, is blocked. One can thus search for forest models only of the latter type. As such the detection of a redundant node constitutes a clash and stops the expansion process.

2.3.5.1 Termination, Soundness, Completeness

An $\mathcal{A}_1$-completion structure is *contradictory* if for some $x \in N_{EF}/A_{EF}$ and $p \in upreds(P)/bpreds(P)$, $\{p, not\ p\} \subseteq \text{CT}(x)$. An $\mathcal{A}_1$-completion structure for a FoLP P and a $p \in upreds(P)$ is *complete* if it is a result of applying the expansion rules to the initial completion structure for p and P, taking into account the applicability rules, s. t. no expansion rules can be further applied.

Also, a complete $\mathcal{A}_1$-completion structure $CS = \langle EF, \text{CT}, ., G \rangle$ is $\mathcal{A}_1$-*clash-free* if: (1) CS is not contradictory (2) EF does not contain redundant nodes (3) G does not contain cycles (4) there is no $p \in upreds(P)/bpreds(P)$ and $x \in N_{EF}/A_{EF}$, x unblocked, s.t. $p \in \text{CT}(x)$, and $.(x, p) = unexp$.

It has been shown that an initial $\mathcal{A}_1$-completion structure for a unary predicate p and a FoLP P can always be expanded to a complete $\mathcal{A}_1$-completion structure (*termination*), that, if p is satisfiable w.r.t. P, there is a complete clash-free $\mathcal{A}_1$-completion structure (*soundness*), and, finally, that, if there is a complete clash-free $\mathcal{A}_1$-completion structure, p is satisfiable w.r.t. P (*completeness*).

In the worst case the algorithm runs in double exponential time, and a complete completion structure has a double exponential number of nodes in the size of the program. The high complexity is mostly due to the fact that blocking is not enough to ensure termination, and that, in particular, "anywhere blocking"[45] cannot be used as a termination

technique. As already explained this peculiarity appears as a result of adopting a minimal model semantics.

2.4 Optimized Reasoning with FoLPs

This section presents a knowledge compilation technique for reasoning with FoLPs together with an algorithm which makes use of this pre-compiled knowledge. The main idea is to capture all possible local computations, which are typically performed over and over again in the process of saturating the content of a node, by pre-computing all possible completion structures of depth 1 using the original algorithm described in the previous section. In the new algorithm, saturating the content of a node reduces to picking up one of the pre-computed structures which satisfies the existing constraints regarding the content of that node and appending the structure to the completion in construction: such constraints are sets of unexpanded (possibly negated) predicates which are needed to motivate the presence/absence in the open answer set of atoms constructed with the current node and the node above it.

Picking up a certain unit completion structure to saturate a node can impose strictly more constraints on the resulted structure than picking another unit completion structure with the same root content. Such constraints refer to: (1) the contents of the successor (non-blocked) nodes in a unit completion structure; (2) the paths from an atom formed with the root node of a unit completion to an atom formed with a successor node of such a completion – the more paths there are the harder blocking becomes. We discard such structures which are strictly more constraining than others, as they can be seen as redundant building blocks for a model.

The rest of the section formalizes and exemplifies these notions.

2.4.1 Optimized Reasoning with FoLPs

2.4.1.1 Unit Completion Structures

As mentioned in the introduction of this section, the intention is to obtain all completion structures of depth 1 which can be used as building blocks in our algorithm. We call such structures *unit completion structures*. The skeleton of such a structure, is a so-called *initial unit completion structure*. If they are to be used as building blocks in the algorithms, unit completion structures have to have as backbones trees of depth 1, and not forests. Hence, an initial unit completion structure is defined as a tree (unlike its counterpart notion from the previous section, initial completion structure, which is defined as a forest) with a single node, the root, which is either an anonymous constant or one of the constants already present in the program. The content of the root is empty.

Definition 2.4.1. *An* initial unit completion structure *for a FoLP P is a completion structure $\langle EF, \text{CT}, ., G\rangle$ with $EF = (F, ES)$, $F = \{T_\varepsilon\}$, where ε is a constant, possibly in $cts(P)$, $T_\varepsilon = \{\varepsilon\}$, $ES = \emptyset$, $G = \langle V, A\rangle$, $V = \emptyset$, $A = \emptyset$, and $\text{CT}(\varepsilon) = \emptyset$.*

A unit completion structure captures local computations which are performed to saturate the content of a node. As such, it is computed by expanding an initial unit completion structure until the root of the tree is saturated. The following definition captures this intuition.

Definition 2.4.2. *A* unit completion structure $\langle EF, \text{CT}, ., G\rangle$ *for a FoLP P, with $EF = (\{T_\varepsilon\}, ES)$, is an $\mathcal{A}_1$-completion structure derived from an initial unit completion structure by application of the expansion rules (i)-(vi) described in Section 2.3.4, according to the applicability rules introduced in Section 2.3.5, which has been expanded such that ε is saturated and for all s such that $\varepsilon \cdot s \in T_\varepsilon$, and for all $\pm p \in \text{CT}(\varepsilon \cdot s)$, $.(\pm p, \varepsilon \cdot s) = unexp$.*[15]

Example 2.4.3. *Consider the program Pr:*

$$
\begin{array}{lrcl}
r_1: & p(X) & \leftarrow & not\ p(X) \\
r_2: & p(X) & \leftarrow & f(X, Y), not\ q(Y) \\
r_3: & p(X) & \leftarrow & f(X, Y), p(Y) \\
r_4: & p(X) & \leftarrow & f(X, Y), not\ q(Y), p(Y) \\
r_5: & q(X) & \leftarrow & f(X, Y), not\ p(Y) \\
r_6: & f(X, Y) \vee not\ f(X, Y) & \leftarrow &
\end{array}
$$

Figure 2.7 depicts three unit completion structures for Pr. They all have the same content for the root node: $\{p, not\ q\}$. The presence of p in the content of the root node has been motivated in the first structure by means of rule r_4, in the second structure by means of rule r_3, and in the third structure by means of rule r_2. The different ways to motivate p lead to different sets of arcs in the dependency graphs belonging to each structure. On the other hand, to motivate that $not\ q$ is in the content of the root node, in each case it was shown that the body of r_5 grounded such that X is instantiated as the root node and Y as the successor node is not satisfied, or more concretely the presence of p in the content of the successor node was enforced in each case ($not\ f$ could not be used to invalidate the triggering of the rule as f was already present in the content of the arc from the root node to the successor node in each case).

One can notice that while the content of the successor node is included in the content of the root node in each of the cases, only for UC_3, the two nodes form a blocking pair as $paths_{G_3}(c, c1) = \emptyset$.

[15]The status function is relevant only in the definition/construction of a unit completion structure, but not in the context of using such structures. As such, we will denote a unit completion structure in the following as a triple $\langle EF, \text{CT}, G\rangle$.

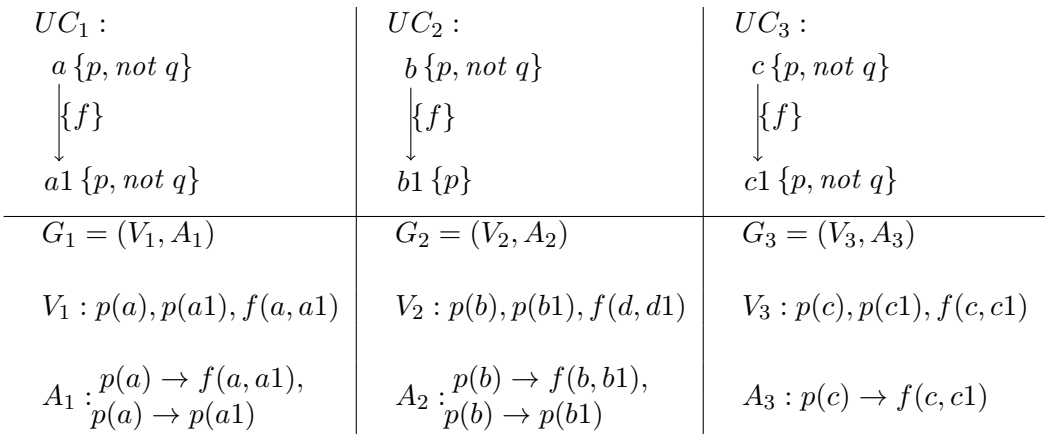


Figure 2.7: Three unit completion structures for Pr: UC_1, UC_2, and UC_3.

Definition 2.4.4. *A unit completion structure is* final *iff all its successor nodes are blocked, or they have empty contents.*

Proposition 2.4.5. *A final unit completion structure is a complete clash-free $\mathcal{A}_1$-completion structure.*

In our example UC_3 is a final unit completion structure, and thus also a complete clash-free $\mathcal{A}_1$-completion structure.

Proposition 2.4.6. *There is a deterministic procedure which computes all unit completion structures for a FoLP P in the worst-case scenario in exponential time in the size of P.*

Proof Sketch. We consider the transformation of the non-deterministic algorithm described in Definition 2.4.2 into a deterministic procedure. There are at most 2^n different values for the content of a saturated node, in this case for the content of the root of a unit completion structure, where $n = |upreds(P)|$. Justifying the presence of a predicate symbol p in the content of a node takes in the worst case polynomial time (choosing a possible grounding with successor nodes for some rule $r \in P_p$), but there is an exponential number of choices to do this (an exponential number of possible groundings for every rule). Justifying the presence of a negated predicate symbol $not\ p$ in the content of a node takes in the worst case exponential time (all possible groundings of every rule $r \in P_p$ have to be considered), while at every step of the computation there is a polynomial number of choices (for the ground rule in consideration, choosing a literal in its body to be refuted). Overall, such a deterministic procedure runs in exponential time in the worst case scenario. □

2.4.1.2 Redundant Unit Completion Structures

As seen in Example 2.4.3, there are unit completion structures with roots with equal content, but possibly different topologies, contents of the successor nodes and/or possibly different dependency graphs. As discussed in the introduction to this section it is worthwhile to identify structures which are strictly more constraining than others, in the sense that they impose more constraints on the content of the successor nodes of the structure and introduce more paths in the dependency graph. Such structures can be seen as redundant, and subsequently they can be discarded, as whenever such a structure is part of a complete clash-free completion structure, there is a complete clash-free completion structure in which a simpler structure is used as a building block. The following definition singles out such redundant structures.

Definition 2.4.7. *A* unit completion structure $UC_1 = \langle EF_1, \text{CT}_1, G_1 \rangle$*, with* $EF_1 = (\{T_{\varepsilon_1}\}, ES_1)$*, is said to be* redundant *iff there is another unit completion structure* $UC_2 = \langle EF_2, \text{CT}_2, G_2 \rangle$*, with* $EF_2 = (\{T_{\varepsilon_2}\}, ES_2)$ *s. t.:*

- *if* $\varepsilon_2 \in cts(P)$*, then* $\varepsilon_2 = \varepsilon_1$*;*
- $\text{CT}(\varepsilon_1) = \text{CT}(\varepsilon_2)$*;*
- *if* $\varepsilon_2 \cdot s_1, \ldots, \varepsilon_2 \cdot s_l$ *are the non-blocked successors of* ε_2*, there exist* l *distinct successors* $\varepsilon_1 \cdot t_1, \ldots, \varepsilon_1 \cdot t_l$ *of* ε_1 *such that:*
  - $\text{CT}(\varepsilon_2 \cdot s_i) \subseteq \text{CT}(\varepsilon_1 \cdot t_i)$*, for every* $1 \leqslant i \leqslant l$*, and*
  - $paths_{G_2}(\varepsilon_2, \varepsilon_2 \cdot s_i) \subseteq paths_{G_1}(\varepsilon_1, \varepsilon_1 \cdot t_i)$*, for every* $1 \leqslant i \leqslant l$*,*

  with at least one inclusion being strict.

The intuition is that the content of the successor nodes of a simpler structure can always be expanded in a similar way to the content of the corresponding successor nodes of the more complex structure, while the fact that there are fewer paths between atoms formed with the root node and atoms formed with successor nodes guarantees that no blocking conditions are violated, and even more, blocking might occur earlier than when using the more complex structure.

Considering the previous example, one can see that all three unit completion structures have the same content of the root and also the same topology: the tree in every structure has one node. However, UC_1 is more constraining than UC_2, and UC_2 at its turn is more constraining than UC_3:

- UC_1 is more constraining than UC_2 as the content of the successor node $a1$ of UC_1 is a strict superset of the content of the successor node $b1$ of UC_2. This is a result of using rule r_4 to motivate p in the first structure and using rule r_3 to motivate p in the second structure: one can see that rule r_3 is more general than r_4 as it allows the deduction of the same fact from fewer prerequisites.

- while the content of $b1$, the successor node of UC_2 is a strict subset of the content of $c1$, the successor node of UC_3, $c1$ is actually a blocked node, and thus UC_3 serves as a witness of the redundancy of UC_2: the third condition of Definition 2.4.7 is trivially fulfilled.

Thus, UC_1, and UC_2 are redundant structures: whenever one has to justify the presence of both p and $not\ q$ in the content of some node, one can do it in the way indicated by UC_3, which as already mentioned in the previous subsection is a final unit completion structure.

Proposition 2.4.8. *Computing the set of non-redundant unit completion structures for a FoLP P can be performed in the worst case in exponential time in the size of P.*

Proof Sketch. The result follows from the fact that there is an exponential number of unit completion structures for a FoLP P in the worst case scenario.□

2.4.1.3 Reasoning with FoLPs Using Unit Completion Structures

We define a new algorithm which uses the set of pre-computed non-redundant completion structures. We call this algorithm $\mathcal{A}_2$. As in the case of the previous algorithm, $\mathcal{A}_2$ starts with an initial $\mathcal{A}_2$-completion structure for checking satisfiability of a unary predicate p w.r.t. a FoLP P and expands this to a so-called $\mathcal{A}_2$-completion structure.

An *$\mathcal{A}_2$-completion structure* $\langle EF, \mathrm{CT}, ., G\rangle$ is defined similarly as an $\mathcal{A}_1$-completion structure, but the *status* function has a different domain, the set of nodes of the forest: $. : N_{EF} \longrightarrow \{exp, unexp\}$.

An *initial $\mathcal{A}_2$-completion structure for a unary predicate p and FoLP P* is defined similarly as an initial $\mathcal{A}_1$-completion structure for p and P, but in this case every node in the extended forest is marked as unexpanded.

Definition 2.4.9. *An* initial $\mathcal{A}_2$-completion structure *for checking satisfiability of a unary predicate p w.r.t. a FoLP P is a completion structure $\langle EF, \mathrm{CT}, ., G\rangle$ with $EF = (F, \emptyset)$, $F = \{T_\varepsilon\} \cup \{T_a \mid a \in cts(P)\}$, where ε is a constant, possibly in $cts(P)$, $T_x = \{x\}$ and $.(x) = unexp$, for $x \in \{\varepsilon\} \cup cts(P)$, $G = \langle V, \emptyset\rangle$, $V = \{p(\varepsilon)\}$, and $\mathrm{CT}(\varepsilon) = \{p\}$.*

In this scenario nodes are marked as expanded or unexpanded as a model is constructed by starting with some constraints in the form of an initial $\mathcal{A}_2$-completion structure, and then subsequently matching the content of the unexpanded nodes with existent non-redundant unit completion structures. We make explicit the notion of matching the content of a node with a unit completion structure by introducing a notion of *local satisfiability*:

Definition 2.4.10. *A unit completion structure UC for a FoLP P, $\langle EF, \mathrm{CT}, G\rangle$, with $EF = (\{T_\varepsilon\}, ES)$,* locally satisfies *a (possibly negated) unary predicate p iff $p \in \mathrm{CT}(\varepsilon)$. Similarly, UC locally satisfies a set S of (possibly) negated unary predicates iff $S \subseteq \mathrm{CT}(\varepsilon)$.*

All three unit completions in Figure 2.7 locally satisfy the set $\{p, not\ q\}$. It is easy to observe that if a unary predicate p is not locally satisfied by any unit completion structure UC for a FoLP P (or equivalently $not\ p$ is locally satisfied by every unit completion structure), p is unsatisfiable w.r.t. P. However, local satisfiability of a unary predicate p in every unit completion structure for a FoLP P does not guarantee 'global' satisfiability of p w.r.t. P. Recall the $\mathcal{A}_1$-completion structure depicted in Figure 2.6: the structure is a unit completion structure which locally satisfies $smember$, but $smember$ is not satisfied by the program considered in that particular example.

In the process of building an $\mathcal{A}_2$-completion structure $CS = \langle EF, \text{CT}, ., G\rangle$, with $G = (V, A)$, for a FoLP P by using unit completion structures as building blocks an operation commonly appears: the expansion of a node $x \in N_{EF}$ by addition of a unit completion structure $UC = \langle EF^{'}, \text{CT}^{'}, G'\rangle$, with $EF^{'} = (\{T_\varepsilon\}, ES^{'})$ and $G^{'} = (V^{'}, A^{'})$, which locally satisfies $\text{CT}(x)$, at x, given that its root matches with x [16]. We call this operation $expand_{CS}(x, UC)$. Formally, its application updates CS as follows:

- $.(x)=exp$,
- $N_{EF} = N_{EF} \cup \{x \cdot s \mid \varepsilon \cdot s \in T_\varepsilon\}$,
- $A_{EF} = A_{EF} \cup \{(x, x \cdot s) \mid (\varepsilon, \varepsilon \cdot s) \in A_{EF'}\}$,
- $\text{CT}(x) = \text{CT}(\varepsilon)$. For all s such that $\varepsilon \cdot s \in T_\varepsilon$, $\text{CT}(x \cdot s) = \text{CT}(\varepsilon \cdot s)$,
- $V = V \cup \{p(x) \mid p \in \text{CT}(\varepsilon)\} \cup \{p(x \cdot s) \mid p \in \text{CT}(\varepsilon \cdot s)\}$,
- $A = A \cup \{(p(\overline{z}), q(\overline{y})) \mid (p(z), q(y)) \in A^{'}\}$, where $\overline{\varepsilon} = x$, and $\overline{\varepsilon \cdot s} = x \cdot s$.

The algorithm has a new rule compared with the original algorithm which we call *Match*. This rule is meant to replace the expansion rules (i)-(vi) and the applicability rule (vii) from the original algorithm.

Match. For a node $x \in N_{EF}$: if $.(x) = unexp$ non-deterministically choose a non-redundant unit completion structure UC with root matching x which satisfies $\text{CT}(x)$ and perform $expand_{CS}(x, UC)$.

In this variant of the algorithm we still employ rules *(viii) Blocking* and *(ix) Redundancy* described in Section 2.3.3.

Definition 2.4.11. *A* complete $\mathcal{A}_2$-completion structure *for a FoLP P and a $p \in upreds(P)$, is an $\mathcal{A}_2$-completion structure that results from applying the rule* Match *to an initial $\mathcal{A}_2$-completion structure for p and P, taking into account the applicability rules (viii) and (ix), s. t. no other rules can be further applied.*

[16] An anonymous individual behaves like a variable: it matches with any term, while a constant matches only with itself; thus, unit completion structures with roots constants can only be used as initial building blocks for the trees with non-anonymous roots in the structure.

The local clash conditions regarding contradictory structures or structures which have cycles in the dependency graph G are no longer relevant:

Definition 2.4.12. *A complete $\mathcal{A}_2$-completion structure $CS = \langle EF, \text{CT}, ., G \rangle$ is* clash-free *if (1) EF does not contain redundant nodes (2) there is no node $x \in N_{EF}$, x unblocked, s.t. $st(x) = unexp$.*

2.4.1.4 Termination, Soundness, Completeness

The termination of the algorithm follows immediately from the usage of the blocking and of the redundancy rule:

Proposition 2.4.13. *An initial $\mathcal{A}_2$-completion structure for a unary predicate p and a FoLP P can always be expanded to a complete $\mathcal{A}_2$-completion structure.*

The algorithm is sound and complete:

Proposition 2.4.14. *A unary predicate p is satisfiable w.r.t. a FoLP P iff there is a complete clash-free $\mathcal{A}_2$-completion structure.*

Proof Sketch. The soundness of $\mathcal{A}_2$ follows from the soundness of $\mathcal{A}_1$: any completion structure computed using $\mathcal{A}_2$ could have actually been computed using $\mathcal{A}_1$ by replacing every usage of the *Match* rule with the corresponding rule application sequence used by $\mathcal{A}_1$ to derive the unit completion structure which is currently appended to the structure.

The completeness of $\mathcal{A}_2$ derives from the completeness of $\mathcal{A}_1$: any clash-free complete $\mathcal{A}_1$-completion structure can actually be seen as a complete clash-free $\mathcal{A}_2$-completion structure. It is essential here that the discarded unit completion structures were strictly more constraining than some other (preserved) unit completion structures. Whenever the expansion of a node in the complete clash-free $\mathcal{A}_1$-completion structure has been performed by a sequence of rules captured by a redundant unit completion structure, it is possible to construct a complete clash-free $\mathcal{A}_2$-completion structure by using the simpler non-redundant unit completion structure instead.□

As we still employ the redundancy rule in this version of the algorithm, a complete $\mathcal{A}_2$-completion structure has in the worst case a double exponential number of nodes in the size of the program. As such:

Proposition 2.4.15. *$\mathcal{A}_2$ runs in the worst-case in double exponential time.*

As reasoning with f-hybrid knowledge bases can be reduced to reasoning with FoLPs, we note that we can employ the new algorithm for reasoning with f-hybrid knowledge bases. However the upper bound concerning the complexity of f-hybrid knowledge bases has not been improved.

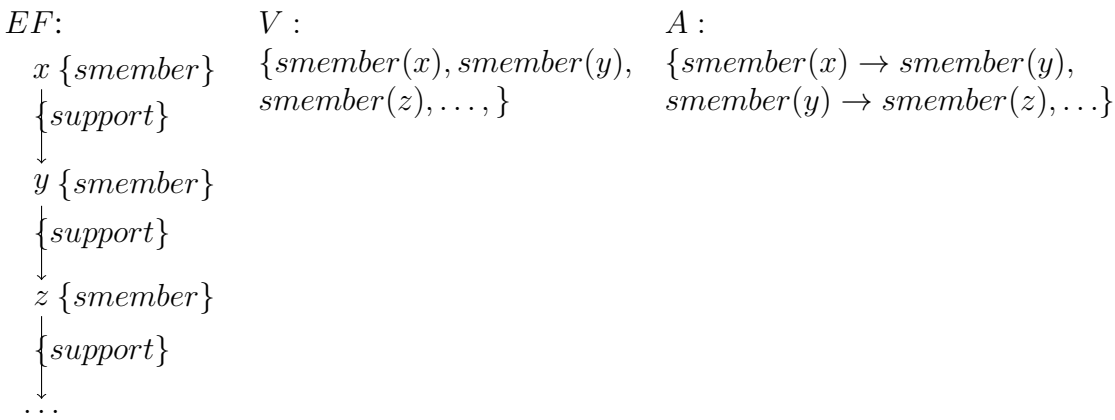


Figure 2.8: Hard problem due to (non-trivial) unsatisfiability of $smember$

2.4.2 Discussion

Our optimized algorithm runs in the worst case in double exponential time: this is not a surprise as the scope of the technique introduced here is saving time by avoiding redundant local computations. The worst-case running complexity of the algorithm depends on the depth of the trees which have to be explored in order to ensure completeness of the algorithm and on the fact that anywhere blocking is not feasible. Even with classical subset blocking one has to explore an exponential number of nodes across a branch in order for the algorithm to terminate. Thus, the only factor which would improve the worst-case performance is finding a termination condition which considers nodes in different branches. At the moment this seems highly unattainable.

The next step of this work is the evaluation of the new algorithm. We expect it will perform considerably better than the original algorithm in returning positive answers to satisfiability checking queries, while it might still take considerable time in the cases where a predicate is not satisfiable. Especially problematic are cases like the one described in Example 2.3.8 where there exists a unit completion structure which locally satisfies the predicate checked to be satisfiable, but the predicate is actually unsatisfiable. The algorithm will match identical structures until the limit imposed by the redundancy rule is met (see Figure 2.8). An obvious strategy for implementation is to establish a limit on the depth of the explored structures: in practice it is highly improbable that if there exists a solution, it can be found only in an open answer set of a considerable size (depth of the corresponding extended forest): actually, it is quite hard to come up with examples of such programs.

In D3.2 [35] we introduced the fragment of simple Forest Logic Programs by imposing a restriction on predicate recursion in programs. Simple Forest Logic Programs are expressive enough to simulate reasoning in the DL $\mathcal{ALCHOQ}$. As such, they gave rise to a tight

combination of simple FoLPs and $\mathcal{ALCHOQ}$ ontologies: simple f-hybrid knowledge bases. We also showed how our tableau algorithm for satisfiability checking w.r.t. FoLPs can be simplified for reasoning with simple FoLPs. The simplified algorithm runs in the worst case in exponential time. The optimization technique presented in this section can be straightforwardly applied to the simplified algorithm for reasoning with simple FoLPs. The worst case complexity remains the same. We note that also related approaches from literature (based on knowledge pre-compilation techniques) do not improve on the worst-case complexity of the algorithm, but, however, in practice they prove to be considerably better than the original algorithms.

For example, a knowledge compilation technique for reasoning with the Description Logic $\mathcal{ALC}$ is described in [22]. The pre-compilation technique consists of two steps. In the first step, all possible sub-concepts of a concept which are conjunctions of simple concepts and role restrictions are computed. These sub-concepts are captured by so-called *paths* which are sets of simple concepts and role restrictions. Paths which contain contradictory concepts are removed (these are called *links*), as well as paths which are super-sets of other paths. Note the similarity with our method as concerns removing local contradictions and redundancy. However our way of removing redundancy is much more sophisticated as we also consider redundancies in the set of dependencies between atoms in the model. In the second step, role restrictions are considered: all links for 'potentially reachable' concepts from the original concept are removed and a so-called *linkless graph* is obtained. Reachability is defined as the transitive closure of the relation between a concept and each of its role restriction fillers. Unlike that, our method investigates structures of depth 1 – we consider that pre-computing structures with higher depth would be an overkill. Finally, the method explores the linkless graph for checking concept consistency and answering subsumption queries. Consistency checking is polynomial in the size of the linkless graph, while query answering is linear in the size of the linkless graph for a certain class of restricted queries. As the linkless graph has a size exponential in the size of the original program, both reasoning tasks take in the worst case exponential time.

In the area of tableau algorithms for DL, several pre-processing techniques were employed successfully so far, like *normalization* and *absorption* [36]. Our method is closest to normalization, which seeks to eliminate local contradictions and tautologies, and as well to simplify some concepts. Absorption is a technique which removes general axioms in a TBox by absorbing them into primitive definition axioms, that is, inclusion axioms which have on the right hand side only simple concepts. Due to the particularities of stable model semantics this technique cannot be straightforwardly generalized to our setting.

Chapter 3

OWL 2 RL in ObjectLogic

3.1 Introduction

The OWL 2 RL profile defines a syntactic subset of OWL 2 which is aimed at applications that require scalable reasoning without sacrificing too much expressive power. There are a small number of experimental reasoners available implementing this profile like Jena [20], OWLIM [48] or the reference implementation OWLRL of Ivan Herman [31], as well as first industrial implementations, like the Oracle Database 11g OWL Reasoner [58].

The ontoprise contribution to the OntoRule Deliverable 3.3 consists in providing an implementation of OWL 2 RL in ObjectLogic. As reasoning engine we use the ontoprise product suite, specifically OntoBroker 6.x with ObjectLogic as ontology and rule language. OntoBroker 6.x supports both ObjectLogic and OWL 2 relying on the same internal representation and set of algorithms.

The implementation does not preserve the OWL semantics entirely. The main difference concerns the semantics of equality and was adopted due to performance reasons. All of the rules which have *owl:sameAs* in the rule head have been replaced by constraints.

In this chapter we give as preliminaries a short overview of OWL, OntoBroker and ObjectLogic, then explain the requirements which led to our specific type of implementation and finally give details of the implementation as well as a syntax reference.

We provide as part of ONTORULE deliverable D3.6 *Efficient processing of expressive combinations* [40] a demonstrator and a tutorial to illustrate the usage of the OWL in ObjectLogic constructs and the reasoning with the RL profile.

3.2 Preliminaries

3.2.1 OWL 2

OWL 2 is an ontology language for the Semantic Web with formally defined meaning. OWL 2 ontologies provide classes, properties, individuals, and data values. Classes provide an abstraction mechanism for grouping resources with similar characteristics. Like RDF schema classes, every OWL class is associated with a set of individuals, called the *class extension*. The individuals in the class extension are called the *instances* of the class. In the following we give a brief enumeration of the syntactical elements of OWL and illustrate them with some self-explaining examples. For a more detailed description we refer to the language reference [47] published online by W3C.

3.2.1.1 OWL 2 Syntax

OWL classes are described through class descriptions, which can be combined into class axioms. OWL has the following types of class descriptions:

1. Enumeration of individuals: *owl:oneOf*
2. Property restrictions
   - value constraints:
     owl:allValuesFrom, *owl:someValuesFrom, owl:hasValue*
   - cardinality constraints
     owl:maxCardinality, owl:minCardinality, owl:Cardinality
3. Intersection of two or more class descriptions *owl:intersectionOf*.
4. Union of two or more class descriptions *owl:unionOf*.
5. Complement of a class description *owl:complementOf*.

The class construct *owl:someValuesFrom* defines those individuals as part of the class which have some (that is, at least one) values for a given property from a given range. The class construct *owl:allValuesFrom* defines those individuals as part of the class which have only values for a given property from a given range and no values from other ranges.

The class constructs *owl:intersectionOf*, *owl:unionOf* and *owl:complementOf* correspond to conjunction, disjunction and negation operators in predicate logic.

Table 3.1 shows examples for classes defined by property restrictions .

Additionally, OWL provides the language constructs *rdfs:subClassOf*, *owl:disjointWith* and *owl:equivalentClass* denoted as class axioms. Table 3.2 shows examples for the intersection of two classes. Table 3.3 shows examples for equivalent and disjoint classes.

```
// owl:allValuesFrom
<owl:Restriction>
<owl:onProperty rdf:resource="#hasParent"/>
<owl:allValuesFrom rdf:resource="#Human" /> </owl:Restriction>
// owl:someValuesFrom
<owl:Restriction>
<owl:onProperty rdf:resource="#hasParent" />
<owl:someValuesFrom rdf:resource="#Mathematician" /> </owl:Restriction>
// owl:hasValue
<owl:Restriction>
<owl:onProperty rdf:resource="#hasParent" />
<owl:hasValue rdf:resource="#Adam" /> </owl:Restriction>
// owl:maxCardinality
<owl:Restriction>
<owl:onProperty rdf:resource="#hasParent" />
<owl:maxCardinality rdf:datatype="xsd:nonNegativeInteger">2
</owl:maxCardinality> </owl:Restriction>
```

Table 3.1: Example of OWL classes defined by property restrictions

```
// owl:intersectionOf
<owl:Class rdf:ID="Woman">
<owl:intersectionOf rdf:parseType="Collection">
<owl:Class rdf:about="#Person" />
<owl:Restriction>
<owl:onProperty rdf:resource="#hasGender" />
<owl:hasValue rdf:resource="#Female" />
</owl:Restriction>
</owl:intersectionOf>
</owl:Class>
```

Table 3.2: Example for the intersection of two classes

```
// owl:equivalentClass
<owl:Class rdf:about="#Human">
<owl:equivalentClass rdf:resource="#Person"/>
</owl:Class>
// owl:disjointWith
<owl:Class rdf:ID="Woman">
<rdfs:subClassOf rdf:resource="#Person"/>
<owl:disjointWith rdf:resource="#Man"/>
</owl:Class>
```

Table 3.3: Example for equivalent classes and disjoint classes

OWL has two types of properties, object properties which connect individuals with individuals and datatype properties which connect individuals with data values. The property axiom constructs are the following:

- RDF schema constructs

rdfs:subPropertyOf, *rdfs:domain*, *rdfs:range*

- Algebraic properties
  symmetric, transitive, reflexive, asymmetric
- Relations between properties
  owl:EquivalentProperty, *owl:inverseProperty*
- Global cardinality constraints
  owl:functionalProperty, *owl:inverseFunctionalProperty*

Individuals are defined by their class membership or by identity. OWL has no unique name assumption. Instead, OWL provides three constructs for defining identity between individuals:

- *owl:sameAs* - means that two URIs refer to the same individual
- *owl:differentFrom* - means that two URIs refer to different individuals
- *owl:allDifferent* - means that a list of individuals are mutually different.

```
// Transitive property
<owl:ObjectProperty rdf:ID="hasAncestor">
<rdf:type rdf:resource="owl:TransitiveProperty"/>
<rdfs:domain rdf:resource="#Person"/>
<rdfs:range rdf:resource="#Person"/>
</owl:ObjectProperty>

// Functional property
<owl:ObjectProperty rdf:ID="husband">
<rdf:type rdf:resource="owl:FunctionalProperty" />
<rdfs:domain rdf:resource="#Woman" />
<rdfs:range rdf:resource="#Man" />
</owl:ObjectProperty>
```

Table 3.4: Example of a transitive and functional property in OWL syntax

3.2.1.2 OWL 2 RL Profile

The OWL 2 RL profile [43] is aimed at applications that require scalable reasoning without sacrificing too much expressive power. OWL 2 RL allows for polynomial reasoning (consistency, classification and instance checking) using rule-based technologies.

The profile defines a syntactic subset of OWL 2 which is amenable to implementation using rule-based technologies and presenting a partial axiomatization in the form of first-order implications that can be used as the basis for such an implementation. For ontologies

satisfying the OWL 2 RL syntactic constraints, a suitable rule-based implementation will have desirable computational properties; for example, it can return *all* and *only* the correct answers to certain kinds of query. The DL underlying the OWL 2 RL profile is a Datalog-rewritable DL, and thus, it can be used in the context of tractable dl-programs

In OWL RL the use of OWL constructs is restricted to defined syntactic positions. In *subClassOf* axioms, the constructs in the subclass and superclass expressions must follow the usage patterns shown in Table 2 in the OWL 2 RL Specification [43].

With exception of *owl:Thing*, any class is allowed as subclass or superclass expression. The intersection of classes and the hasValue class expression may be a subclass or a superclass expression. For the other constructs, the following holds: oneOf, unionOf, someValuesFrom are allowed only as subclass expressions; complementOf, allValuesFrom and maxCardinality 0/1 only as superclass expressions.

The semantics of the RL profile is given by the set of rules specified and published online by the W3C recommendation [43]. Each rule is given a short unique name. For example, the rules for the semantics of equality (*eq-ref, eq-sym, eq-trans, eq-rep-s, eq-rep-p, eq-rep-o, eq-diff1, eq-diff2, eq-diff3*) define the *owl:sameAs* relation as being reflexive, symmetric and transitive and axiomatize the standard replacement properties of equality.

The set of rules is not minimal, as certain rules are implied by other ones. This was done to make the definition of the semantic consequences of each piece of OWL 2 vocabulary self-contained. For example the second rule for the intersection of classes *cls-int2* is implied by the schema rule *scm-int*, see Section 3.4.2.11.

3.2.2 OntoBroker

The ontoprise product suite is the most complete SemanticWeb infrastructure and worldwide the only one supporting all major W3C SemanticWeb recommendations including OWL, RDF(s), RIF and ObjectLogic. According to the "OpenRuleBench" benchmark, OntoBroker is the leading rule and ontology engine with respect to the evaluation performance.

The current version of the product suite is OntoBroker 6.0/OntoStudio 3.0, released in June 2010. The main changes with respect to the previous release OntoBroker 5.3/OntoStudio 2.3 is that F-logic as main supported language is replaced by ObjectLogic, as well as improved performance, optimization and support for collaborative ontology engineering.

The ObjectLogic engine is a deductive, object oriented database system. Its underlying formalism is the frame logic ObjectLogic, the successor of F-Logic. Its main features are reasoning and query answering over ObjectLogic knowledge bases.

ObjectLogic rules consist of conjunctions of literals in the rule head and arbitrary predicate logic formulas in the rule body. In OntoBroker the rules are transformed in an extended form of datalog programs using the Lloyd-Topor [42] transformation and eval-

uated during query time. Functions and negations are supported as well. OntoBroker supports multiple query languages. The primary query language and the native format of OntoBroker is ObjectLogic. Other supported languages are a subset of SPARQL and SQL. Disjunctive queries or queries which contain builtins, temporary facts, etc. are only supported by ObjectLogic.

OntoBroker supports high-performance and scalable reasoning using several different evaluation algorithms like *Bottom-Up Evaluation*, *Magic Sets* [9] or *Dynamic Filtering*. The performance of OntoBroker can be increased by other means like the optimisation of ordering of body literals or the materialisation of rules at start time.

The functionality can be extended with procedural attachments and built-ins. With this functionality it is possible to connect external data sources during runtime in order to integrate the data for query evaluation. Built-ins are Java programs that can be seamlessly accessed from ObjectLogic rules and queries via built-in *predicates*. They can perform arithmetic or string operations. The list of built-ins can be extended by implementing the provided interface.

3.2.3 ObjectLogic

ObjectLogic is the ontology and rule language supported by OntoBroker 6.x and the successor of F-logic with additional extensions like property hierarchies, algebraic properties and constraints. An ObjectLogic ontology consists of a TBox (schema definitions, concept/class and property hierarchy), an ABox (instances), user-defined rules, constraints and queries. Variable names are preceded by a question mark. Rules consist of a premise (rule body) and a conclusion (rule head) part, separated by the infer-symbol :-. ObjectLogic supports the closed world assumption (everything which is not explicitly known, is assumed to be false) and the unique name assumption (every resource is identified by a unique name). For further details see [3].

```
head(?X) :- body(?X).
```

A constraint has the syntax

```
!- body(?X).
```

A constraint is considered violated if a result is yielded when the constraint is posed as a query. A query has the syntax

```
?- body(?X).
```

The following example in Table 3.5 illustrates a small ObjectLogic-ontology, excerpt from a genealogy, with a property hierarchy, userdefined-rule, constraint and queries.

```
// Schema:
Person[].
Woman::Person.
Person[hasAncestor {0:*,transitive} *=> Person].
Person[hasSister*=>Woman].
hasMother << hasParent. hasParent << hasAncestor.
// Instances:
X:Woman[hasMother->XX].
XX:Woman[hasMother->XXX].
// User-defined rule
@{hasAunt}
?A[hasAunt->?B] :-
?A:Person[hasParent->?P] AND ?P:Person[hasSister->?B] AND ?B:Woman.
// Constraint
!- ?X:Person[hasAunt -> ?C:Woman] AND (NOT EXIST ?S ?C[hasSibling->?S]).
// Query:
?- ?X:Person[hasAncestor->?Y].
// Results:
XX
XXX
XXXX
```

Table 3.5: Example ontology in ObjectLogic: genealogy

3.3 Requirements

ontoprise has planned to implement the OWL 2 RL profile in ObjectLogic as part of its product strategy. The intention is to come as close as possible to the OWL standard while prioritizing features with high industrial relevance and avoiding features that lead to performance problems. Features identified as candidates for being included in the product are, in the order of their importance: property hierarchies, functional properties, algebraic properties, cardinality restrictions.

3.3.1 Property Hierarchies and Chains

Property hierarchies have been identified as an important and useful feature to be included in ObjectLogic core, since it occurs often that properties are specializations of other properties. A property hierarchy is realized via the subproperty relationship << which states that one property is subproperty of another. The following sample ontology illustrates the property hierarchy concept, where hasMother is a subproperty of hasParent and hasParent is a subproperty of hasAncestor. Without a property hierarchy, it would be necessary to write user-defined rules in order to find all ancestors of a person.

Property chains enable the definitions of new properties as a chain of existing properties, like defining hasAunt by chaining hasParent and hasSister:

_PropertyChain([hasParent,hasSister])«hasAunt.

3.3.2 Algebraic Properties

Currently supported features for relations within ObjectLogic core are symmetric properties, transitive properties and the inverse of a property. A symmetric property/relation *r* is a relation for which holds that if *a[r->b]* then *b[r->a]* also holds. Examples for symmetric relations are *hasRelative*, *hasSibling* or *hasFriend*.

A transitive relation *r* is a relation for which holds that if *a[r->b]* and *b[r->c]* then also holds *a[r->c]*. An example for a transitive relation from common sense is *hasAncestor*. An inverse relation *R2* of a relation *R1* is a relation for which holds that if *A[R1 -> B]* then *B[R2 -> A]* also holds. Example for an inverse relation: *hasSon* is the inverse of *hasFather*.

The OWL constructs for irreflexive and asymmetric properties are also to be considered for support in ObjectLogic core. A property is declared *reflexive* if each individual is related to itself by the property, like the hasRelative property: every person is trivially a relative of itself. A property is declared *irreflexive* if no individual can be related to itself by the property, like the parentOf property: a person cannot be its own parent.

3.3.3 Cardinality Restrictions

With cardinality constraints it is possible to restrict the number of values for a particular property. The OWL 2 RL profile restricts MaxCardinality to the values 0 and 1. An OWL reasoner, following the model theoretic semantics of OWL, would derive that, if the cardinality is one and there exist two values of the property, the two instances are the same. Within ObjectLogic the cardinality constraints are interpreted as actual constraints. So in the example above ObjectLogic would derive that the model described above does not satisfy the constraint.

3.3.4 Equality

OWL does not make the unique name assumption. Different names may refer to the same entity. Instead, OWL supports the equality expressions *owl:sameAs*, *owl:differentFrom* and *owl:allDifferent*. The ontology engineer may declare arbitrary individuals as same or different.

In order to decide on the support for these constructs in ObjectLogic, we have to consider first the (industrial) usecase aspect and second the performance aspect.

For the usecase aspect, we found that the most widespread use of *owl:sameAs* is in the linked data community, where it is used in interlinking data-sets on the web. The latter practice seems error-prone, in particular with regards to its interactions with inference. Halpin and Hayes [30] discuss four distinct uses of *owl:sameAs* in addition to the precise idea of “same thing as”, namely: *Same Thing As But Different Context*, *Same Thing As*

But Referentially Opaque, *Represents*, *Very Similar To* which all lead to non standard interpretations of the *owl:sameAs* construct.

An example (see [30]) for misused *owl:sameAs* is the concept of sodium in DBpedia, which has an *owl:sameAs* link to the concept of sodium in OpenCyc. The OpenCyc ontology says that an element is the set (class) of all pieces of the pure element, so that for example sodium in Cyc has a member which is the lump of pure metallic sodium. On the other hand, sodium as defined by DBPedia is used to also include isotopes, which have different number of neutrons than standard sodium. So, one should not state the number of neutrons in DBPedias use of sodium, but one can with OpenCyc. Therefore, *owl:sameAs* here is an error, as it does not allow mutual substitutivity.

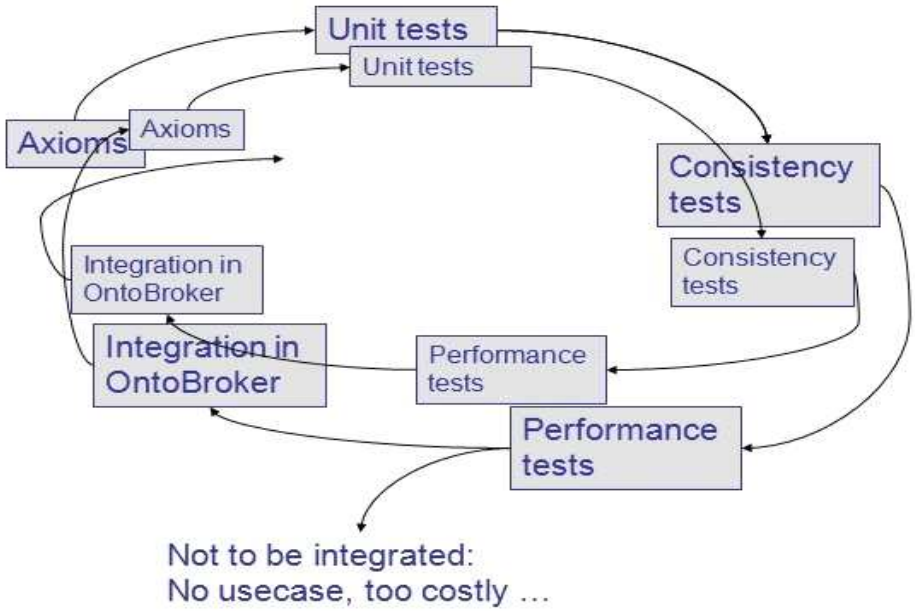
Another typical use of *owl:sameAs* could be to equate individuals defined in different documents to one another, as part of unifying two ontologies. This assumes that the ontology engineer has a good knowledge of the respective ontologies in order to be able to declare individuals as same.

The usecases with industrial relevance for the *owl:sameAs* construct are yet scarse, since most applications rely on data from databases, where the unique name assumption hold.

For the performance aspect, we found that inferring equality is performance-expensive. A rule engine needs additional mechanisms to assure efficient reasoning, like performance-enhancing algorithms and parallel inferencing. We analyzed different methods of supporting efficient reasoning.The most promising is the concept of ground equality, an approach where equality is not inferred and all individuals declared as same are put in equivalence classes. From each equivalence class one individual is chosen as a representative and all of the inferences for the chosen equivalence class are consolidated using that representative.

3.4 Implementation

In the period M13-M24 of the ONTORULE project ontoprise has realized the implementation of the OWL 2 RL Profile in ObjectLogic. The implementation follows an iterative process. After defining and testing the syntactical elements and part of the axioms (subclass and subproperties), unit tests were run for entailment.



3.4.1 Syntax

In a first step the necessary syntactical elements were implemented.

Every OWL 2 RL construct which has an equivalent construct in ObjectLogic is mapped directly, as the following examples in Table 3.6 show for subClassOf and subPropertyOf. All OWL 2 RL constructs which do not have an equivalent in ObjectLogic are mapped to special predicates, as the following example in Table 3.7 shows for the *owl:allValuesFrom* restriction.

The OWL 2 RL in ObjectLogic syntax implementation has some specific details. First, we make no distinction between ObjectProperty and DataProperty constructs. Second, we do not support restrictions using n-ary data range since they are not supported by the OWL API [4]. Third, some of the newly introduced predicates are rewritten internally as predicates with additional facts, needed by the axiomatization. An example: The OWL construct *owl:oneOf* is represented by the ObjectLogic predicate *_OneOf* and has the

Language feature	**OWL**	**ObjectLogic**
subClassOf	<owl:Class rdf:about="#Man"> <rdfs:subClassOf> <owl:Class rdf:about="#Person"/> </rdfs:subClassOf> </owl:Class>	Man**::**Person.
subPropertyOf	<owl:ObjectProperty rdf:ID="hasDesc"> <rdfs:domain rdf:resource="#Wine" /> <rdfs:range rdf:resource="#Desc"/> </owl:ObjectProperty> <owl:ObjectProperty rdf:ID="hasColor"> <rdfs:subPropertyOf rdf:resource="#hasDesc" /> <rdfs:range rdf:resource="#Color" /> </owl:ObjectProperty>	Wine[hasDesc*=>Desc]. hasColor<<hasDesc.

Table 3.6: SubClassOf and SubPropertyOf in OWL vs. ObjectLogic

Language feature	**OWL**	**ObjectLogic**
allValuesFrom	<owl:Restriction> <owl:onProperty rdf:resource="#hasParent" /> <owl:allValuesFrom rdf:resource="#Human" </owl:Restriction>	C::_AllValuesFrom (hasParent, Human).

Table 3.7: AllValuesFrom class descriptor in OWL and ObjectLogic

internal representation *$OneOf*. The first position in the internal predicate *$OneOf(?, ?c, ?y)* is placeholder for an internal unique id, *?c* is the name of the complex class and *?y* represents a list variable.

The complete syntax reference is given in Section 3.6.

3.4.2 Semantics

The semantics for the OWL-equivalent predicates is given by implementing the rule subset from the W3C OWL 2 RL profile specification [43] in ObjectLogic.

In the following subsections, we point out the details for the axiomatization of the single language constructs. Note that we have deviations from the standard OWL semantics for a number of constructs, in particular: equality, functional and inverse functional properties, equivalent and disjoint classes, complement of a class. We replace all the rules which have *owl:sameAs* in the rule head by constraints. We have a special handling for equality. We use internal predicates to represent a number of rules which require additional facts.

Where the rules are self-explaining, we simply state them in the form of a table.

3.4.2.1 owl:Thing

The axiomatization of *owl:Thing* could be done explicitly with *scm-cls*, stating that each class is a subclass of *owl:Thing*. This would be performance costly, and we choose not to axiomatize it, rather adding additional axioms where needed.

3.4.2.2 Equality

The semantics of the equality relation owl:sameAs is given by the axioms *eq-ref, eq-sym, eq-trans and eq-rep-s, eq-rep-s,eq-rep-o*. The constructs *owl:differentFro*m and *owl:AllDifferent* are realized by *eq-diff1, eq-diff2, eq-diff3*.

Axiom	**RDF Syntax**
eq-ref	IF T(?s, ?p, ?o) THEN T(?s, owl:sameAs, ?s) T(?p, owl:sameAs, ?p) T(?o, owl:sameAs, ?o)
eq-sym	IF T(?x, owl:sameAs, ?y) THEN T(?y, owl:sameAs, ?x)
eq-trans	IF T(?x, owl:sameAs, ?y) T(?y, owl:sameAs, ?z) THEN T(?x, owl:sameAs, ?z)
eq-rep-s	IF T(?s, owl:sameAs, ?s1) T(?s, ?p, ?o) THEN T(?s1, ?p, ?o)
eq-rep-p	IF T(?p, owl:sameAs, ?p1) T(?s, ?p, ?o) THEN T(?s, ?p1, ?o)
eq-rep-o	IF T(?o, owl:sameAs, ?o1) T(?s, ?p, ?o) THEN T(?s, ?p, ?o1)
eq-diff1	IF T(?x, owl:sameAs, ?y) T(?x, owl:differentFrom, ?y) THEN false
eq-diff2	IF T(?x, rdf:type, owl:AllDifferent) T(?x, owl:members, ?y) LIST[?y, ?z1, ..., ?zn] T(?zi, owl:sameAs, ?zj) THEN false
eq-diff3	IF T(?x, rdf:type, owl:AllDifferent) T(?x, owl:distinctMembers, ?y) LIST[?y, ?z1, ..., ?zn] T(?zi, owl:sameAs, ?zj) THEN false

Table 3.8: Semantics of equality

We analyzed three variants for implementing *owl:sameAs* in ObjectLogic.

Equality handling #1: *owl:sameAs* is fully axiomatized.

The axioms *eq-diff2* and *eq-diff3* define the semantics for the AllDifferent construct, in ObjectLogic denoted by DifferentIndividuals1. The auxiliary axiom *reduce-diff* (see table 3.9) reduces DifferentIndividuals1, the construct for a list with mutually different members, to DifferentIndividuals, the predicate for a list with 2 different members.

Equality handling #2: *owl:sameAs* is not axiomatized.

We state a constraint which is violated by all individuals in the ontology declared as *_SameIndividual* and not having the same name.

ObjectLogic rules

@{'eq-ref'} _SameIndividual(?s,?s) AND _SameIndividual(?p,?p) AND _SameIndividual(?o,?o) :- ?s[?p->?o].

@{'eq-sym'} _SameIndividual(?y,?x):-_SameIndividual(?x,?y).

@{'eq-trans'} _SameIndividual(?x,?z) :- _SameIndividual(?x,?y) AND _SameIndividual(?y,?z).

@{'eq-diff1'} !- _SameIndividual(?x,?y) AND _DifferentIndividuals(?x,?y).

@{'eq-diff2'} _DifferentIndividuals(?A, ?B) :- _DifferentIndividuals1(?X) AND ?X[_member->?A] AND ?X[_member->?B] AND NOT _SameIndividual(?A, ?B).

@{'eq-diff3'} _SameIndividual(?A, ?B) :- _SameIndividual1(?X) AND ?X[_member->?A] AND ?X[_member->?B].

@{'reduce-diff'} _DifferentIndividuals(?i,?j) :- _DifferentIndividuals1(?x) AND ?x[_member->?i] AND ?x[_member->?j] AND NOT _SameIndividual(?i,?j).

Table 3.9: Fully axiomatized equality in ObjectLogic

ObjectLogic rules

@{'eq-sameas-cst'} !- _SameIndividual(?x,?y) AND NOT ?x=?y.

Table 3.10: Constraint for Unique Name Assumption.

Equality handling #3: *owl:sameAs* is axiomatized partially.

For performance reasons, we choose not to infer equality. *_SameIndividual(?x,?y)* is not inferred. Internally we create equivalence classes for same individuals. Thus, the axioms *eq-ref*, *eq-sym*, *eq-trans* are not needed.

3.4.2.3 Equivalent Classes

Classes are equivalent when they are mutually in a subclass relationship and thus contain the same set of individuals. Table 3.11 shows in the upper part the axioms for equivalent classes in RDF syntax and in the lower part their implementation as ObjectLogic rules. *cax-eqc1* and *cax-eqc1* state that an individual from one of the classes must be also in the other class. *scm-eqc1* states that if we have two equivalent classes, they must be mutually in a subclass relationship. *scm-eqc2* states that if we have two classes with a mutual subclass relationship, they must be equivalent classes. Note that *scm-eqc2* is implemented as constraint rather than as rule: for performance reasons, we do not want to have the *owl:sameAs* construct in a rule header.

Axiom	**RDF Syntax**
cax-eqc1	IF T(?c1, owl:equivalentClass, ?c2) T(?x, rdf:type, ?c1) THEN T(?x, rdf:type, ?c2)
cax-eqc2	IF T(?c1, owl:equivalentClass, ?c2) T(?x, rdf:type, ?c2) THEN T(?x, rdf:type, ?c1)
scm-eqc1	IF T(?c1, owl:equivalentClass, ?c2) THEN T(?c1, rdfs:subClassOf, ?c2) T(?c2, rdfs:subClassOf , ?c1)
scm-eqc2	IF T(?c1, rdfs:subClassOf, ?c2) T(?c2, rdfs:subClassOf, ?c1) THEN T(?c1, owl:equivalentClass , ?c2)

ObjectLogic rules

```
@{'cax-eqc1'} ?x:?c2 :- _EquivalentClasses(?c1,?c2) AND ?x:?c1.
@{'cax-eqc2'} ?x:?c1 :- _EquivalentClasses(?c1,?c2) AND ?x:?c2.
@{'scm-eqc1'} ?c1::?c2 AND ?c2::?c1 :- _EquivalentClasses(?c1,?c2).
@{'scm-eqc2'} !- ?c1::?c2 AND ?c2::?c1 AND NOT _EquivalentClasses(?c1,?c2).
```

Table 3.11: Axioms for Equivalent Classes in OWL and ObjectLogic

3.4.2.4 Disjoint Classes

Axiom *cax-dw* is realized by a constraint which states a violation if an individual is simultaneously in two disjoint classes.

Axiom	**RDF Syntax**
cax-dw	IF T(?c1, owl:disjointWith, ?c2) T(?x, rdf:type, ?c1) T(?x, rdf:type, ?c2) THEN false

ObjectLogic rules

```
@{'cax-dw'} !- _DisjointClasses(?c1,?c2) AND ?x:?c1 AND ?x:?c2.
```

Table 3.12: Axioms for Disjoint Classes in ObjectLogic

3.4.2.5 OneOf Class Description

Axiom	**RDF Syntax**
cls-oo	IF T(?c, owl:oneOf, ?x) LIST[?x, ?y_1, ..., ?y_n] THEN T(?y_1, rdf:type, ?c) . . . T(?y_n, rdf:type, ?c)

ObjectLogic rules

```
@{'cls-oo'} ?y_1:?c :- $OneOf(?, ?c, ?y_1).
```

Table 3.13: Axioms for OneOf Class Description in ObjectLogic

3.4.2.6 HasValue Class Description

The HasValue class description is a particular case of the someValuesFrom class description.

Axiom	RDF Syntax
cls-hv1	IF T(?x, owl:hasValue, ?y) T(?x, owl:onProperty, ?p) T(?u, rdf:type, ?x) THEN T(?u, ?p, ?y)
cls-hv2	IF T(?x, owl:hasValue, ?y) T(?x, owl:onProperty, ?p) T(?u, ?p, ?y) THEN T(?u, rdf:type, ?x)
scm-hv	IF T(?c1, owl:hasValue, ?i) T(?c1, owl:onProperty, ?p1) T(?c2, owl:hasValue, ?i) T(?c2, owl:onProperty, ?p2) T(?p1,rdfs:subPropertyOf, ?p2) THEN T(?c1, rdfs:subClassOf, ?c2)

ObjectLogic rules

@{'cls-hv1'} ?u[?p->?y] :- $HasValue(?,?x,?p,?y) AND ?u:?x.
@{'cls-hv2'} ?u:?x :- $HasValue(?,?x,?p,?y) AND ?u[?p->?y].
@{'scm-hv'} ?c1::?c2 :- $HasValue(?,?c1,?p1,?y) AND $HasValue(?,?c2,?p2,?y) AND ?p1<<?p2.

Table 3.14: Axioms for HasValue Class Description in ObjectLogic

3.4.2.7 SomeValuesFrom Class Description

An individual ?u is in the SomeValuesFrom class description for the property ?p and the class ?x if there is at least one individual ?v such that ?u[?p->?v]. The class axioms *cls-svf1* and *cls-svf2* state the rules which infer when an individual is part of the class ?x given by the SomeValuesFrom restriction on the property p. The schema axiom *scm-svf1* relates the class with concept hierarchies and *scm-svf2* relates it with property hierarchies.

Axiom	RDF Syntax
cls-svf1	IF T(?x, owl:someValuesFrom, ?y) T(?x, owl:onProperty, ?p) T(?u, ?p, ?v) T(?v, rdf:type, ?y) THEN T(?u, rdf:type, ?x)
cls-svf2	IF T(?x, owl:someValuesFrom, owl:Thing) T(?x, owl:onProperty, ?p) T(?u, ?p, ?v) THEN T(?u, rdf:type, ?x)
scm-svf1	IF T(?c1, owl:someValuesFrom, ?y1) T(?c1, owl:onProperty, ?p) T(?c2, owl:someValuesFrom, ?y2) T(?c2, owl:onProperty, ?p) T(?y1, rdfs:subClassOf, ?y2) THEN T(?c1, rdfs:subClassOf , ?c2)
scm-svf2	IF T(?c1, owl:someValuesFrom, ?y) T(?c1, owl:onProperty, ?p1) T(?c2, owl:someValuesFrom, ?y) T(?c2, owl:onProperty, ?p2) T(?p1, rdfs:subPropertyOf, ?p2) THEN T(?c1, rdfs:subClassOf, ?c2)

ObjectLogic rules

@{'cls-svf1'} ?u:?x :- $SomeValuesFrom(?,?x,?p,?y) AND ?u[?p->?v] AND ?v:?y.
@{'cls-svf2'} ?u:?x :- $SomeValuesFrom(?,?x,?p,owl#Thing) AND ?u[?p->?v].
{'scm-svf1'}
?c1::?c2 :- $SomeValuesFrom(?,?c1,?p,?y1) AND $SomeValuesFrom(?,?c2,?p,?y2) AND ?y1::?y2.
@{'scm-svf2'}
?c1::?c2 :- $SomeValuesFrom(?,?c1,?p1,?y) AND $SomeValuesFrom(?,?c2,?p2,?y) AND ?p1<<?p2.

Table 3.15: Axioms for SomeValuesFrom Class Description in ObjectLogic

3.4.2.8 AllValuesFrom Class Description

cls-avf states that all values *?v* of individuals *?u* in the *owl:allValuesFrom* class description *?x* on the property *?p* must be in restricting range *?y*. The schema axiom *scm-avf1* relates the class with concept hierarchies and *scm-avf2* relates it with property hierarchies.

Axiom	**RDF Syntax**
cls-avf	IF T(?x, owl:allValuesFrom, ?y) T(?x, owl:onProperty, ?p) T(?u, rdf:type, ?x) T(?u, ?p, ?v) THEN T(?v, rdf:type, ?y)
scm-avf1	IF T(?c1, owl:allValuesFrom, ?y1) T(?c1, owl:onProperty, ?p) T(?c2, owl:allValuesFrom, ?y2) T(?c2, owl:onProperty, ?p) T(?y1, rdfs:subClassOf, ?y2) THEN T(?c1, rdfs:subClassOf, ?c2)
scm-avf2	IF T(?c1, owl:allValuesFrom, ?y) T(?c1, owl:onProperty, ?p1) T(?c2, owl:allValuesFrom, ?y) T(?c2, owl:onProperty, ?p2) T(?p1, rdfs:subPropertyOf, ?p2) THEN T(?c2, rdfs:subClassOf, ?c1)

ObjectLogic rules

```
@{'cls-avf'} ?v:?y :- $AllValuesFrom(?,?x,?p,?y) AND ?u:?x AND ?u[?p->?v].
@{'scm-avf1'}
?c1::?c2 :- $AllValuesFrom(?,?c1,?p,?y1) AND $AllValuesFrom(?,?c2,?p,?y2) AND ?y1::?y2.
@{'scm-avf2'}
?c2::?c1 :- $AllValuesFrom(?,?c1,?p1,?y) AND $AllValuesFrom(?,?c2,?p2,?y) AND ?p1<<?p2
```

Table 3.16: Axioms for AllValuesFrom Class Description in ObjectLogic

3.4.2.9 MaxCardinality Class Descriptions

The OWL 2 RL profile allows only *MaxCardinality* 0 and 1. We state *cls-maxc1* as a constraint, which is violated if the *_MaxCardinality(0, p, C)* class contains individuals *?u* having a value for the property *?p*.

Axiom	**RDF Syntax**
cls-maxc1	IF T(?x, owl:maxCardinality, "0"^^xsd:nonNegativeInteger) T(?x, owl:onProperty, ?p) T(?u, rdf:type, ?x) T(?u, ?p, ?y) THEN false
cls-maxc2	T(?x, owl:maxCardinality, "1"^^xsd:nonNegativeInteger) T(?x, owl:onProperty, ?p) T(?u, rdf:type, ?x) T(?u, ?p, ?y1) T(?u, ?p, ?y2) THEN T(?y1, owl:sameAs, ?y2)

ObjectLogic rules

```
@{'cls-maxc1'} !- $MaxCardinality(?,?x,0,?p,?c) AND ?u:?x AND ?u[?p->?y].
@{'cls-maxc2'}
!- $MaxCardinality(?,?x,1,?p,?c) AND ?u:?x AND ?u[?p->?y1] AND ?u[?p->?y2] AND NOT ?y1=?y2.
```

Table 3.17: Axioms for MaxCardinality Class Description in ObjectLogic

3.4.2.10 Union of Classes

The class construct *owl:unionOf* corresponds to the disjunction operator in predicate logic. The axiomatization of the union of classes is given by the axioms *cls-uni* and *scm-uni*.

Axiom	**RDF Syntax**
cls-uni	IF T(?c, owl:unionOf, ?x) LIST[?x, ?c1, ..., ?cn] T(?y, rdf:type, ?ci) THEN T(?y, rdf:type, ?c)
scm-uni	IF T(?c, owl:unionOf, ?x) LIST[?x, ?c1, ..., ?cn] THEN T(?c1, rdfs:subClassOf, ?c) T(?c2, rdfs:subClassOf, ?c) ... T(?cn, rdfs:subClassOf, ?c)
ObjectLogic rules	
@{'cls-uni'} ?y:?c :- $UnionOf(?,?c,?c1) AND ?y:?c1.	
@{'scm-uni'} ?c1::?c :- $UnionOf(?,?c,?c1).	

Table 3.18: Axioms for UnionOf Class Description in ObjectLogic

3.4.2.11 Intersection of Classes

The class construct *owl:intersectionOf* corresponds to the conjunction operator in predicate logic. The axiomatization of the intersection of classes is given by the axioms *cls-int1*, *cls-int2* and *scm-int*, as shown in the upper part of table 3.19.

We do not need *cls-int2* as it is a consequence of *scm-int* and *cax-sco*. Cf. *cls-int2*, the intersection *?c* of a list of classes is a subclass of every member class $?c_i$ in the list. Then according to the subclass axiom *@{'cax-sco'} ?x:?c2 :- ?c1::?c2 AND ?x:?c1*, every individual *?y* from *?c* must also be an instance of each list member $?c_i$, which implies that *cls-int2* holds true.

The axioms for axiomatizing the intersection of classes contain lists. They were splitted for better performance in sub-axioms for lists with less than 9 members plus one axiom defining the recursion:

If C_{1-9} is the identifier for *_IntersectionOf([*C_1*,* C_2*, ...,*C_9*])*, then this predicate is represented internally as

$IntersectionOf8(<id>, C_{1-9}*,* C_1*, ...,*C_7*, _IntersectionOf([*C_8*,* C_9*]))*

and

_IntersectionOf([C_8*,* C_9*])*

is given by

$IntersectionOf2(<id>, C_{8-9} *,*C_8*,* C_9*).*

Axiom	**RDF Syntax**
cls-int1	IF T(?c, owl:intersectionOf, ?x) LIST[?x, ?c1, ..., ?cn] T(?y, rdf:type, ?c1) T(?y, rdf:type, ?c2) ... T(?y, rdf:type, ?cn) THEN T(?y, rdf:type, ?c)
cls-int2	IF T(?c, owl:intersectionOf, ?x) LIST[?x, ?c1, ..., ?cn] T(?y, rdf:type, ?c) THEN T(?y, rdf:type, ?c1) T(?y, rdf:type, ?c2) ... T(?y, rdf:type, ?cn)
scm-int	IF T(?c, owl:intersectionOf, ?x) LIST[?x, ?c1, ..., ?cn] THEN T(?c, rdfs:subClassOf, ?c1) T(?c, rdfs:subClassOf, ?c2) ... T(?c, rdfs:subClassOf, ?cn)

ObjectLogic rules

```
@{'cls-int1-2'} ?y:?c :- $IntersectionOf2(?,?c,?c1,?c2) AND ?y:?c1 AND ?y:?c2.
@{'cls-int1-3'} ?y:?c :- $IntersectionOf3(?,?c,?c1,?c2,?c3) AND ?y:?c1 AND ?y:?c2 AND ?y:?c3.
..
@{'scm-int-2'} ?c::?c1 AND ?c::?c2 :- $IntersectionOf2(?,?c,?c1,?c2).
@{'scm-int-3'} ?c::?c1 AND ?c::?c2 AND ?c::?c3 :- $IntersectionOf3(?,?c,?c1,?c2,?c3).
..
```

Table 3.19: Axioms for IntersectionOf Class Description in ObjectLogic

3.4.2.12 Complement of a Class

The ObjectLogic equivalent of the *cls-com* rule is an ObjectLogic constraint which states that an individual *?x* cannot be an instance of a class *?c1* and of a class *?c2* if *?c1* is defined as the complement of *?c2*. Table 3.20 shows in the upper part the axiom for the complement of a class in RDF syntax and in the lower part the implementation as ObjectLogic constraint: an individual cannot be in a class and in its complement simultanously.

Axiom	**RDF Syntax**
cls-com	IF T(?c1, owl:complementOf, ?c2) T(?x, rdf:type, ?c1) T(?x, rdf:type, ?c2) THEN false

ObjectLogic rules

```
!- $ComplementOf(?,?c1,?c2) AND ?x:?c1 AND ?x:?c2.
```

Table 3.20: Axioms for ComplementOf Class Description in ObjectLogic

3.4.2.13 Algebraic and Inverse Properties

A property is symmetric, if from relating an individual *?x* with an individual *?y* follows that *?y* is also related with *?x*, as expressed by *prp-symp*. Symmetric properties have identical domain and range. A property is asymmetric, if there may not be a relation between *?x* and *?y* and *?y* and *?x* simultanously, as expressed by *prp-asymp*.

Axiom	RDF Syntax
prp-irp	IF T(?p, rdf:type, owl:IrreflexiveProperty) T(?x, ?p, ?x) THEN false
prp-symp	IF T(?p, rdf:type, owl:SymmetricProperty) T(?x, ?p, ?y) THEN T(?y, ?p, ?x)
prp-asyp	IF T(?p, rdf:type, owl:AsymmetricProperty) T(?x, ?p, ?y) T(?y, ?p, ?x) THEN false
prp-trp	IF T(?p, rdf:type, owl:TransitiveProperty) T(?x, ?p, ?y) T(?y, ?p, ?z) THEN T(?x, ?p, ?z)
prp-inv1	IF T(?p1, owl:inverseOf, ?p2) T(?x, ?p1, ?y) THEN T(?y, ?p2, ?x)
prp-inv2	IF T(?p1, owl:inverseOf, ?p2) T(?x, ?p2, ?y) THEN T(?y, ?p1, ?x)

ObjectLogic rules

```
// Irreflexive property
@{'prp-irp'} !- ?c[?p{_irreflexive}*=>()] AND ?x:?c[?p->?x].
// Symmetric property
@{'prp-symp'} ?y[?p->?x] :- ?c[?p{_symmetric}*=>()] AND ?x:?c[?p->?y].
// Asymmetric property
@{'prp-asyp'} ?c[?p{_asymmetric}*=>()] AND ?x:?c[?p->?y] AND ?y[?p->?x].
// Transitive property @{'prp-trp'} ?x[?p->?z] :- ?c[?p{_transitive}*=>()] AND ?x:?c[?p->?y] AND
?y:?c[?p->?z].
// Inverse property @{'prp-inv1'} ?y:?c2[?p2->?x] :- ?c1[?p1 {inverseOf(?p2)} *=> ?c2] AND
?x:?c1[?p1->?y].
@{'prp-inv2'} ?y:?c1[?p1->?x] :- ?c1[?p1 {inverseOf(?p2)} *=> ?c2] AND ?x:?c2[?p2->?y].
```

Table 3.21: Axioms for Algebraic and Inverse Properties

3.4.2.14 Functional and Inverse Functional Properties

For functional and inverse functional properties we provide a special handling, addressing thus performance issues of *owl:sameAs*. Instead of inferring *owl:sameAs*, we state a constraint violation if a functional property has two different values for the same individual or an inverse functional property has the same value for two different individuals. Our interpretation of *prp-fp* states a constraint violation each time a property *?p* is defined as functional *?C[?p{0:1}*=>()]* and there is an individual *?x* having two values *?y1, ?y2* for *?p* which are not related through the *_SameIndividual* relation. Our interpretation of *prp-ifp* states a constraint violation each time a property *?p* is defined as inverse functional *?C[?p{inverseFunctional}*=>()]* and there are two different individuals *?x1, ?x2* having the same value *?y* for *?p*.

Axiom	RDF Syntax
prp-fp	IF T(?p, rdf:type, owl:FunctionalProperty) T(?x, ?p, ?y1) T(?x, ?p, ?y2) THEN T(?y1, owl:sameAs, ?y2)
prp-ifp	IF T(?p, rdf:type, owl:InverseFunctionalProperty) T(?x1, ?p, ?y) T(?x2, ?p, ?y) THEN T(?x1, owl:sameAs, ?x2)

ObjectLogic rules

```
@{'prp-fp'} !- ?C[?p{0:1}*=>()] AND
?x[?p->?y1] AND ?x[?p->?y2] AND NOT _SameIndividual(?y1, ?y2).
@{'prp-ifp'} !- ?C[?p{inverseFunctional}*=>()] AND
?x1[?p->?y] AND ?x2[?p->?y] AND NOT _SameIndividual(?x1, ?x2).
```

Table 3.22: Axioms for Functional and Inverse Functional Properties

3.4.2.15 Equivalent and Disjoint Properties

We provide a special handling for *prp-pwd* and *scm-eqp2*, which we implement as constraint in order to avoid inferring *owl:sameAs*.

Axiom	RDF Syntax
prp-eqp1	IF T(?p1, owl:equivalentProperty, ?p2) T(?x, ?p1, ?y) THEN T(?x, ?p2, ?y)
prp-eqp2	IF T(?p1, owl:equivalentProperty, ?p2) T(?x, ?p2, ?y) THEN T(?x, ?p1, ?y)
prp-pwd	IF T(?p1, owl:propertyDisjointWith, ?p2) T(?x, ?p1, ?y) T(?x, ?p2, ?y) THEN false
scm-eqp1	IF T(?p1, owl:equivalentProperty, ?p2) THEN T(?p1, rdfs:subPropertyOf, ?p2) T(?p2, rdfs:subPropertyOf, ?p1)
scm-eqp2	IF T(?p1, rdfs:subPropertyOf, ?p2) T(?p2, rdfs:subPropertyOf, ?p1) THEN T(?p1, owl:equivalentProperty, ?p2)

ObjectLogic rules

```
@{'prp-eqo1'} ?x[?p2 -> ?y] :- _EquivalentProperties(?p1,?p2) AND ?x[?p1->?y].
@{'prp-eqp2'} ?x[?p1 -> ?y] :- _EquivalentProperties(?p1,?p2) AND ?x[?p2 ->?y].
@{'prp-pwd'} !- _DisjointProperties(?p1,?p2) AND ?x[?p1->?y] AND ?x[?p2->?y].
@{'scm-eqp1'} ?p1<<?p2 AND ?p2<<?p1 :- _EquivalentProperties(?p1,?p2).
@{'scm-eqp2'} !- ?p1<<?p2 AND ?p2<<?p1 AND NOT _EquivalentProperties(?p1, ?p2).
```

Table 3.23: Axioms for Equivalent and Disjoint Properties

3.4.2.16 Property Chains

The property chain axiom *prp-spo2* has been split for performance reasons in the axioms *prp-spo2-2* (for a chain given by two properties) to *prp-spo2-8* (for a chain given by eight properties). Larger property chains are computed recursively.

Axiom	RDF Syntax
prp-spo1	IF T(?p1, rdfs:subPropertyOf, ?p2) T(?x, ?p1, ?y) THEN T(?x, ?p2, ?y)
prp-spo2	IF T(?p, owl:propertyChainAxiom, ?x) LIST[?x, ?p1, ..., ?pn] T(?u1, ?p1, ?u2) T(?u2, ?p2, ?u3) ... T(?un, ?pn, ?un+1) THEN T(?u1, ?p, ?un+1)

ObjectLogic rules

```
@{'prp-spo1'} ?x[?p2->?y] :- ?p1<<?p2 AND ?x[?p1->?y].
@{'prp-spo2-2'}
?u1[?p->?u3] :- $PropertyChain2(?,?p,?p1,?p2) AND ?u1[?p1->?u2] AND ?u2[?p2->?u3].
@{'prp-spo2-3'}
?u1[?p->?u4] :- $PropertyChain3(?,?p,?p1,?p2,?p3) AND ?u1[?p1->?u2] AND ?u2[?p2->?u3] AND
?u3[?p3->?u4].
...
```

Table 3.24: Axioms for Property Chains in ObjectLogic

3.4.2.17 Keys

A set of properties is a key for a class, if no two named instances of the class coincide on all values of the properties. This language feature is interesting when mapping database tables with primary keys consisting of more than one attribute to a concept in an ontology. We provide a special handling for *prp-key*, which we implement as constraint in order to avoid inferring *owl:sameAs*. The key axiom *prp-key* is split for performance reasons in the constraints *prp-key-1* (for a key given by one property) to *prp-key-8* (for a key given by eight properties). Keys consisting of more than eight properties are rewritten recursively.

Axiom	**RDF Syntax**
prp-key	IF T(?c, owl:hasKey, ?u) LIST[?u, ?p1, ..., ?pn] T(?x, rdf:type, ?c) T(?x, ?p1, ?z1) ... T(?x, ?pn, ?zn)T(?y, rdf:type, ?c) T(?y, ?p1, ?z1) ... T(?y, ?pn, ?zn) THEN T(?x, owl:sameAs, ?y)
ObjectLogic rules	
@{'prp-key-1'} !- $HasKey1(?,?c,?p) AND ?x:?c AND ?x[?p->?r] AND ?y:?c AND ?y[?p->?r] AND NOT _SameIndividual(?x,?y).	
@{'prp-key-2'} !- $HasKey2(?,?c,?p1,?p2) AND ?x:?c AND ?x[?p1->?r1,?p2->?r2] AND ?y:?c AND ?y[?p1->?r1,?p2->?r2] AND NOT _SameIndividual(?x,?y).	
@{'prp-key-3'} !- $HasKey3(?,?c,?p1,?p2,?p3) AND ?x:?c AND ?x[?p1->?r1,?p2->?r2,?p3->?r3] AND ?y:?c AND ?y[?p1->?r1,?p2->?r2,?p3->?r3] AND NOT _SameIndividual(?x,?y).	
...	

Table 3.25: Axioms for Keys in ObjectLogic

3.4.3 Tests

Tests for the OWL 2 RL in ObjectLogic implementation consisted in a series of syntactical translation tests, followed by semantical tests for the axiomatization part. Before including the OWL 2 RL functionality in the product, a profile identification test and a benchmark test will be performed.

3.4.3.1 Semantic Tests

Semantic tests specifically address the functionality of OWL entailment checkers. Each test case of this type specifies necessary requirements that must be satisfied by any entailment checker that meets the according conformance conditions. Semantic tests specify one or more OWL 2 ontology documents and check semantic conditions defined with respect to abstract structures obtained from the ontology documents, typically via a parsing process. The abstract structure is an OWL 2 ontology. Let *Ont(d)* be the abstract structure obtained from the ontology document *d*.

Entailment tests as specified by the OWL 2 Conformance and Test Cases [53] specify two ontology documents: a premise ontology document *d1* and a conclusion ontology document *d2* where *Ont(d1)* entails *Ont(d2)* with respect to the specified semantics. If provided with inputs *d1* and *d2* a conforming entailment checker should return True, it should not return Unknown, and it must not return False. Non-Entailment tests (or negative entailment tests) specify two ontology documents: a premise ontology document *Ont(d1)* and a non-conclusion ontology *Ont(d2)* where *Ont(d1)* does not entail *Ont(d2)* with respect to the specified semantics. The tests we performed cover the OWL 2 RL Test Cases [1].

```
//Schema
_IntersectionOf([_HasValue(p,v),_HasValue(q,w)])::C.
a[p->v].a[q->w].b[p->v].c[q->w].
d[p->z].d[q->z].
//Axioms
@{'cls-hv1'} ?u[?p->?y] :- $HasValue(?,?x,?p,?y) AND ?u:?x.
@{'cls-hv2'} ?u:?x :- $HasValue(?,?x,?p,?y) AND ?u[?p->?y].
// ...
@{'cls-int1-2'} ?y:?c :- $IntersectionOf2(?,?c,?c1,?c2) AND ?y:?c1 AND ?y:?c2.
@{'cls-int1-3'} ?y:?c :- $IntersectionOf3(?,?c,?c1,?c2,?c3) AND ?y:?c1 AND ?y:?c2 AND ?y:?c3.
@{'scm-int-2'} ?c::?c1 AND ?c::?c2 :- $IntersectionOf2(?,?c,?c1,?c2).
@{'scm-int-3'} ?c::?c1 AND ?c::?c2 AND ?c::?c3 :- $IntersectionOf3(?,?c,?c1,?c2,?c3).
// ...
// Query
?- ?i:C.
//Expected results
a
```

Table 3.26: Entailment test for the production IntersectionOf-HasValue

3.4.3.2 Tests with Optimization Switches

OntoBroker has a number of performance-related configuration switches which improve performance significantly depending on the type of the rule. If configuration option 'ConceptNamesGround' is on, rules must not have variables in concept positions. If 'AttributeNamesGround' is used, no rule may have a head containing a variable for the relation name as these cannot be reinterpreted during compile time.

The configuration switch ConceptNamesGround may be used for equality, hasKey, complementOf, maxCardinality and disjointClasses. On the other hand, most of the axioms for complex classes have variables in concept positions which excludes the usage of ConceptNamesGround. The configuration switch AttributesNamesGround may be used for equality, hasKey, complementOf, disjointClasses, equivalentClasses, functional properties, but not for any axiom containing the subproperty relationship, and thus not for hasValue, someValuesFrom, allValuesFrom.

[1] OWL Test Cases http://owl.semanticweb.org/page/OWL_2_Test_Cases

```
//Schema
_UnionOf([_UnionOf([E,F]),_IntersectionOf([E,F]),
_OneOf([a,b]),_HasValue(p,u),_SomeValuesFrom(q,D)])::C.
// Instances
e:E. f:F. x[p->u]. y[q->v]. v:D. z[q->w].
// Axioms
@{'scm-int-2'} ?c::?c1 AND ?c::?c2 :- $IntersectionOf2(?,?c,?c1,?c2).
@{'scm-int-3'} ?c::?c1 AND ?c::?c2 AND ?c::?c3 :- $IntersectionOf3(?,?c,?c1,?c2,?c3).
@{'cls-int1-2'} ?y:?c :- $IntersectionOf2(?,?c,?c1,?c2) AND ?y:?c1 AND ?y:?c2.
@{'cls-int1-3'} ?y:?c :- $IntersectionOf3(?,?c,?c1,?c2,?c3) AND ?y:?c1 AND ?y:?c2 AND ?y:?c3.
// ...
@{'cls-uni'} ?y:?c :- $UnionOf(?,?c,?ci) AND ?y:?ci.
@{'scm-uni'} ?ci::?c :- $UnionOf(?,?c,?ci).
@{'cls-hv1'} ?u[?p->?y] :- $HasValue(?,?x,?p,?y) AND ?u:?x.
@{'cls-hv2'} ?u:?x :- $HasValue(?,?x,?p,?y) AND ?u[?p->?y].
@{'scm-hv'} ?c1::?c2 :- $HasValue(?,?c1,?p1,?y) AND $HasValue(?,?c2,?p2,?y) AND ?p1«?p2.
@{'cls-oo'} ?yi:?c :- $OneOf(?,?c,?yi).
@{'cls-svf'} ?u:?x :- $SomeValuesFrom(?,?x,?p,?y) AND ?u[?p->?v] AND ?v:?y.
@{'scm-svf1'} ?c1::?c2 :- $SomeValuesFrom(?,?c1,?p,?y1) AND $SomeValuesFrom(?,?c2,?p,?y2) AND
?y1::?y2.
@{'scm-svf2'} ?c1::?c2 :- $SomeValuesFrom(?,?c1,?p1,?y) AND $SomeValuesFrom(?,?c2,?p2,?y) AND
?p1«?p2.
// Query ?- ?x:C.
//Results a,b,e,f,x,y
```

Table 3.27: Entailment test for the combination of UnionOf with all admissible subclass expressions

3.5 Conclusions

The OWL 2 RL in ObjectLogic implementation is an example of a tight coupling of rules and ontologies. Ontology and rule language is the same: ObjectLogic. We have implemented the OWL syntactical constructs as ObjectLogic predicates and added the semantics of the RL profile as given by its defining rule set in the form of ObjectLogic rules.

Some of the OWL features - especially *owl:sameAs* - were found to be performance-critical. The performance issues were addressed by means of a specific interpretation for equality, functional and inverse functional properties, equivalent and disjoint classes. We use constraints and a partial axiomatization of equality. In this we have a deviation from the standard OWL semantics. This should not be critical, since we are able to satify the higher prioritized features from industrial usecases while maintaining performance requirements.

Finally, we learned that although part of the OWL functionality can be expressed by core ObjectLogic rules, other OWL specific features like *owl:sameAs* offer new and promising ways of reasoning and inferring facts from existing knowledge bases.

3.6 OWL 2 RL in ObjectLogic Syntax Reference

The following section enumerates the predicates introduced in ObjectLogic as equivalents of the OWL syntactical elements. We present them in the form of tables. The tables for the constructs without special internal representation have as first column the name of the language feature, as second column the OWL 2 functional syntax, as third column the corresponding ObjectLogic Syntax. The tables for features with an internal ObjectLogic representation (boolean class expressions, datarange expressions, property chains, keys) have as first column the functional syntax, as second column the ObjectLogic syntax and as third column the ObjectLogic internal representation. Note that the internal representation is used in particular for the axiomatization of language constructs containing lists.

PREDEFINED AND NAMED CLASSES

Feature	**Functional Syntax**	**ObjectLogic Syntax**
named class	CN	CN
universal class	owl:Thing	owl#Thing
empty class	owl:Nothing	owl#Nothing

DATATYPE DEFINITIONS

Feature	**Functional Syntax**	**ObjectLogic Syntax**
datatype definition	DatatypeDefinition(DN D)	_DatatypeDefinition(DN, D)

ASSERTIONS

Feature	**Functional Syntax**	**ObjectLogic Syntax**
equality	SameIndividual($a_1 \ldots a_n$)	_SameIndividual(a_1, a_2) _SameIndividual1([a_1, $\ldots a_n$]).
inequality	DifferentIndividuals(a_1 a_2)	_DifferentIndividuals(a_1, a_2).
pairwise inequality	DifferentIndividuals($a_1 \ldots a_n$)	_DifferentIndividuals1([a_1, $\ldots a_n$])
class assertion	ClassAssertion (C a)	a:C.
positive object property assertion	ObjectPropertyAssertion(PN a_1 a_2)	a_1[PN -> a_2].
positive data property assertion	DataPropertyAssertion(R a v)	a[R ->v].
negative object property assertion	NegativePropertyAssertion(P $a_1 a_2$)	_NegativePropertyAssertion(P, a_1, a_2).
negative data property assertion	NegativePropertyAssertion(R a v)	_NegativePropertyAssertion(R, a, v).

CLASS EXPRESSION AXIOMS

Feature	**Functional Syntax**	**ObjectLogic Syntax**
subclass	SubClassOf(C_1 C_2)	C_1::C_2.
equivalent classes	EquivalentClasses(C_1 C_2)	_EquivalentClasses(C_1, C_2).
equivalent classes	EquivalentClasses(C_1 ... C_n)	_EquivalentClasses1([C_1,..., C_n]).
disjoint classes	DisjointClasses(C_1 C_2)	_DisjointClasses(C_1,C_2).
pairwise disjoint classes	DisjointClasses(C_1 ... C_n)	_DisjointClasses1([C_1,..., C_n]).

BOOLEAN CONNECTIVES

Functional Syntax	**ObjectLogic Syntax**	**ObjectLogic Internal Facts**
intersection		
ObjectIntersectionOf(C_1 ... C_n)	_IntersectionOf([C_1, C_2]) ... _IntersectionOf([C_1,C_2,...,C_8]) ... _IntersectionOf([C_1,C_2,...,C_9])	$IntersectionOf2(<id>,C,$C_1$,$C_2$) ... $IntersectionOf8(<id>, C, C_1, ..., C_8) ... $IntersectionOf8(<id>, C, C_1,..., C_7, _IntersectionOf([C_8,C_9]))
union		
ObjectUnionOf(C_1 ... C_n)	_UnionOf([C_1, ... C_n])	$UnionOf(<id>,C, C_1) ... $UnionOf(<id>,C, C_n)
complement		
ObjectComplementOf(C)	_ComplementOf(C)	$ComplementOf(<id>, _ComplementOf(C), C)
enumeration		
ObjectOneOf(a_1 ... a_n)	_OneOf([a_1, ..., a_n])	$OneOf(<id>, C, a_1) ... $OneOf(<id>, C, a_n)

PROPERTY EXPRESSIONS

Functional Syntax	**ObjectLogic Syntax**	**ObjectLogic Internal Facts**
named object property		
PN	PN	
universal object property		
owl:topObjectProperty	owl#topObjectProperty	
empty object property		
owl: bottomObjectProperty	owl#bottomObjectProperty	
inverse property		
ObjectInverseOf(PN)	_InverseOf(PN)	$InverseOf(<id>, _InverseOf(PN), PN)

DATARANGE EXPRESSIONS

Functional Syntax	**ObjectLogic Syntax**	**ObjectLogic Internal Facts**
data range intersection		
DataIntersectionOf(D_1 ... D_n)	_IntersectionOf([D_1, D_2])	$IntersectionOf2(<id>, _IntersectionOf([D_1, D_2]), D_1, D_2)
	...	...
	_IntersectionOf([D_1, D_2, ...,D_8])	$IntersectionOf8(<id>, D, D_1, ...,D_8)
	_IntersectionOf([D_1, D_2, ...,D_9])	$IntersectionOf8(<id>, D, D_1, ...,D_7, _IntersectionOf([D_8, D_9]))
data range union		
DataUnionOf(D_1 ... D_n)	_UnionOf([D_1, ...,D_n])	$UnionOf(<id>,D, D_1) ... $UnionOf(<id>,D,$D_n$)
literal enumeration		
DataOneOf(v_1 ... v_n)	_OneOf(v_1, ... v_n)	$OneOf(<id>,_OneOf([$v_1$, ...,$v_n$]), v_1) ... $OneOf(<id>,_OneOf([$v_1$, ...,$v_n$]), v_n)
datatype restriction		
DataTypeRestriction(DN f_1 v_1 ... f_n v_n)	_DatatypeRestriction(DN, [f_1, v_1, ... f_n, v_n])	none

PROPERTY CHAINS

Functional Syntax	**ObjectLogic Syntax**	**ObjectLogic Internal Facts**
SubObjectPropertyOf (ObjectPropertyChain(P_1 ... P_n) P)	_PropertyChain([P_1, P_2])	$PropertyChain2(<id>,P,$P_1$, P_2)
	_PropertyChain([P_1, P_2, ...,P_8])	$PropertyChain8(<id>,P,$P_1$, ...,$P_8$)
	_PropertyChain([P_1, P_2, ...,P_9])	$PropertyChain8(<id>, _ PropertyChain ([P_1,..., P_9]), P_1, ...,P_7, _PropertyChain ([P_8, P_9])) where _PropertyChain([P_8, P_9]) is given by $PropertyChain2(<id>, _PropertyChain([P_8, P_9],P_8, P_9)

OBJECT PROPERTY AXIOMS

Feature	**Functional Syntax**	**ObjectLogic Syntax**
subproperty	SubPropertyOf(P_1 P_2)	$\mathrm{P}_1 << \mathrm{P}_2$
property domain	ObjectPropertyDomain(P C)	C[P*=>rdfs#Resource]
property range	ObjectPropertyRange(P C)	
equivalent properties	EquivalentObjectProperties($\mathrm{P}_1 \ldots \mathrm{P}_n$)	_EquivalentProperties($\mathrm{P}_1, \ldots \mathrm{P}_n$)
disjoint properties	DisjointObjectProperties(P_1 P_2)	_DisjointProperties(P_1, P_2)
pairwise disjoint properties	DisjointObjectProperties($\mathrm{P}_1 \ldots \mathrm{P}_n$)	_DisjointProperties1([$\mathrm{P}_1, \ldots, \mathrm{P}_n$])
inverse properties	InverseObjectProperty(P_1 P_2)	owl#Thing[P_1{inverseOf(P_2)*=>()]

FUNCTIONAL AND INVERSE FUNCTIONAL PROPERTIES

Feature	**Functional Syntax**	**ObjectLogic Syntax**
functional property	FunctionalProperty(P)	owl#Thing[P{0:1}*=>()]
inverse functional property	InverseFunctionalProperty(P)	owl#Thing[P{inverseFunctional}*=>()]

ALGEBRAIC PROPERTIES

Feature	**Functional Syntax**	**ObjectLogic Syntax**
irreflexive property	IrreflexiveObjectProperty(P)	owl#Thing [P{irreflexive} *=> ()].
symmetric property	SymmetricObjectProperty(P)	owl#Thing [P{symmetric} *=> ()].
asymmetric property	AsymmetricObjectProperty(P)	owl#Thing [P{asymmetric} *=> ()].
transitive property	TransitiveObjectProperty(P)	owl#Thing [P{transitive} *=> ()].

KEYS

Functional Syntax	**ObjectLogic Syntax**	**ObjectLogic Internal Facts**
HasKey (C ($\mathrm{P_1}$...$\mathrm{P_m}$)) m>0	_HasKey(C, [$\mathrm{P_1}$])	$HasKey1(<id>, C, $\mathrm{P_1}$)
	_HasKey(C, [$\mathrm{P_1}$,$\mathrm{P_2}$])	$HasKey2(<id>, C, $\mathrm{P_1}$,$\mathrm{P_2}$)
	...	...
	_HasKey(C, [$\mathrm{P_1}$,..,$\mathrm{P_8}$])	$HasKey8(<id>, C, $\mathrm{P_1}$,..,$\mathrm{P_8}$)
	_HasKey(C, [$\mathrm{P_1}$,..,$\mathrm{P_9}$])	$HasKeyAnyArity(<id>,C, [$\mathrm{P_1}$,..,$\mathrm{P_9}$])

DATA PROPERTY AXIOMS

Feature	**Functional Syntax**	**ObjectLogic Syntax**
subproperty	SubPropertOf($\mathrm{R_1}$ $\mathrm{R_2}$)	$\mathrm{R_1} << \mathrm{R_2}$
property domain	DataPropertyDomain(R C)	C[R*=>rdfs#Resource]
property range	DataPropertyRange(R C)	owl#Thing[R*=>D]
equivalent properties	EquivalentDataProperties($\mathrm{R_1}$ ... $\mathrm{R_n}$)	_EquivalentProperties1([$\mathrm{R_1}$ ... $\mathrm{R_n}$])
disjoint properties	DisjointDataProperties($\mathrm{R_1}$ $\mathrm{R_2}$)	_DisjointProperties(R_1 R_2)
pairwise disjoint properties	DisjointDataProperties($\mathrm{R_1}$ ... $\mathrm{R_n}$)	_DisjointProperties1([R_1 ... R_n])

INDIVIDUALS & LITERALS

Feature	**Functional Syntax**	**ObjectLogic Syntax**
named individual	aN	aN
anonymous individual	_:a	_#‘a‘
literal (datatype value)	"abc"^^DN	"abc"^^DN

DECLARATIONS

Feature	**Functional Syntax**	**ObjectLogic Syntax**
datatype	Declaration(Datatype(DN))	DN:owl#DataProperty
object property	Declaration(ObjectProperty(PN))	PN:owl#ObjectProperty
data property	Declaration(DataProperty(R))	R:owl#DataProperty
annotation property	Declaration(AnnotationProperty(A))	A:owl#AnnotationProperty
named individual	Declaration(NamedIndividual(aN))	aN:owl#NamedIndividual

Chapter 4

Combining Production Rules and Ontologies

The main goal of this work is to combine production rules and ontologies. The aim of such a combination is to use an ontology to describe the vocabulary used in the rules. The language we consider for the description of such ontologies is the W3C standard OWL [47], which is a declarative, logical language, based on Description Logics family of knowledge representation languages [8]. In contrast, the language of production rules has an operational semantics, i.e., the semantics is given by the algorithm used to execute the rules.

The integration of two knowledge representation languages with such different semantics requires a solid theoretical foundation in order to understand the implications of the combination – both semantical and operational – on a deep level. In this chapter we bridge the gap between the semantics of production rules and ontologies.

In Section 4.1 we present a *loosely-coupling* approach, where the semantics of the production rules and the ontologies are decoupled: the interaction between the two semantics is based on entailment. In Section 4.2 we present a *tightly-coupling* approach based on fixedpoint logic, where there is a stronger potential interaction between the rules and ontologies. In particular, in this section we provide a new semantics for production systems augmented with DL ontologies, and discuss several issues that arise when combining production rules (PR) with description logic (DL) ontologies. In addition, we extend the FPL embedding presented in deliverable D3.2 to cover the semantic of the combination. Finally, we present conclusions in Section 4.3.

4.1 Loose Coupling of Production Rules and Ontologies

There exist different variants of descriptions of Production Rules (PRs). They differ mainly in descriptions of conditions and actions. Regularly conditions corresponds to

sets of patterns, i.e. sets of templates of nested objects, that are being matched against elements of a working memory, that are called facts. A working memory represents data and exploit a Closed World Assumption (CWA). If a condition of a PR is successfully matched, an action of the rule can be applied. Actions regularly perform operations, changing a working memory.

There are different ways of representing a working memory, different patterns matching and action performance algorithms (depending on the way facts in a working memory are represented) were developed (see, for example, [21]). In this section we try to combine ontologies with PRs. Together with a working memory we consider an ontology, imposing restrictions and giving a structure to the working memory. And while checking conditions and performing actions, the ontology should be taken into account.

In contrast to databases, ontologies exploit an Open World Assumption (OWA). So, as far as we use ontologies, we take an OWA in our approach, that fundamentally differs it from other known approaches to PRs descriptions.

To describe an ontology and a working memory, we use a First-Order Logic (FOL) in general and a Description Logics (DLs) in particular and define conditions and actions using capabilities of these logics. We develop here different kinds of semantics of a condition satisfiability (patterns matching) and an action performance using some known results of research (for example, in the area of DL ontologies) and adapting them to our case.

The rest of the section is organized as follows. In Section 4.1.1 we introduce FOL and DLs and give definitions to main components of PRs using FOL. Section 4.1.2 presents a semantics of conditions and actions of production rules over a First-Order Knowledge Base. In Section 4.1.2.1 we consider conditions of PRs and introduce two kinds of condition satisfiability, giving different semantics to this notion using a logical entailment. In Section 4.1.2.2 we developed two different semantics of actions, changing a working memory. One of them (formula-based) is syntactical and based on a direct changing of a working memory by adding and removing of some facts, while the second one (Possible Models Approach) is based on changing of common models of an ontology and a working memory, considering these models as a possible states-of-affairs of the application. Section 4.1.3 considers some peculiarities of semantics of PRs over DL ontologies. In section 4.1.3.1 we show that in this particular case (when we consider a DL ontology as a KB) definitions of a satisfiability of a set of patterns can be simplified. In section 4.1.3.2.1 on the base of algorithms, computing an ABox update (or its approximation) of an ontology, we introduce a procedure for computing the result of an action, i.e. a new working memory for some DLs from a $DL-Lite$ family. In Section 4.1.3.2.2 we introduce an alternative semantics for update and erasure (i.e. action performance) in DLs and prove its equivalence to the Possible Models Approach semantics, introduced in Section 4.1.2.4.

4.1.1 Preliminaries

Here we use FOL in general to define FO KBs that are FO theories and DLs to define DL KBs (ontologies) that are finite sets of DL assertions. So, we introduce these notions first in Sections 4.1.1.1 and 4.1.1.2 respectively. In Section 4.1.1.3 we define PRs.

4.1.1.1 First-Order Logic

We consider a set $\mathcal{C}$ of constants, a set $\mathcal{P}$ of predicate symbols and a set $\mathcal{X}$ of variables. Each predicate symbol has a fixed *arity* $n \geqslant 0$ (number of its arguments). A pair $(\mathcal{C}, \mathcal{P})$ is called *signature*. We also include an equality symbol '=', that syntactically behaves like a binary predicate and has special semantics. *Term* is either a variable or a constant.

An *atomic formula* is any expression of the form $P(t_1, \ldots, t_n)$, where $t_1, \ldots, t_n$ are terms and P is an n-ary predicate symbol. Complex formulas are constructed as usual by using *logical connectives* $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, *quantifiers* $\forall$, $\exists$ and *punctuation symbols* '(', ')',',','. Set of all formulas over a signature $\Sigma = (\mathcal{C}, \mathcal{P})$ constitutes a *first-order language* $\mathcal{L}(\mathcal{C}, \mathcal{P})$.

A variable occurrence in a formula is called *free* if it does not occur in the scope of any quantifier, otherwise it is called *bound*. A *sentence* is a formula with no free-variable occurrences.

Let G be a formula. We denote the set of all variables of the formula G with $Var(G)$ and the set of all variables, that occur free in the formula G with $FreeVar(G)$. This extends to the sets of formulas in the natural way.

A *substitution* is a mapping $\sigma : \mathcal{X} \mapsto \mathcal{C} \cup \mathcal{X}$. Also we set $x\sigma = \sigma(x)$ for any variable x, $c\sigma = c$ for any constant c, $P\sigma = P$ for any nullary predicate symbol P and $[P(t_1, \ldots, t_n)]\sigma = P(t_1\sigma, \ldots, t_n\sigma)$, where $t_1, \ldots, t_n$ are terms and P is n-ary predicate symbol. Substitution extends to sets of formulas in a natural way.

We give the usual semantics in terms of interpretations, and main terms accompanying this notion are supposed to be defined as usual. Let us remind some of them, that we directly use here.

An *interpretation* of a language $\mathcal{L}(\mathcal{C}, \mathcal{P})$ is a pair $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$, where $\Delta^{\mathcal{I}}$ is a nonempty set, called *domain* and $\cdot^{\mathcal{I}}$ is a mapping, called an *interpretation function* that associates to every constant $c \in \mathcal{C}$, some $c^{\mathcal{I}} \in \Delta^{\mathcal{I}}$ and to every $n-$ary predicate symbol $P \in \mathcal{P}$, some $n-$ary relation $P^{\mathcal{I}} \in (\Delta^{\mathcal{I}})^n$.

A *variable assignment* in an interpretation $\mathcal{I}$ is a mapping A which assigns to every variable $x \in \mathcal{X}$, an element $x^A \in \Delta^{\mathcal{I}}$ of the domain. A variable assignment A' is an $x-$*variant of* A if $y^A = y^{A'}$ for every variable $y \in \mathcal{X}$ such that $y \neq x$.

Given an interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$, a variable assignment A, and a term $t \in \mathcal{X} \cup \mathcal{C}$, $t^{\mathcal{I},A}$ is defined as follows: $t^{\mathcal{I},A} = t^A$ if $t \in \mathcal{X}$, and $t^{\mathcal{I},A} = t^{\mathcal{I}}$, if $t \in \mathcal{C}$.

We say that an interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ *satisfies* an atomic formula $P(t_1, \ldots, t_n)$

relative to a variable assignment A, where $t_1, \ldots, t_n$ are terms and denote it $\mathcal{I}, A \models P(t_1, \ldots, t_n)$, if $(t_1^{\mathcal{I},A}, \ldots, t_n^{\mathcal{I},A}) \in P^{\mathcal{I}}$. And $\mathcal{I}, A \models t_1 = t_2$, if $t_1^{\mathcal{I},A} = t_2^{\mathcal{I},\mathcal{A}}$, where t_1 and t_2 are terms. This is extended to arbitrary formulas as usual.

Given a formula ϕ, an interpretation $\mathcal{I}$ is a *model* of ϕ, denoted $\mathcal{I} \models \phi$, if $\mathcal{I}, A \models \phi$, for every variable assignment A.

Let **S** be some set of sentences. We say that some interpretation *is a model of* **S**, if it is a model of each sentence from **S**. We denote a set of all the models of **S** by $Mod(\mathbf{S})$. A set **S** of sentences is *satisfiable* (*consistent*), if there exists at least one model of it. A set **S** of sentences *logically entails* a sentence s, denoted $\mathbf{S} \models s$, if every model of **S** is also model of s.

4.1.1.2 Description Logics

DL languages may be considered as proper subsets of FO language. In DLs only predicates of arity 1 and 2 are considered. Predicates of arity 1 denoting sets of individuals are called *atomic concepts*, and predicates of arity 2 denoting binary relations between individuals are called *atomic roles*. Using atomic concepts and roles one can build complex descriptions of concepts and roles. The language for building descriptions is specified for each DL, and different DLs are distinguished by their description languages [8]. The common rule is that descriptions (usually concept descriptions) are built from atomic concepts and roles by using DL constructors, such as $\sqcap, \sqcup, \neg, \exists, \forall$. For example, in basic DL $\mathcal{ALC}$ concepts (concept descriptions) are defined as follows:

$$C ::= \bot \mid A \mid \neg C \mid C \sqcap D \mid C \sqcup D \mid \exists R.C \mid \forall R.C,$$

where A denotes an atomic concept, R an atomic role, and C, D are concepts. By adding or removing some constructs we can get other DLs. For instance, if we add an inverse role constructor R^- (R is an atomic role) to $\mathcal{ALC}$, we will get a DL $\mathcal{ALCI}$.

A DL KB, i.e. an ontology, consists of two components, a TBox and an ABox. A TBox represents the terminology by a set of *assertions*, that are usually *general concept inclusion axioms* $C \sqsubseteq D$, where C and D are concepts (sometimes we say that C is a left-hand side (LHS) and D is a right-hand side (RHS) of the general concept inclusion axiom $C \sqsubseteq D$). Allowing another TBox assertions or restricting existing ones can also result in another DLs. For example, adding to $\mathcal{ALC}$ a possibility to define hierarchies of roles through *role inclusions axioms* $R \sqsubseteq S$ (R and S are roles) in the TBox, we get a DL $\mathcal{ALCH}$. There are also DLs, where only restricted general concept inclusion axioms can be used, e.g. in many DLs from a $DL-Lite$ family negation ($\neg$) can occur only in a RHS of a general concept inclusion axiom.

An ABox introduces knowledge about named individuals denoted by constants. It contains *concept assertions* $A(a)$ and *role assertions* $R(a, b)$, where A is a concept, R is a role and a, b are constants. And again ABox assertions can be extended or restricted. For example, in $DL-Lite_S$ assertions of the form $C(z)$, where z is a variable, are allowed.

DL Constructor	FOL formula
A	$\phi_A(x)$
$\neg C$	$\neg\phi_C(x)$
$C \sqcap D$	$\phi_C(x) \wedge \phi_D(x)$
$C \sqcup D$	$\phi_C(x) \vee \phi_D(x)$
$C \sqsubseteq D$	$\forall x(\phi_C(x) \rightarrow \phi_D(x))$
$\exists R.C$	$\exists y(\phi_R(x,y) \wedge \phi_C(y))$
$\forall R.C$	$\forall y(\phi_R(x,y) \rightarrow \phi_C(y))$
$A(a)$	$\phi_A(a)$
$R(a,b)$	$\phi_R(a,b)$

Table 4.1: Translation from $\mathcal{ALC}$ to FOL.

This assertion states, that there exists an object, denoted by the variable z, that is an instance of the general concept C. Concerning restrictions of ABox assertions, in many logics from the $DL-Lite$ family concept assertions to negated concepts are not allowed.

In classical DLs all the assertions in the TBox and in the ABox can be identified with sentences in FOL (see Table 4.1 for $\mathcal{ALC}$ concepts, for example). And we transfer the usual semantics in terms of interpretations from FOL to DLs.

4.1.1.3 Production Rules

In this section we introduce main terms we will use in conjunction with production rules and a production rule itself.

A *fact type* is an n-ary predicate symbol ($n \geqslant 0$). Let P be a fact type of arity n ($n \geqslant 0$) and $a_1, \ldots, a_n$ be constants, then the atomic sentence $P(a_1, \ldots, a_n)$ is called *fact*.

A *pattern* is any formula. Let us consider a set of patterns. Each pattern can be *positive* or *negative*. The fact, that some pattern p_i from the set of patterns is positive or negative is denoted by marking it as p_i^+ or p_i^- respectively. We also denote finite sets of positive and negative patterns by $\mathbf{p}^+$ and $\mathbf{p}^-$ correspondingly. That is, $\mathbf{p}^+ = \{p_1^+, \ldots, p_k^+\}$ and $\mathbf{p}^- = \{p_1^-, \ldots, p_m^-\}$, where $k \geqslant 0$, $m \geqslant 0$. We assume $\mathbf{p}^+$ and $\mathbf{p}^-$ to be disjoint. Then the set of positive and negative patterns $\mathbf{p} = \mathbf{p}^+ \cup \mathbf{p}^-$.

Definition 4.1.1. *Let* $\mathbf{L}$ *be some set of labels and* $\mathbf{p}$ *be a finite set of positive and negative patterns. Let* $\mathbf{r}$ *and* $\mathbf{a}$ *be two finite sets of atomic formulas such that* $Var(\mathbf{r}) \subseteq FreeVar(\mathbf{p})$ *and* $Var(\mathbf{a}) \subseteq FreeVar(\mathbf{p})$. *A production rule (PR) is an expression of the form:*

$$[l] \textit{ if } \mathbf{p} \textit{ then remove } \mathbf{r} \textit{ add } \mathbf{a},$$

where l *is some label from* $\mathbf{L}$.

A set $\mathbf{p}$ is called *condition* or *left-hand side* (LHS) of the rule l, and a pair of sets $(\mathbf{r}, \mathbf{a})$ is called *action* or *right-hand side* (RHS) of the rule l. Let σ be an any substitution, such that $x\sigma$ is a constant for any $x \in Var(\mathbf{r}) \cup Var(\mathbf{a})$. Then a pair $(\mathbf{r}\sigma, \mathbf{a}\sigma)$ is called an *action instance of an action* $(\mathbf{r}, \mathbf{a})$.

This variant of a PR definition is very similar to the one proposed in [21]. The main difference is that here we consider formulas as patterns (sentences as facts) rather than terms (ground terms).

4.1.2 Semantics of Production Rules over First-Order Knowledge Bases

Here we look deeply into conditions and actions of a PR. Conditions are checked and are performed over a working memory taking into account an unchangeable KB. It this section we consider a satisfiable set of sentences as a KB. We say that *a set of sentences* $\mathbf{F}$ (in particular working memory) *is consistent with a KB* $\mathcal{K}$, if the set $\mathcal{K} \cup \mathbf{F}$ is satisfiable.

In Subsection 4.1.2.1 we consider conditions of PRs and introduce two kinds of condition satisfiability, giving different semantics to this notion. They differ in matching of negative patterns. It is defined as not entailment in the first approach and as an entailment of negations in the second approach. In Subsection 4.1.2.2 we consider actions as an update and an erasure of a working memory with corresponding facts, and specify again two fundamentally different variants of semantics of these notions. The first one is based on changing of facts of working memory, while the second one is based on changing of models.

4.1.2.1 Conditions

Checking fireability, i.e. checking conditions of a rule is an important part of a Production System execution. Let us define this notion.

Definition 4.1.2. *Given a KB* $\mathcal{K}$*, a working memory* WM *and a finite set of positive and negative patterns* $\mathbf{p} = \mathbf{p}^+ \cup \mathbf{p}^-$*. If* WM *is consistent with* $\mathcal{K}$ *and there exists some substitution* σ *such that:*

1. $\forall x \in FreeVar(p)\ \ x\sigma \in \mathcal{C}$,
2. $\forall p' \in \mathbf{p}^+\ \ \mathcal{K} \cup WM \models p'\sigma$,
3. $\forall p' \in \mathbf{p}^-\ \ \mathcal{K} \cup WM \not\models p'\sigma$,

then the set $\mathbf{p}$ *is said to be* σ*-satisfiable in* $\mathcal{K}$ *and* WM.

We can consider another alternative definition of the satisfiability of a set of positive and negative patterns.

Definition 4.1.3. *Given a KB $\mathcal{K}$, a working memory WM and a finite set of positive and negative patterns $\mathbf{p} = \mathbf{p}^+ \cup \mathbf{p}^-$. If WM is consistent with $\mathcal{K}$ and there exists some substitution σ such that:*

1. $\forall x \in FreeVar(p)\ \ x\sigma \in \mathcal{C}$,
2. $\forall p' \in \mathbf{p}^+\ \ \mathcal{K} \cup WM \models p'\sigma$,
3. $\forall p' \in \mathbf{p}^-\ \ \mathcal{K} \cup WM \models \neg p'\sigma$,

then the set $\mathbf{p}$ is said to be strongly σ-satisfiable in $\mathcal{K}$ and WM.

Since, by definition, any fact from a working memory WM is a sentence, and for any pattern p' from a set of patterns $\mathbf{p}$, $p'\sigma$ is a sentence because of the item 1., the above definitions are well defined.

Definition 4.1.4. *Given a KB $\mathcal{K}$, a working memory WM and a finite set of positive and negative patterns $\mathbf{p} = \mathbf{p}^+ \cup \mathbf{p}^-$ and sets $\mathbf{r}$ and $\mathbf{a}$ defined as in the definition of a production rule. A PR*

[l] if $\mathbf{p}$ then remove $\mathbf{r}$ add $\mathbf{a}$

is said to be σ-fireable (strongly σ-fireable) in $\mathcal{K}$ and WM, if $\mathbf{p}$ is σ-satisfiable (strongly σ-satisfiable) in $\mathcal{K}$ and WM.

Since the condition 3. of the definition of a strongly σ-satisfiable set of patterns implies the condition 3 of the definition of a σ-satisfiable set of patterns, any strongly σ-satisfiable set of patterns is also σ-satisfiable and, hence, any strongly σ-fireable Production Rule is also σ-fireable. However these definitions are not equivalent.

Roughly speaking, the pattern satisfiability takes the assumption that all unknown information (it is not known for sure that it is positive) is considered as negative, while strong pattern satisfiability means that information is considered to be negative only if it is known for sure ("for sure" here means "holds for all the models of the KB").

Example 4.1.5. *Let everything, that begins with '*', be a constant, and everything, that begins with '?' be a variable.*

First, we define a special kind of KB - ontology $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$ and a working memory WM:

$$\mathcal{T} = \{HighPrice \sqsubseteq Price, LowPrice \sqsubseteq Price,$$
$$LowPrice \sqsubseteq \neg HighPrice\}$$
$$\mathcal{A} = \{Price(*300)\}$$
$$WM = \{Searching, House(*55, *red, *300), Available(*55),$$
$$HouseAddress(*55, *8, *Saint - Martin, *Paris),$$

$Person(*36, *John, *Smith),$

$Address(*36, *48, *Dante, *Milan)\}$

Second, we define a set of positive and negative patterns $\mathbf{p}$ *(here we consider only atomic patterns) and sets* $\mathbf{r}$ *and* $\mathbf{a}$*:*

$\mathbf{p} := \mathbf{p}^+ \cup \mathbf{p}^-$*, where*

$\mathbf{p}^+ := \{Searching, House(?id1, *red, ?price), Available(?id1),$

$HouseAddress(?id1, ?number1, ?street1, *Paris),$

$Person(?id2, ?name, ?surname),$

$Address(?id2, ?number2, ?street2, ?city2)\}$

$\mathbf{p}^- := \{HighPrice(?price)\}$

$\mathbf{r} := \{Searching, Available(?id1),$

$Address(?id2, ?number2, ?street2, ?city2)\}$

$\mathbf{a} := \{UnAvailable(?id1),$

$Address(?id2, ?number1, ?street1, *Paris)\}$

And then we consider a production rule

$$[HouseSearch] \text{ if } \mathbf{p} \text{ then remove } \mathbf{r} \text{ add } \mathbf{a}$$

Let us check, if this rule is fireable (strongly fireable) in $\mathcal{K}$ *and* WM*. It is enough to check if the set of patterns* $\mathbf{p}$ *is satisfiable (strongly satisfiable) in* $\mathcal{K}$ *and* WM*.*

We consider both the definition of a σ*-satisfiable set of patterns and the definition of a strongly* σ*-satisfiable set of patterns.* WM *is consistent with* $\mathcal{K}$*. Conditions 1 and 2 are the same in both definitions. It is evident, that there exists only one substitution* σ *such that conditions 1 and 2 are satisfied.*

Let $\mathcal{I} = (\Delta, \cdot^{\mathcal{I}})$ *be some model from* $Mod(\mathcal{T} \cup \mathcal{A} \cup WM)$ *such that*

$(*300)^{\mathcal{I}} = 300 \in \Delta$

$HighPrice^{\mathcal{I}} = \{x \in \mathbb{R} \,|\, x \geqslant 400\} \subset \Delta$

Then $\mathcal{I} \not\models HighPrice(*300)$ *and, hence,* $\mathbf{p}$ *is* σ*-satisfiable (the rule is* σ*-fireable).*

But the same set of patterns p *is not strongly* σ*-satisfiable in the same Knowledge Base* $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$ *and Working Memory* WM*. One can consider, for example, a model* $\mathcal{I}' = (\Delta, \cdot^{\mathcal{I}'})$*, where* $\cdot^{\mathcal{I}'}$ *is the same as* $\cdot^{\mathcal{I}}$ *except one assignment:*

$HighPrice^{\mathcal{I}'} = \{x \in \mathbb{R} \,|\, x \geqslant 200\} \subset \Delta$

Then $\mathcal{I}' \not\models \neg HighPrice(*300)$*. Thus the condition 3 of the definition of a strongly* σ*-satisfiable set of patterns is violated and the set* $\mathbf{p}$ *is not strongly* σ*-satisfiable (the rule is not strongly* σ*-fireable).*

Let us modify the example by replacing the A-Box assertion $Price(*300)$ *by* $LowPrice(*300)$

(that is defining a new A-Box $\mathcal{A}' := \{LowPrice(*300)\}$*). Then* $\mathcal{T} \cup \mathcal{A}' \cup WM \models \neg HighPrice(*300)$. *Thus, the set of patterns* $\mathbf{p}$ *is strongly* σ*-satisfiable (the rule is strongly* σ*-fireable) and, hence,* σ*-satisfiable (the rule is also* σ*-fireable).*

The example above shows that the choice of the definition of satisfiability (strong or not strong) of a set of patterns depends on an idea of fireability of rules that we have in mind. If we want to apply the rule to houses that together with other conditions are not ranked as having a high price (it is not implied, that they have a high price), we should use the definition of a σ-satisfiable set of patterns. If we want to apply the rule to houses that together with other conditions are ranked as having a low price or at least not having a high price (it is implied, that they have not a high price), we should use the definition of a strongly σ-satisfiable set of patterns.

4.1.2.2 Actions

Here we consider the second important aspect of PRs, that is action and its application, that represent the last step in each iteration of the inference cycle of Production Systems.

Any fireable Production Rule can be *fired* (*applied*). It means, that its RHS (action) is executed.

Given a Production Rule

$$[l] \text{ if } \mathbf{p} \text{ then remove } \mathbf{r} \text{ add } \mathbf{a},$$

which is σ-fireable (or strongly σ-fireable) in $\mathcal{K}$ and WM, its application leads to a new Working Memory WM' defined as output of some operation $Act(\mathcal{K}, WM, \mathbf{r}\sigma, \mathbf{a}\sigma)$.

We should note here, that if new working memory WM' is inconsistent with $\mathcal{K}$, the workflow stops, because for any substitution σ no rules are σ-fireable (strongly σ-fireable) for a working memory that is inconsistent with the KB, since a consistency of a working memory with a KB is one of the requirements of the pattern satisfiability (strong satisfiability).

The operation Act may be realized in different ways, that correspond to different approaches to update and erasure of FO theories. Many semantics for update (and erasure as its counterpart) are described in literature [60]. Here we follow two main directions of research in this topic:

- **formula-based**, that changes formulas of a working memory
- **model-based**, that changes models of a a working memory and a KB

Both of the approaches has advantages and disadvantages. Let us consider each of them in details.

4.1.2.3 Formula-Based Approach

This approach consists in simple removing and adding corresponding sets of facts from/to working memory:

$$Act(\mathcal{K},\, WM, f_r, f_a) := (\,WM - f_r) \cup f_a,$$

where f_r and f_a are sets of facts, that are supposed to be removed and added from/to working memory WM[1].

One of the drawbacks of this approach is that it depends on the formulas. So, it is *syntax-dependant*. And, for example, logically equivalent but syntactically different theories may behave differently under this formula-base semantics.

Example 4.1.6. *Let us consider some DL TBox $\mathcal{K}$ as a KB and two consistent with it ABoxes WM_1, WM_2 as WMs:*

$$\mathcal{K} = \{LowPrice \equiv Price \sqcap \neg HighPrice, NotHighPrice \equiv \neg HighPrice\}$$

$$WM_1 = \{LowPrice(a)\}$$

$$WM_2 = \{Price(a), NotHighPrice(a)\}$$

Then theories $\mathcal{K} \cup WM_1$ and $\mathcal{K} \cup WM_2$ are logically equivalent (have the same sets of models). Consider action instance $(\{NotHighPrice(a)\}, \{\})$, that has to be performed over both of the theories:

$$Act(\mathcal{K},\, WM_1, \{NotHighPrice(a)\}, \{\}) = \{LowPrice(a)\}$$

$$Act(\mathcal{K},\, WM_2, \{NotHighPrice(a)\}, \{\}) = \{Price(a)\}$$

So, in spite of the logical equivalence of the initial theories, we obtained different (not logically equivalent) results, that shows, that formula-based approach is really syntax-dependent.

Adding new facts to working memory can also lead to undesirable situations. It can result in inconsistency of a new working memory with respect to the KB.

Example 4.1.7. *Let $\mathcal{K}$ be some DL TBox, that we consider as a KB, and WM be some DL ABox, considered as a working memory consistent with $\mathcal{K}$. And let $LowPrice \sqsubseteq \neg HighPrice \in \mathcal{K}$ and $WM = \{LowPrice(a)\}$. Perform operation Act, adding $\{HighPrice(a)\}$ to the WM. Then a new working memory*

$$WM' = Act(\mathcal{T}, \mathcal{A}, \{\}, \{HighPrice(a)\}) = \{LowPrice(a), HighPrice(a)\}.$$

But it is inconsistent with the KB $\mathcal{K}$, that stops the workflow of the production system.

One more drawback resulting from the syntax-dependency is revealed when the operation Act removes facts from the working memory. The point is that the removed facts can still be entailed from the new theory (KB and a new working memory), so that the removing does not have any semantic impact.

[1]The order of removing and adding here is important, because in general for any three sets A, B and C, $(A - B) \cup C$ is not equal to $(A \cup C) - B$.

Example 4.1.8. *Let $\mathcal{K}$ be some DL TBox, that we consider as a KB, and WM be some DL ABox, considered as a working memory consistent with $\mathcal{K}$. And let $HighPrice \sqsubseteq Price \in \mathcal{K}$ and $WM = \{HighPrice(a), Price(a)\}$. Perform operation Act, removing $\{Price(a)\}$ from the working memory WM. Then a new working memory*

$$WM' = Act(\mathcal{K}, WM, \{Price(a)\}, \{\}) = \{HighPrice(a), \}.$$

So, in fact, semantically performing of this operation does not make any changes in a sense that the old theory $\mathcal{K} \cup WM$ and the new one $\mathcal{K} \cup WM'$ are logically equivalent and, hence, the same rules will be fireable again for the same substitutions.

4.1.2.4 Model-Based Approach

There are different model-based semantics for update and erasure [60], but all of them are based on the same idea. First, to perform an update and an erasure of a theory with f_a and f_r respectively, changes are to be done over each model of the theory. And second, these changes have to be as little as necessary to make f_a consistent and f_r inconsistent with the KB.

Here we consider so-called Possible Models Approach (PMA) introduced by Winslett [59]. To define it formally we need some assumptions and definitions.

We again consider a FO theory $\mathcal{K} \cup WM$ consisting of a protected set of sentences $\mathcal{K}$ and unprotected set of facts WM.

As usual we give the semantics of the theory in terms of interpretations and use Standard Names Assumption (SNA), i.e.:

1. All the interpretations share the same domain of interpretations, that is some fixed infinite domain Δ of objects;
2. All the interpretations interpret constants equally. That is, for any constant a and any interpretations $\mathcal{I} = (\Delta, \cdot^{\mathcal{I}})$ and $\mathcal{I}' = (\Delta, \cdot^{\mathcal{I}'})$ $a^{\mathcal{I}} = a^{\mathcal{I}'} = a^* \in \Delta$;
3. There is a constant for each object in Δ denoting that object. That is, for any $a^* \in \Delta$ there exists a constant a s.t. for any interpretation $\mathcal{I} = (\Delta, \cdot^{\mathcal{I}})$, $a^{\mathcal{I}} = a^*$.

So, in this case a set of constants and shared domain of interpretations are coincide and we can assume, that for any interpretation $\mathcal{I} = (\Delta, \cdot^{\mathcal{I}})$ (satisfying conditions above) and for any constant a, $a^{\mathcal{I}} = a$. So, we take a SNA for this subsection.

We use SNA to simplify comparison between models needed for updates. In fact, the use of standard names could be avoided, but this would make some of the definitions below clumsier.

Definition 4.1.9. *Let $\mathcal{I} = (\Delta, \cdot^{\mathcal{I}})$ and $\mathcal{I}' = (\Delta, \cdot^{\mathcal{I}'})$ be two interpretations. We say that $\mathcal{I}$ is contained in $\mathcal{I}'$, written $\mathcal{I} \subseteq \mathcal{I}'$, iff $\mathcal{I}$, $\mathcal{I}'$ are such that $P^{\mathcal{I}} \subseteq P^{\mathcal{I}'}$ for every atomic*

predicate P. We say that $\mathcal{I}$ is properly contained in $\mathcal{I}'$, written $\mathcal{I} \subset \mathcal{I}'$, iff $\mathcal{I} \subseteq \mathcal{I}'$ but $\mathcal{I}' \not\subseteq \mathcal{I}$.

Definition 4.1.10. *Let $\mathcal{I} = (\Delta, \cdot^{\mathcal{I}})$ and $\mathcal{I}' = (\Delta, \cdot^{\mathcal{I}'})$ be two interpretations. We define the difference between $\mathcal{I}$ and $\mathcal{I}'$, written $\mathcal{I} \ominus \mathcal{I}'$, as the interpretation $(\Delta, \cdot^{\mathcal{I} \ominus \mathcal{I}'})$ such that $P^{\mathcal{I} \ominus \mathcal{I}'} = P^{\mathcal{I}} \ominus P^{\mathcal{I}'}$ for every atomic predicate P, where $S \ominus S'$ denotes the symmetric difference between sets S and S', i.e. $S \ominus S' = (S \cup S') - (S \cap S')$.*

An *update set* is a finite set of sentences to be incorporated into a theory. Hereafter we consider a finite set of facts or negated facts as an update set, but in general the definitions below can be also used for any kinds of update set.

Definition 4.1.11. *Let $\mathcal{K}$ be a KB, WM be a working memory and $\mathbf{F}$ be an update set. Consider an interpretation $\mathcal{I} \in Mod(\mathcal{K})$ (where $Mod(\mathcal{K})$ denote the set of models of $\mathcal{K}$). The update of $\mathcal{I}$ with $\mathbf{F}$ is defined as follows:*

$$U^{\mathcal{K}}(\mathcal{I}, \mathbf{F}) = \{\mathcal{I}' \,|\, \mathcal{I}' \in Mod(\mathcal{K} \cup \mathbf{F}) \text{ and}$$
$$\text{there exists no } \mathcal{I}'' \in Mod(\mathcal{K} \cup \mathbf{F})$$
$$\text{s.t. } \mathcal{I} \ominus \mathcal{I}'' \subset \mathcal{I} \ominus \mathcal{I}'\}$$

In other words we take some model $\mathcal{I}$ of the TBox and consider all the models of the TBox, that are also models of the update set $\mathbf{F}$ and "closest" to the model $\mathcal{I}$.

In the following definition we just unite all such sets of models defined (by the Definition 4.1.11) for all models of the theory $\mathcal{K} \cup WM$.

Definition 4.1.12. *Let $\mathcal{K}$ be a KB, WM be a working memory and $\mathbf{F}$ be an update set. The update of $\mathcal{K} \cup WM$ with $\mathbf{F}$ is defined as follows:*

$$(\mathcal{K} \cup WM) \circ_{\mathcal{K}} \mathbf{F} = \bigcup_{\mathcal{I} \in Mod(\mathcal{K} \cup WM)} U^{\mathcal{K}}(\mathcal{I}, \mathbf{F}).$$

Definition 4.1.13. *Let $\mathcal{K}$ be a KB, WM be a working memory, $\mathbf{F}$ be an update set and $\neg\mathbf{F} = \{\neg F_i \,|\, F_i \in \mathbf{F}\}$. The erasure of $\mathcal{K} \cup WM$ with $\mathbf{F}$ is defined as follows:*

$$(\mathcal{K} \cup WM) \bullet_{\mathcal{K}} \mathbf{F} = Mod(\mathcal{K} \cup WM) \cup (\mathcal{K} \cup WM) \circ_{\mathcal{K}} (\neg\mathbf{F})$$

So, in this semantics update (erasure) is represented as a set of models, each corresponding to an updated (erased) state of affairs, that we consider possible. And here we encounter new problem: realization of the update (erasure) in some new working memory WM', that together with the protected set $\mathcal{K}$ expresses the update (erasure). That is,

$$Mod(\mathcal{K} \cup WM') = (\mathcal{K} \cup WM) \circ_{\mathcal{K}} \mathbf{F} \qquad (Mod(\mathcal{K} \cup WM') = (\mathcal{K} \cup WM) \bullet_{\mathcal{K}} \mathbf{F}).$$

In our framework update is being performed after erasure. So, in general, there is no need to materialize a new working memory after erasure is performed, since only models

are needed to perform the next operation, that is update. The same is true if the order of erasure and update operations is inverse. The only requirement is that the last performed operation (update in our case) has to materialize the resulting working memory.

There is an only open question here: how to find this procedure for building resulting working memory, and if it is always possible. It is shown in [41], that if we consider an ontology expressed in some DL as a KB, the update may become not expressible in this language (see also Example 4.1.19 and Example 4.1.20) and a notion of approximation should be studied (see Section 4.1.3.2).

4.1.3 Peculiarities of Semantics of Production Rules over Description Logic Knowledge Bases

FOL is very expressive, but the automated reasoning over it is hard. So, it seems reasonable to restrict expressivity to get the possibility to use reasoning algorithms and other techniques, developed for such restrictions of FOL. Here we consider DLs as a family of such restrictions and investigate a peculiarities of semantics of conditions and actions over DL KBs.

We employ some classical DL and take some ontology $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$ described in this logic as a KB.

In a working memory we take all unary and binary predicates as DL atomic concepts and roles respectively (that is the same) and all unary and binary facts from the working memory as DL *membership assertions* to atomic concepts and roles respectively (that is the same). Then we can consider two parts of the WM separately:

- *DL part of the working memory*, the part containing all the DL membership assertions,
- *Oth part of the working memory*, the part containing all the other n-ary facts ($n > 2$ or $n = 0$).

By analogy with working memory we can also consider DL and Oth parts of any set of atomic formulas $\mathbf{F}$, supposing that

- *DL part of the set* $\mathbf{F}$ consists of unary or binary predicates,
- *Oth part of the set* $\mathbf{F}$ contains of all other n-ary predicates ($n > 2$ or $n = 0$).

We denote DL and Oth parts of the working memory (set of atomic formulas $\mathbf{F}$) by indexing it by DL and oth respectively. For example WM_{DL} and WM_{oth} ($\mathbf{F}_{DL}$ and $\mathbf{F}_{oth}$) are respectively DL and Oth parts of the working memory WM (set $\mathbf{F}$).

In this section we consider only atomic patterns (atomic formulas).

It is logical to expect, that restriction of a general FOL KB to a DL KB and a set of any patterns to a set of atomic patterns can simplify a process of checking conditions and performing actions of PRs. And it is really the case.

Since in a DL KB only predicates of arity 1 and 2 occur, the ontology (DL KB) does not affect matching of patterns from $(p)_{oth}$ and simplify checking satisfiability of conditions of rules.

Concerning actions, under a supposition $WM = WM_{DL}$ we also present here some procedure of their performing (in PMA semantics) in some DLs from $DL-Lite$ family based on a DL ABox update and erasure. For some other DLs we introduce a notion of approximation and also give a procedure to find approximate results of actions in the case when a real result (new working memory) cannot be expressed in this logic.

4.1.3.1 Conditions

For this subsection we assume that any interpretation $\mathcal{I}$ respects the *unique name assumption* (UNA), that is, if a, b are distinct constants, then $a^{\mathcal{I}} \neq b^{\mathcal{I}}$.

$Prop$ Given some DL KB $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$, some working memory WM consistent with $\mathcal{K}$ and some set of patterns $\mathbf{p}$. For any substitution σ the following equivalence holds:

$$(\forall p'_{DL} \in \mathbf{p}_{DL}\ \ \mathcal{T} \cup \mathcal{A} \cup WM_{DL} \models p'_{DL}\sigma \text{ and}$$
$$\forall p'_{oth} \in \mathbf{p}_{oth}\ \ p'_{oth}\sigma \in WM_{oth}) \Longleftrightarrow \forall p' \in \mathbf{p}\ \ \mathcal{T} \cup \mathcal{A} \cup WM \models p'\sigma$$

Proof.

"$\Longrightarrow$" $Mod(\mathcal{T} \cup \mathcal{A} \cup WM) \subseteq Mod(\mathcal{T} \cup \mathcal{A} \cup WM_{DL})$ and $Mod(\mathcal{T} \cup \mathcal{A} \cup WM) \subseteq Mod(WM_{oth})$, then for any substitution σ

$$(\forall p'_{DL} \in \mathbf{p}_{DL}\ \ \mathcal{T} \cup \mathcal{A} \cup WM_{DL} \models p'_{DL}\sigma \text{ and}$$
$$\forall p'_{oth} \in \mathbf{p}_{oth}\ \ WM_{oth} \models p'_{oth}\sigma) \Longrightarrow \forall p' \in \mathbf{p}\ \ \mathcal{T} \cup \mathcal{A} \cup WM \models p'\sigma.$$

Since WM_{oth} by definition consists of only atomic sentences (that is atomic formulas with no free variables), and no dependences are defined between them, then

$$\forall\, p'_{oth} \in \mathbf{p}_{oth}\,(\,WM_{oth} \models p'_{oth}\sigma \Leftrightarrow p'_{oth}\sigma \in WM_{oth}), \tag{4.1}$$

and the straight implication holds.

"$\Longleftarrow$" Since WM_{DL} and WM_{oth} are well separated, that is no dependences are defined between facts from WM_{DL} and facts from WM_{oth}, then for any substitution σ sentences from WM_{oth} do not influence on logical implication of assertions from $\mathbf{p}_{DL}\sigma$, and assertions from $\mathcal{T} \cup \mathcal{A} \cup WM_{DL}$ do not influence on logical implication of sentences from $\mathbf{p}_{oth}\sigma$. Then the implication may be separated:

$$(\forall p'_{DL} \in \mathbf{p}_{DL}\ \ \mathcal{T} \cup \mathcal{A} \cup WM_{DL} \models p'_{DL}\sigma \text{ and}$$
$$\forall p'_{oth} \in \mathbf{p}_{oth}\ \ WM_{oth} \models p'_{oth}\sigma) \Longleftarrow \forall p' \in \mathbf{p}\ \ \mathcal{T} \cup \mathcal{A} \cup WM \models p'\sigma.$$

And taking into account statement (4.1), it is evident, that a backward implication also holds.

It should be noticed, that UNA is important here, and absence of it makes the above proposition in general false.

Example 4.1.14. *Suppose, that we do not make unique name assumption and consider a DL KB $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$, a working memory WM and a set of patterns $\mathbf{p}$, where*

$$\begin{aligned} &\mathcal{T} := \{(funct\ R)\} \\ &\mathcal{A} := \{\} \\ &WM := \{R(a,b),\ R(a,c),\ P(a,b,c)\} \\ &\mathbf{p} := \{P(x,y,y)\}, \end{aligned} \tag{4.2}$$

where the assertion $(funct\ R)$ represents the formula $\forall\, x, y, z\ R(x,y) \wedge R(x,z) \rightarrow y = z$. Then $WM_{DL} = \{R(a,b),\ R(a,c)\}$, $WM_{oth} = \{P(a,b,c)\}$, $\mathbf{p}_{DL} = \{\}$ and $\mathbf{p}_{oth} = \{P(x,y,y)\}$. For the substitution $\sigma := \{x/a,\ y/b\}$ $P(x,y,y)\sigma = P(a,b,b)$ and $\mathcal{T} \cup \mathcal{A} \cup WM \models P(a,b,b)$, but $P(a,b,b) \notin WM_{oth}$. Hence, the proposition 4.1.3.1 does not hold.

If we made UNA, we could not consider such a knowledge base $\mathcal{K}$ and a working memory WM together, because the set $\mathcal{K} \cup WM$ is not satisfiable in this case (if $\mathcal{T} := \{(funct\ R)\}$ and we assume UNA, WM can not contain both of the assertions $R(a,b)$ and $R(a,c)$).

It is evident from the example above, that the crucial thing here is an equality. A presence of an equality in a language (without UNA) makes the proposition 4.1.3.1 invalid in general. And we fix this bug by UNA.

Based on the above proposition we get equivalent definitions of a σ-satisfiable set of patterns and a strongly σ-satisfiable set of patterns that can be derived from propositions below. $Prop$ Given some DL KB $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$, some working memory WM and some set of positive and negative patterns $\mathbf{p} = \mathbf{p}^+ \cup \mathbf{p}^-$. Then the set $\mathbf{p}$ is σ-satisfiable in $\mathcal{K}$ and WM iff WM is consistent with $\mathcal{K}$ and there exists some substitution σ such that:

1. $\forall x \in Var(\mathbf{p})\ \ x\sigma \in \mathcal{C}$,
2. $\forall p'_{DL} \in \mathbf{p}^+_{DL}\ \ \mathcal{T} \cup \mathcal{A} \cup WM_{DL} \models p'_{DL}\sigma$,
3. $\forall p'_{oth} \in \mathbf{p}^+_{oth}\ \ p'_{oth}\sigma \in WM_{oth}$,
4. $\forall p'_{DL} \in \mathbf{p}^-_{DL}\ \ \mathcal{T} \cup \mathcal{A} \cup WM_{DL} \not\models p'_{DL}\sigma$,
5. $\forall p'_{oth} \in \mathbf{p}^-_{oth}\ \ p'_{oth}\sigma \notin WM_{oth}$.

$Prop$ Given some DL KB $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$, some working memory WM and some set of positive and negative patterns $\mathbf{p} = \mathbf{p}^+ \cup \mathbf{p}^-$. Then the set $\mathbf{p}$ is strongly σ-satisfiable in $\mathcal{K}$ and WM iff WM is consistent with $\mathcal{K}$ and there exists some substitution σ such that:

1. $\forall x \in Var(\mathbf{p})\ \ x\sigma \in \mathcal{C}$,
2. $\forall p'_{DL} \in \mathbf{p}^+_{DL}\ \ \mathcal{T} \cup \mathcal{A} \cup WM_{DL} \models p'_{DL}\sigma$,
3. $\forall p'_{oth} \in \mathbf{p}^+_{oth}\ \ p'_{oth}\sigma \in WM_{oth}$,
4. $\forall p'_{DL} \in \mathbf{p}^-_{DL} = \mathbf{p}^-\ \ \mathcal{T} \cup \mathcal{A} \cup WM_{DL} \models \neg p'_{DL}\sigma$,
5. $\mathbf{p}^-_{oth} = \emptyset$.

4.1.3.2 Actions

Here we consider the PMA semantics for update and erasure.

First, we restrict a working memory to its DL part, i.e. $WM = WM_{DL}$. Second, we extend a working memory with all the other ABox assertions allowed in the DL under consideration. Usually these assertions are membership assertions to general concepts, that are represented by concept expressions, for example $\exists R(a)$, $\neg B(b)$.

Notation 4.1.15. *The extension of a working memory with these general membership assertions is not so critical. Instead of considering in a working memory a membership assertion $C(a)$, where C is some general concept (for example, $\exists R$, $\neg B$) one can introduce a new atomic concept A_C, set it equivalent to C by adding an equivalence axiom $A_C \equiv C$ to a TBox of the KB and replace the assertion $C(a)$ by $A_C(a)$.*

Thus, since a working memory is a finite set, a finite set of such rewritings, introducing a finite set of new atomic concepts and adding a finite set of new equivalence axioms to the TBox, has to be done.

In some DLs (for example, some DLs from the DL-Lite family) so called inverse role membership assertions, that is assertions of the form $P^-(a, b)$, where P is an atomic role,are allowed in the ABox. Any such assertion $P^-(a, b)$ can also be replaced by an equivalent one $P(b, a)$.

As we mentioned above, in general results of an update and an erasure can not always be expressed in the same logic, in which the knowledge base is expressed (see [41] and also Example 4.1.19 and Example 4.1.20). In such cases it is useful to use an approximation of an instance level update and erasure introduced in [26] modified and adapted to our framework.

Definition 4.1.16. *Let $\mathcal{K}$ be an ontology (KB) in a DL $\mathcal{L}$, WM be a working memory and $\mathcal{M}$ be a set of models s.t. $\mathcal{M} \subseteq Mod(\mathcal{K})$. We say that WM is a sound $(\mathcal{L}, \mathcal{K})$-approximation of $\mathcal{M}$ in $\mathcal{L}$, if*

(i) WM is an ABox in $\mathcal{L}$,

(ii) $\mathcal{M} \subseteq Mod(\mathcal{K} \cup WM)$.

Definition 4.1.17. *Let $\mathcal{K}$ be an ontology (KB) in a DL $\mathcal{L}$, WM be a working memory and $\mathcal{M}$ be a set of models s.t. $\mathcal{M} \subseteq Mod(\mathcal{K})$. We say that WM is a maximal $(\mathcal{L}, \mathcal{K})$-approximation of $\mathcal{M}$, if*

(i) WM is a sound $(\mathcal{L}, \mathcal{K})$-approximation of $\mathcal{M}$,

(ii) there exists no working memory WM' that is a sound $(\mathcal{L}, \mathcal{K})$-approximation of $\mathcal{M}$, and is s.t. $Mod(\mathcal{K} \cup WM') \subset Mod(\mathcal{K} \cup WM)$.

Similar to a working memory we extend an update set with all the other ABox assertions allowed in the DL under consideration.

Definition 4.1.18. *Let $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$ be an ontology (KB) in a DL $\mathcal{L}$, WM, WM' two working memories and $\mathbf{F}$ an update set in $\mathcal{L}$. We say that WM' is a $(\mathcal{L}, \mathcal{K})$-update of WM with $\mathbf{F}$ ($(\mathcal{L}, \mathcal{K})$-erasure of WM with $\mathbf{F}$) if WM' is a maximal $(\mathcal{L}, \mathcal{K})$-approximation of $(\mathcal{K} \cup WM) \circ_{\mathcal{K}} \mathbf{F}$ ($(\mathcal{K} \cup WM) \bullet_{\mathcal{K}} \mathbf{F}$).*

$(\mathcal{L}, \mathcal{K})$-update ($(\mathcal{L}, \mathcal{K})$-erasure) WM' can be considered as a working memory expressed in the logic under consideration $\mathcal{L}$, such that the set of models of the theory $\mathcal{K} \cup WM'$, i.e. $Mod(\mathcal{K} \cup WM')$, is"closest" to the update $(\mathcal{K} \cup WM) \circ_{\mathcal{K}} \mathbf{F}$ (erasure $(\mathcal{K} \cup WM) \bullet_{\mathcal{K}} \mathbf{F}$) itself.

$(\mathcal{L}, \mathcal{K})$-update and $(\mathcal{L}, \mathcal{K})$-erasure have some good properties. For example, an approximation does not influence on the implication of membership assertions expressed in the same DL $\mathcal{L}$, i.e. for every membership assertion α in $\mathcal{L}$ we have that $(\mathcal{K} \cup WM) \circ_{\mathcal{K}} \mathbf{F} \models \alpha$ iff $\mathcal{K} \cup WM' \models \alpha$ and $(\mathcal{K} \cup WM) \bullet_{\mathcal{K}} \mathbf{F} \models \alpha$ iff $\mathcal{K} \cup WM' \models \alpha$. That is, an approximation does not influence on satisfiability of conditions of rules and, hence, on their fireability.

4.1.3.2.1 Computing Update and Erasure in *DL-Lite*

Some languages from *DL-Lite* family possess a good properties of expressivity of update/erasure or existence of its approximations.

Let again $\mathcal{L}$ be some DL and $\mathbf{F}$ be an update set. In the remainder suppose that an ABox of a DL KB $\mathcal{K}$ is empty[2]. Then the ontology is presented only as a TBox. Since a working memory WM is an ABox in $\mathcal{L}$, the theory $\mathcal{K} \cup WM$ can be viewed as an ontology with a TBox $\mathcal{K}$ and an ABox WM. So, we apply here research results in an area of an ABox update and erasure.

It is notable, that for $DL-Lite_{FS}$ language a result of an update is expressible in the same language, that is there exists a working memory WM' expressed in $DL-Lite_{FS}$ such that

$$(\mathcal{K} \cup WM) \circ_{\mathcal{K}} \mathbf{F} = Mod(\mathcal{K} \cup WM').$$

[2]This supposition is made for the sake of simplicity, and this particular case can be trivially extended to the general one, when an ABox of $\mathcal{K}$ is not empty.

And a sound and complete algorithm for computing the updated ABox (that is thw orking memory in our case) WM' in polynomial time (in the size of $\mathcal{K} \cup WM \cup \mathbf{F}$) was presented in [25].

Example 4.1.19. *Let us consider DL $DL-Lite_{\mathcal{FS}}$. Let $\mathcal{K}$ and WM be a KB and a working memory expressed in this logic respectively, and $\mathbf{F}$ be an update set, where*

$$\begin{aligned} \mathcal{K} &= \{HighRankedAccomodation \sqsubseteq RecentlyRepared, \\ &\qquad OutARepair \sqsubseteq \neg RecentlyRepared\}, \\ WM &= \{OutARepair(a)\}, \\ \mathbf{F} &= \{RecentlyRepared(a)\}. \end{aligned} \tag{4.3}$$

Let us compute the update of $\mathcal{K} \cup WM$ with $\mathbf{F}$. According to the aforementioned algorithm a new working memory WM' has to include all the assertions from $\mathbf{F}$, all the assertions from WM, that do not contradict $\mathbf{F}$ and all the assertions, that do not contradict $\mathbf{F}$ and are logically implied by assertions from WM, that contradict $\mathbf{F}$. Thus, $WM' = \{RecentlyRepared(a)$, $\neg HighRankedAccomodation(a)\}$, that is an ABox in $DL-Lite_{\mathcal{FS}}$.

But if we consider initially the logic $DL-Lite_{\mathcal{F}}$ (this is possible, because all the assertions from $\mathcal{K}$, WM, $\mathbf{F}$ are allowed also in $DL-Lite_{\mathcal{F}}$), the new working memory WM' is not expressible in the same logic any more, because the assertion $\neg HighRankedAccomodation(a)$ can not be expressed in $DL-Lite_{\mathcal{F}}$.

Example 4.1.20. *Let us consider DL $DL-Lite_{\mathcal{FS}}$ again. Let $\mathcal{K}$ and WM be a KB and a working memory expressed in this logic respectively, and $\mathbf{F}$ be an update set, where*

$$\begin{aligned} \mathcal{K} &= \{HighRankedAccomodation \sqsubseteq RecentlyRepared, \\ &\qquad HighRankedAccomodation \sqsubseteq \exists hasBalcony\}, \\ WM &= \{HighRankedAccomodation(a)\}, \\ \mathbf{F} &= \{\exists hasBalcony(a)\}. \end{aligned} \tag{4.4}$$

Let us compute the erasure of $\mathcal{K} \cup WM$ with $\mathbf{F}$. According to the definition 4.1.13, first we need to compute the update of $\mathcal{K} \cup WM$ with $\neg\mathbf{F}$. By analogy with the previous example an updated working memory is equal to $\{RecentlyRepared(a), \neg\exists hasBalcony(a)\}$. Then $(\mathcal{K} \cup WM) \bullet_{\mathcal{K}} \mathbf{F} = Mod(\mathcal{K} \cup WM) \cup (\mathcal{K} \cup WM) \circ_{\mathcal{K}} (\neg\mathbf{F}) = Mod(\mathcal{K} \cup \{HighRankedAccomodation(a)\}) \cup Mod(\mathcal{K} \cup \{RecentlyRepared(a), \neg\exists hasBalcony(a)\}) = Mod(\mathcal{K} \cup \{RecentlyRepared(a), (HighRankedAccomodation \sqcup \neg\exists hasBalcony)(a)\})$.

Then a new working memory

$$WM' = \{RecentlyRepared(a), (HighRankedAccomodation \sqcup \neg\exists hasBalcony)(a)\},$$

and it is not expressible in $DL-Lite_{\mathcal{FS}}$ (and, hence, in $DL-Lite_{\mathcal{F}}$), because a disjunction of concepts is not allowed in this logic.

Although in general update and erasure of a theory $\mathcal{K} \cup WM$ with an update set $\mathbf{F}$ considered in $DL-Lite_{\mathcal{F}}$ is not expressible in $DL-Lite_{\mathcal{F}}$, and erasure of a theory $\mathcal{K} \cup WM$ with an update set $\mathbf{F}$ considered in $DL-Lite_{\mathcal{FS}}$ is also not expressible in $DL-Lite_{\mathcal{FS}}$, there exists a common algorithm, originally introduced in [26], for computing corresponding approximations. The algorithm has two parameters ($(\mathbf{S})$ and $(\mathcal{L})$) and consists of two steps:

1. Compute update of $\mathcal{K} \cup WM$ with $(\mathbf{S})$ in $DL-Lite_{\mathcal{FS}}$ using the algorithm mentioned above,
2. Delete all the membership assertions, that are not allowed in $(\mathcal{L})$.

$(\mathcal{L})$ here stands for logic, in which we consider update or erasure ($DL-Lite_{\mathcal{F}}$ or $DL-Lite_{\mathcal{FS}}$). To compute $(\mathcal{L},\mathcal{K})$-*update of* WM *with* $\mathbf{F}$ ($(\mathcal{L},\mathcal{K})$-*erasure of* WM *with* $\mathbf{F}$) we need to substitute $(\mathbf{S})$ with $\mathbf{F}$ ($\neg\mathbf{F}$)[3].

With this algorithm in place let us recall Example 4.1.19 and Example 4.1.20. Applying this algorithm we get $\mathcal{A}' = \{RecentlyRepared(a)\}$ for both examples.

Let us now define the operation Act.

Given a DL KB $\mathcal{K}$, a working memory WM both expressed in DL $\mathcal{L}$, that stands for $DL-Lite_F$ or $DL-Lite_{FS}$, and an update set $\mathbf{F}$. Then we introduce auxiliary operations:

$$ComputeUpdate_{\mathcal{L}}^{app}(\mathcal{K}, WM, \mathbf{F}),\ \ ComputeErasure_{\mathcal{L}}^{app}(\mathcal{K}, WM, \mathbf{F}),$$

that using the algorithm above computes the $(\mathcal{L},\mathcal{K})$-update of the WM with $\mathbf{F}$ and $(\mathcal{L},\mathcal{K})$-erasure of the WM with $\mathbf{F}$ respectively.

Let $\mathbf{R}$ and $\mathbf{A}$ be two update sets. The operation Act returning a new working memory after erasing of the old one with $\mathbf{R}$ and updating with $\mathbf{A}$ is defined as follows:

$$Act(\mathcal{K}, WM, \mathbf{R}, \mathbf{A}) = ComputeUpdate_{\mathcal{L}}^{app}(\mathcal{K}, ComputeErasure_{\mathcal{L}}^{app}(\mathcal{K}, WM, \mathbf{R}), \mathbf{A}).$$

4.1.3.2.2 Alternative Semantics for Update and Erasure over Acyclic TBoxes In this section having in mind the same notion of "minimal changes" adopted in PMA we introduce a new model-based semantics of update and erasure for a special kind of a DL KB, namely finite acyclic set of concept definitions. We transform and adapt an approach proposed in [41] and introduce new definitions of update and erasure of a model by explicit and intuitive fixing of interpretations of primitive concepts. This fixing is simply adding (in the case of update) and removing (in the case of erasure) elements occurring in the update set to/from interpretations of particular primitive concepts. But is there any connection between this constructive and intuitive approach and the PMA

[3]The step 2 in the algorithm is redundant for computing an update in $DL-Lite_{\mathcal{FS}}$, because an update is expressible in this logic.

introduced before? Here we prove a theorem stating that these two approaches give the same result, i.e update and erasure introduced in this approach coincide with update and erasure in PMA, so we can use any of them depending on a particular situation.

Suppose here we consider some DL $\mathcal{L}$. A *concept definition* is of the form $A \equiv C$, where A is an atomic concept and C is a concept (concept description allowed in $\mathcal{L}$). The concept definition is logically equivalent to two concept inclusions: $A \sqsubseteq C$ and $C \sqsubseteq A$.

Given a set $\mathcal{K}$ of concept definitions, we say that the atomic concept A is *directly defined* by the atomic concept B, if $\mathcal{K}$ contains a concept definition $A \equiv C$ s.t B occurs in C. Let *is defined* be the transitive closure of the relation is directly defined. We say that $\mathcal{T}$ is *acyclic*, if no concept is defined (directly or indirectly) by itself.

In this section as a KB $\mathcal{K}$ we consider a finite acyclic set of concept definitions (TBox) expressed in $\mathcal{L}$ with unique left-hand sides (in contrast with so-called *general TBox*, i.e. a finite set of *general concept inclusion axioms*, that we considered before). Remember, a working memory is a finite set of *concept assertions* $C(a)$, where C is a concept (concept description in $\mathcal{L}$) and *role assertions* $R(a, b)$, where R is a role.

We call an atomic concept A *defined* in a $\mathcal{K}$ and write $A \in def(\mathcal{K})$, if A occurs on the left-hand side of a concept definition in $\mathcal{K}$. Otherwise we call A *primitive* and write $A \in prim(\mathcal{K})$.

A *primitive interpretation of* $\mathcal{K}$ is an interpretation that interprets only the primitive atomic concepts in $\mathcal{K}$ and the atomic roles, but not the defined atomic concepts. In the case of acyclic TBoxes, as in our case, any primitive interpretation of $\mathcal{K}$ can uniquely be extended to a model of $\mathcal{K}$. It means, that any KB $\mathcal{K}$ is satisfiable, because a primitive interpretation always exists and it is always extended to a model of $\mathcal{K}$. In the following we use $prim_{\mathcal{K}}(\mathcal{I})$ to denote the primitive interpretation of $\mathcal{K}$ that can be extended to the model $\mathcal{I}$ of $\mathcal{K}$. If this model is also a model of a working memory WM, then the original primitive interpretation of $\mathcal{K}$ is said to be a *primitive model of a theory* $\mathcal{K} \cup WM$. In other words, a primitive interpretation of $\mathcal{K}$ is a *primitive model of a theory* $\mathcal{K} \cup WM$ if it can be extended to a model of $\mathcal{K} \cup WM$ by additionally interpreting the defined atomic concepts. So, the set of all the models of the theory $\mathcal{K} \cup WM$ is completely and uniquely defined by the set of all primitive models of $\mathcal{K}$.

Let us consider a finite set $\mathbf{F}$ of primitive concept assertions and atomic role assertions as an update set. Any update set is satisfiable, because the update set represents its model by itself.

$Prop$ Let $\mathcal{K}$ be a KB and $\mathbf{F}$ be an update set. Then $\mathbf{F}$ is consistent with $\mathcal{K}$.

Proof. Since $\mathbf{F}$ is satisfiable, let us take any model $\mathcal{I}$ of $\mathbf{F}$ and restrict it to an interpretation that interprets only the primitive concepts and the atomic roles. This restriction is a primitive interpretation of $\mathcal{K}$ by definition, and, hence, we can extend it to a model $\mathcal{I}'$ of $\mathcal{K}$ in a unique way. $\mathcal{I}'$ is also a model of $\mathbf{F}$, because it interprets the primitive concepts and the atomic roles equally to $\mathcal{I}$. $\square$

Definition 4.1.21. *Let $\mathcal{K}$ be a KB, $\mathbf{F}$ an update set and $\mathcal{I} = (\Delta, \cdot^{\mathcal{I}})$, $\mathcal{I}' = (\Delta, \cdot^{\mathcal{I}'})$ models of $\mathcal{K}$ (extended primitive interpretations of $\mathcal{T}$). Then $\mathcal{I}'$ is the result of updating $\mathcal{I}$ with $\mathbf{F}$ for $\mathcal{K}$, written $\mathcal{I} \Rightarrow^{\mathcal{K}}_{\mathbf{F}} \mathcal{I}'$ if the following hold:*

- *for all atomic concepts $A \in prim(\mathcal{T})$*

$$A^{\mathcal{I}'} = A^{\mathcal{I}} \cup \{a \mid A(a) \in \mathbf{F}\} \tag{4.5}$$

- *forall atomic roles R*

$$R^{\mathcal{I}'} = R^{\mathcal{I}} \cup \{(a,b) \mid R(a,b) \in \mathbf{F}\} \tag{4.6}$$

And $\mathcal{I}'$ is the result of erasing $\mathcal{I}$ with $\mathbf{F}$ for $\mathcal{K}$, written $\mathcal{I} \Rightarrow^{\mathcal{K}}_{(\neg\mathbf{F})} \mathcal{I}'$ if the following hold:

- *for all atomic concepts $A \in prim(\mathcal{T})$*

$$A^{\mathcal{I}'} = A^{\mathcal{I}} - \{a \mid A(a) \in \mathbf{F}\} \tag{4.7}$$

- *forall atomic roles R*

$$R^{\mathcal{I}'} = R^{\mathcal{I}} - \{(a,b) \mid R(a,b) \in \mathbf{F}\} \tag{4.8}$$

Observe, that the relations $\Rightarrow^{\mathcal{K}}_{\mathbf{F}}$ and $\Rightarrow^{\mathcal{K}}_{(\neg\mathbf{F})}$ are functional (deterministic) because, as we said above, in models of a KB, the interpretation of primitive concepts and atomic roles determines the interpretation of defined concepts in a unique way. Therefore we can write $\mathcal{I}^{\mathbf{F}}_{\mathcal{K}}$ to denote the unique $\mathcal{I}'$ s.t. $\mathcal{I} \Rightarrow^{\mathcal{K}}_{\mathbf{F}} \mathcal{I}'$ and $\mathcal{I}^{(\neg\mathbf{F})}_{\mathcal{K}}$ to denote the unique $\mathcal{I}'$ s.t. $\mathcal{I} \Rightarrow^{\mathcal{K}}_{(\neg\mathbf{F})} \mathcal{I}'$.

Notation 4.1.22. *Since a domain of interpretations Δ is fixed and is essentially the set of all constants and in any model of a KB, the interpretation of primitive concepts and atomic roles determines the interpretation of defined concepts in a unique way, it is possible to see any model $\mathcal{I}$ of a TBox as a subset of $\mathcal{B}^{\Delta}_{H}$ (a set of all constructible primitive concept assertions and role assertions):*

$$\mathcal{I} = \{A(a) \in \mathcal{B}^{\Delta}_{H} \mid \mathcal{I} \models A(a)\} \cup \{R(a,b) \in \mathcal{B}^{\Delta}_{H} \mid \mathcal{I} \models R(a,b)\} \tag{4.9}$$

If we consider $\mathcal{I}^{\mathbf{F}}_{\mathcal{T}}$ and $\mathcal{I}^{(\neg\mathbf{F})}_{\mathcal{T}}$ as sets of assertions from $\mathcal{B}^{\Delta}_{H}$, then

$$\mathcal{I}^{\mathbf{F}}_{\mathcal{T}} = \mathcal{I} \cup \mathbf{F} \tag{4.10}$$

$$\mathcal{I}^{(\neg\mathbf{F})}_{\mathcal{T}} = \mathcal{I} - \mathbf{F} \tag{4.11}$$

The following two theorems build a bridge between PMA semantics and the new semantics introduced in this subsection. The theorem 4.1.23 states that the model updates coincide. And the theorem 4.1.24 states that the model erasures coincide.

Theorem 4.1.23. *Let $\mathcal{K}$ be a KB in $\mathcal{L}$, $\mathcal{I}$ a model of $\mathcal{K}$ and $\mathbf{F}$ an update set. Then*

$$U^{\mathcal{K}}(\mathcal{I}, \mathbf{F}) = \{\mathcal{I}^{\mathbf{F}}_{\mathcal{K}}\} \tag{4.12}$$

Proof.

1. Let us show that $\mathcal{I}^{\mathbf{F}}_{\mathcal{K}} \in U^{\mathcal{K}}(\mathcal{I}, \mathbf{F})$. $\mathcal{I}^{\mathbf{F}}_{\mathcal{K}} \in Mod(\mathcal{K})$ by definition of $\mathcal{I}^{\mathbf{F}}_{\mathcal{K}}$. For any primitive concept assertion $A(a)$ from $\mathbf{F}$ $a \in A^{\mathcal{I}^{\mathbf{F}}_{\mathcal{K}}}$ by definition of $\mathcal{I}^{\mathbf{F}}_{\mathcal{K}}$. And for any atomic role assertion $R(a, b)$ from $\mathbf{F}$ $(a, b) \in R^{\mathcal{I}^{\mathbf{F}}_{\mathcal{K}}}$ also by definition of $\mathcal{I}^{\mathbf{F}}_{\mathcal{K}}$. Hence, $\mathcal{I}^{\mathbf{F}}_{\mathcal{K}} \in Mod(\mathbf{F})$.

   Suppose now, that there exists $\mathcal{I}'' \in Mod(\mathcal{K} \cup \mathbf{F})$ s.t. $\mathcal{I} \ominus \mathcal{I}'' \subset \mathcal{I} \ominus \mathcal{I}^{\mathbf{F}}_{\mathcal{K}}$, that is

   - for every primitive concept A

   $$(A^{\mathcal{I}} \cup A^{\mathcal{I}''}) - (A^{\mathcal{I}} \cap A^{\mathcal{I}''}) \subset (A^{\mathcal{I}} \cup A^{\mathcal{I}^{\mathbf{F}}_{\mathcal{K}}}) - (A^{\mathcal{I}} \cap A^{\mathcal{I}^{\mathbf{F}}_{\mathcal{K}}}) \tag{4.13}$$

   - for every atomic role R

   $$(R^{\mathcal{I}} \cup R^{\mathcal{I}''}) - (R^{\mathcal{I}} \cap R^{\mathcal{I}''}) \subset (R^{\mathcal{I}} \cup R^{\mathcal{I}^{\mathbf{F}}_{\mathcal{K}}}) - (R^{\mathcal{I}} \cap R^{\mathcal{I}^{\mathbf{F}}_{\mathcal{K}}}) \tag{4.14}$$

   If there are primitive concept assertions in $\mathbf{F}$ (not only atomic role assertions), then consider some primitive concept name A, that occurs in $\mathbf{F}$. Then $(A^{\mathcal{I}} \cup A^{\mathcal{I}^{\mathbf{F}}_{\mathcal{K}}}) - (A^{\mathcal{I}} \cap A^{\mathcal{I}^{\mathbf{F}}_{\mathcal{K}}}) = A^{\mathcal{I}^{\mathbf{F}}_{\mathcal{K}}} - A^{\mathcal{I}} = \{a \,|\, A(a) \in \mathbf{F}\} - A^{\mathcal{I}}$. Since $\mathcal{I}'' \in Mod(\mathbf{F})$, $A^{\mathcal{I}''} \supseteq \{a \,|\, A(a) \in \mathbf{F}\}$ and $A^{\mathcal{I}} \cup A^{\mathcal{I}''} \supseteq A^{\mathcal{I}''} \supseteq \{a \,|\, A(a) \in \mathbf{F}\}$. $A^{\mathcal{I}} \cap A^{\mathcal{I}''} \subseteq A^{\mathcal{I}}$. Hence, $(A^{\mathcal{I}} \cup A^{\mathcal{I}''}) - (A^{\mathcal{I}} \cap A^{\mathcal{I}''}) \supseteq \{a \,|\, A(a) \in \mathbf{F}\} - A^{\mathcal{I}} = (A^{\mathcal{I}} \cup A^{\mathcal{I}^{\mathbf{F}}_{\mathcal{K}}}) - (A^{\mathcal{I}} \cap A^{\mathcal{I}^{\mathbf{F}}_{\mathcal{K}}})$. Contradiction with 4.13.

   If $\mathbf{F}$ consists of only atomic role assertions, then consider some atomic role R, that occurs in $\mathbf{F}$ and acting similarly come to contradiction with 4.14.

2. Let us show now that $U^{\mathcal{K}}(\mathcal{I}, \mathbf{F})$ consists of only one interpretation $\mathcal{I}^{\mathbf{F}}_{\mathcal{K}}$. As we showed in item 1 of the proof, $\mathcal{I}^{\mathbf{F}}_{\mathcal{K}} \in Mod(\mathcal{K} \cup \mathbf{F})$. If $\mathcal{I}^{\mathbf{F}}_{\mathcal{K}}$ is the only model of $\mathcal{K} \cup \mathbf{F}$, the current item is proved by definition of $U^{\mathcal{K}}(\mathcal{I}, \mathbf{F})$. If it is not the case, consider any interpretation $\mathcal{I}' \in Mod(\mathcal{K} \cup \mathbf{F})$ different from $\mathcal{I}^{\mathbf{F}}_{\mathcal{K}}$ (i.e., there exists some primitive concept A or atomic role R s.t. $A^{\mathcal{I}'} \neq A^{\mathcal{I}^{\mathbf{F}}_{\mathcal{K}}}$ or $R^{\mathcal{I}'} \neq R^{\mathcal{I}^{\mathbf{F}}_{\mathcal{K}}}$).

   As was already shown, for any primitive concept A $(A^{\mathcal{I}} \cup A^{\mathcal{I}^{\mathbf{F}}_{\mathcal{K}}}) - (A^{\mathcal{I}} \cap A^{\mathcal{I}^{\mathbf{F}}_{\mathcal{K}}}) = \{a \,|\, A(a) \in \mathbf{F}\} - A^{\mathcal{I}}$. Since $\mathcal{I}' \in Mod(\mathcal{K} \cup \mathbf{F})$, then $\{a \,|\, A(a) \in \mathbf{F}\} \subseteq A^{\mathcal{I}'}$, hence $\{a \,|\, A(a) \in \mathbf{F}\} - A^{\mathcal{I}} \subseteq A^{\mathcal{I}'} - A^{\mathcal{I}} \subseteq (A^{\mathcal{I}} \cup A^{\mathcal{I}'}) - (A^{\mathcal{I}} \cap A^{\mathcal{I}'})$. Then $(A^{\mathcal{I}} \cup A^{\mathcal{I}^{\mathbf{F}}_{\mathcal{K}}}) - (A^{\mathcal{I}} \cap A^{\mathcal{I}^{\mathbf{F}}_{\mathcal{K}}}) \subseteq (A^{\mathcal{I}} \cup A^{\mathcal{I}'}) - (A^{\mathcal{I}} \cap A^{\mathcal{I}'})$. Analogously, for any atomic role R $(R^{\mathcal{I}} \cup R^{\mathcal{I}^{\mathbf{F}}_{\mathcal{K}}}) - (R^{\mathcal{I}} \cap R^{\mathcal{I}^{\mathbf{F}}_{\mathcal{K}}}) \subseteq (R^{\mathcal{I}} \cup R^{\mathcal{I}'}) - (R^{\mathcal{I}} \cap R^{\mathcal{I}'})$. Hence, $\mathcal{I} \ominus \mathcal{I}^{\mathbf{F}}_{\mathcal{K}} \subseteq \mathcal{I} \ominus \mathcal{I}'$. It is easy to show that $\mathcal{I} \ominus \mathcal{I}^{\mathbf{F}}_{\mathcal{K}} = \mathcal{I} \ominus \mathcal{I}'$ iff $\mathcal{I}^{\mathbf{F}}_{\mathcal{K}} = \mathcal{I}'$ (interpretations of all primitive concepts and atomic roles coincide). But we supposed, that it is not the case. Hence, $\mathcal{I} \ominus \mathcal{I}^{\mathbf{F}}_{\mathcal{K}} \subset \mathcal{I} \ominus \mathcal{I}'$. Then $\mathcal{I}' \notin U^{\mathcal{K}}(\mathcal{I}, \mathbf{F})$, that means, $U^{\mathcal{K}}(\mathcal{I}, \mathbf{F}) = \{\mathcal{I}^{\mathbf{F}}_{\mathcal{K}}\}$.

Theorem 4.1.24. *Let $\mathcal{K}$ be a KB in $\mathcal{L}$, $\mathcal{I}$ a model of $\mathcal{K}$ and $\mathbf{F}$ an update set. Then*

$$U^{\mathcal{K}}(\mathcal{I}, \neg\mathbf{F}) = \{\mathcal{I}_{\mathcal{K}}^{(\neg\mathbf{F})}\} \tag{4.15}$$

Proof. is similar to proof of the theorem 4.1.23. □

Corollary 4.1.25. *Let $\mathcal{K}$ be a KB in $\mathcal{L}$, WM a working memory and $\mathbf{F}$ an update set. Then*

$$(\mathcal{K} \cup WM) \circ_{\mathcal{K}} \mathbf{F} = \{\mathcal{I}_{\mathcal{K}}^{\mathbf{F}} \,|\, \mathcal{I} \in Mod(\mathcal{K} \cup WM)\} \tag{4.16}$$

$$(\mathcal{K} \cup WM) \bullet_{\mathcal{K}} \mathbf{F} = Mod(\mathcal{K} \cup WM) \cup \{\mathcal{I}_{\mathcal{K}}^{(\neg\mathbf{F})} \,|\, \mathcal{I} \in Mod(\mathcal{K} \cup WM)\}, \tag{4.17}$$

where $\mathcal{I}_{\mathcal{T}}^{\mathbf{F}}$ and $\mathcal{I}_{\mathcal{T}}^{(\neg\mathbf{F})}$ are building according to definition 4.1.21 or, if they are considered as sets of assertions (see Notation 4.1.22), according to 4.10 and 4.11.

In this section we introduced a variant of description of PRs in attempt to combine them with ontologies, represented as a FO or DL KB. Since this approach is new, it can be extended in several directions.

In a condition part it would be interesting, for example, to try to combine a kind of semantics for conditions satisfiability we introduced here with standard patterns matching algorithms, such as RETE [21]. It could allow us to use different kinds of patterns in a condition, raising its expressivity.

Concerning actions, different kinds of semantics for update (erasure) and maybe even their combinations can be considered here. Following the PMA semantics it would be useful to try to find some other DLs, where update (erasure), or at least its approximation, is expressible.

4.2 Tightly Coupling Production Rules and Ontologies

As discussed in the introduction, there is a set of problems when combining DL ontologies and production systems which need to be solved. Summarizing:

1. How the ontology affects the PS, and how the PS affects ontology
2. How to model the execution of the system in a fair and still "relevant" way.[1]
3. Which is the Domain of discourse and which assumption we have about the logic

[1] With relevant we mean that when reasoning over the model, we can infer relevant information about the system.

In Section 4.2.1 we start tackling point 1, i.e., how to interpret predicates and formulas when combing the *open-world assumption* (OWA) of description logics and the *closed-world assumption* (CWA) of production systems. In that section we also introduce the main definitions relating production systems and ontologies. After that, in Section 4.2.2, we axiomatize in Fix Point logic the possible executions of a production system augmented with an ontology Σ (a Production system). In that section we solve point 2.

4.2.1 Augmenting production systems with ontologies

In this section, we present the formal definitions extending a production system with a DL ontology (*TBox*) Σ. The main new concepts are production systems, run, and computation tree. One of the most critical points of this section is Definition 4.2.6 where we formally state when a rule is fired in a working memory, and how the resulting working memory looks like. That definition summarize the answer to the problem presented in point 1 in the introduction of Section 4.2

A notable difference with traditional production systems (Prod. Sys.), is that the set of predicates in the Prod. Sys. is divided between description logic (DL) predicates, i.e. predicates which may occur in the ontology, and production system predicates. In this way we interpret DL predicates under OWA, and Prod. Sys. predicates under CWA. Formally, given a production system with a FO signature τ we divide the set of predicates into two disjoint sets: DL predicates (P_{DL}) and production system predicates (P_{PS}). Now we proceed to define a Generic Production System.

Definition 4.2.1 (Production System)**.** *Given a DL ontology Σ a* Generic Production System *is a tuple $PS = (\tau, \Sigma, L, R)$, where*
– $\tau = (P, C)$ is a first-order signature, with P a set of
predicate symbols (DL and Prod. Sys. predicates), each with an associated nonnegative arity, and C a nonempty set of constant symbols,
– Σ is a DL $TBox$ which predicates belong to P_{DL}
– L is a set of rule labels, and
– R is a set of rules, which are statements of the form

$$r : \textit{ if } \phi_r(\vec{x}) \textit{ then } \psi_r(\vec{x})$$

where

- $r \in L$,
- ϕ_r *is an FO formula of τ and free variables $\vec{x}$. and*
- $\psi_r(\vec{x}) = (a_1 \wedge \cdots \wedge a_k \wedge \neg b_1 \wedge \cdots \wedge \neg b_l)$, *where $a_1, \ldots, a_k, b_1, \ldots, b_l$ are atomic formulas with free variables among $\vec{x}$, such that no a_i and b_j share the same predicate symbol,*

We assume each rule has a distinct label.
□

Given a Prod. Sys. , the corresponding *language* $\mathcal{L}$ of the Prod. Sys. is $P \cup L \cup C$.

We view rule labels $r \in L$ also as n-ary predicates, where n is the number of free variables in the condition ϕ_r; with AL we denote the set of ground atoms constructed from the predicate symbols in L and the constants in C.

We assume that AT denotes the set of equality-free ground atomic formulas (atoms) of τ. With LT we mean the set of equality-free ground literals of τ.

Definition 4.2.2. *Given a production system $PS = (\tau, \Sigma, L, R)$ such that the production rule*

$$r : \textit{ if } \phi_r(\vec{x}) \textit{ then } \psi_r(\vec{x})$$

is in PS, we define:

- $\psi_r^{add} = a_1 \wedge \cdots \wedge a_k$
- $\psi_r^{removePS} = \neg b_j \wedge \cdots \wedge \neg b_k$ *such that the predicate symbols in $b_j \ldots b_k$ are in P_{PS}*
- $\psi_r^{removeDL} = \neg b_i \wedge \cdots \wedge \neg b_h$ *such that the predicate symbols in $b_i \ldots b_h$ are in P_{DL}.*

□

Before going on with the rest of the definitions, we proceed to set some notation. Let S be a set of literals, by $\overline{S}$ we mean the set composed by the elements of S complemented with respect to $\neg$.

Definition 4.2.3. *(Working Memory) A* working memory *WM for PS is a subset of AT.*
□

Definition 4.2.4. *Let $PS = (\tau, \Sigma, L, R)$ be a generic production system and let WM_0 be a working memory. Then a* concrete production system *is a pair (PS, WM_0).* □

In the beginning of this section we split the predicates of the signature of a Prod. Sys. in DL predicates (P_{DL}) and Prod. Sys. predicates (P_{PS}). Now we proceed to split the predicates in the action of a rule, and the predicates of a working memory. We will use these sets of predicates later on in this section:

Given a production system $PS = (\tau, \Sigma, L, R)$, let $r \in L$ be a rule and σ be a valuation, then:
–$NegDL_r(\sigma) = \{p(c) : p(c)$ occurs in $\sigma(\psi_r^{removeDL})\}$
–$PosDL_r(\sigma) = \{p(c) : p(c)$ occurs in $\sigma(\psi_r^{add})$ and $p \in P_{DL}\}$
–$NegPS_r(\sigma) = \{p(c) : p(c)$ occurs in $\sigma(\psi_r^{removePS})\}$

–$PosPS_r(\sigma) = \{p(c) : p(c)$ occurs in $\sigma(\psi_r^{add})$ and $p \in P_{PS}\}$

We also need to split a working memory WM in two parts:
–$WM_{DL} = \{p(c) : p(c) \in WM$ and $p \in P_{DL}\}$
–$WM_{PS} = \{p(c) : p(c) \in WM$ and $p \in P_{PS}\}$

(Note that p here stand for a predicate).

The next definition is one of the most relevant in this section. Here we formalize the interaction between the ontology and the Prod. Sys. . We define when a rule's condition is satisfied in a working memory given a $TBox$ (i.e., an ontology) and a valuation.

Definition 4.2.5 (Holds)**.** *An interpretation $\mathcal{M}$* satisfies *an FO formula ϕ with a valuation σ, relative to a working memory WM, denoted $\mathcal{M}, \sigma \models_{WM} \phi$, iff*

- $\phi = p(\vec{x})$, $p \in P_{PS}$, *and* $p(\sigma(\vec{x})) \in WM$;
- $\phi = p(\vec{x})$, $p \in P_{DL}$, *and* $\mathcal{M} \models p(\sigma(\vec{x}))$;
- $\phi = \neg\psi$ *and* $\mathcal{M}, \sigma \not\models_{WM} \psi$
- $\phi = \psi_1 \wedge \psi_2$, $\mathcal{M}, \sigma \models_{WM} \psi_1$, *and* $\mathcal{M}, \sigma \models_{WM} \psi_2$;
- $\phi = \exists x : \psi$ *and there is some valuation σ' such that $\sigma'(y) = \sigma(y)$ for every variable $y \neq x$ and $\mathcal{M}, \sigma' \models_{WM} \psi$.*

A formula ϕ holds *in a working memory WM with a valuation σ, relative to a theory Σ, denoted $\Sigma, \sigma \models_{WM} \phi$, iff $\mathcal{M}, \sigma \models_{WM} \phi$ for every model $\mathcal{M}$ of $\Sigma \cup WM$.* □

Now we are ready to define when a rule is fireable, and what is the content of the resulting working memory.

Definition 4.2.6 (Firable)**.** *Let $PS = (\tau, \Sigma, L, R)$ be a production system. A rule $r \in L$ is* fireable *in a working memory WM with a valuation σ iff*

- *$WM \cup \Sigma$ is consistent, i.e., it has a model*
- *ϕ_r holds in WM with a valuation σ*
- *ψ_r does not hold in WM with a valuation σ*
- *The resulting working memory WM' is consistent with Σ*

Where the resulting working memory WM' after firing r in WM is:

$$WM' = WM'_{DL} \cup WM'_{PS}$$

where:
–$WM'_{DL} = ((WM_{DL} - NegDL_r(\sigma)) \cup PosDL_r(\sigma))$
–$WM'_{PS} = ((WM_{PS} - NegPS_r(\sigma)) \cup PosPS_r(\sigma))$
□

Consider a production system with a rule r with action $\neg p(c)$. Suppose that WM is the working memory resulting from the application of r. It is worth noting that if $p \in P_{PS}$, then the formula $\neg p(c)$ holds in WM_0 thanks to the CWA in Prod. Sys. . Instead if $p \in P_{DL}$ (assuming the ontology is empty) the $\neg p(c)$ does not hold in WM. It is due to the fact that $\Sigma \cup WM \not\models \neg p(c)$.

Note that filtering out the DL augmentation in this definition, what we obtain is a classical production system.

Definition 4.2.7 (Run)**.** *Let $PS = (\tau, \Sigma, L, R)$ be a generic production system. A* Run $Run^{PS}_{WM_0}$ *for a Σ concrete production system (PS, WM_0) is an (AT)-labeled sequence (S, V) , such that:*

- $S = 0 \dots n \dots$
- $V(0) = WM_0$,
- *and for each element $n \in S$ such that it has a successor, there is some rule r, with some valuation σ such that r is fireable in the working memory $WM = V(n) \cap AT$ using σ, and $V(n+1) = WM' \cup \{r(\sigma(\vec{x}))\}$, with*

$$WM' = WM'_{DL} \cup WM'_{PS}$$

as defined in Definition 4.2.6.

There are no other nodes in $Run^{PS}_{WM_0}$. □

Now we are ready to define *computation trees* which stands for all the possible runs of the system.

Definition 4.2.8. *A* computation tree *for a concrete production system $(\Sigma - PS, WM_0)$, denoted $CT^{\Sigma-PS}_{WM_0}$, is an $(AT \cup AL)$-labeled tree (T, V) such that the root of T is 0, $V(0) = WM_0$, and for each node $n \in T$, every rule r, and every valuation σ such that r is fireable in the working memory $WM = V(n) \cap AT$ using σ, there is a child node $n' \in T$ of n such that $V(n') = WM' \cup \{r(\sigma(\vec{x}))\}$, with $WM' = WM'_{DL} \cup WM'_{PS}$ as defined in Definition 4.2.6. There are no other nodes in $CT^{\Sigma-PS}_{WM_0}$.* □

4.2.2 Axiomatization

In this section we present the axiomatization of a run of a production systems. Given a language $\mathcal{L}$ of a Prod. Sys. , the corresponding language $\mathcal{L}_{\Phi_{PS}}$ of the target FPL

axiomatization contain the predicate symbols and constants in $\mathcal{L}$ plus the predicates R, B and A.

We capture the structure of the run using the binary predicate R. To enforce a sequence structure on R, we introduce the predicate A which elements are totally ordered. The predicate A contains the elements from the domain which are in R.

The arity of the predicates in $\mathcal{L}_{\Phi_{PS}}$ corresponding to predicate names in $\mathcal{L}$ is increased by one, and the first argument of each predicates will signify the state; $p(y, x_1, \ldots, x_n)$ intuitively means that $p(x_1, \ldots, x_n)$ holds in state y.

Given a model $\mathcal{M}$, let $\mathcal{M}^c$ be a model such that $(t_1, \ldots, t_n) \in p^{\mathcal{M}^c}$ iff $p(c, t_1 \ldots t_n) \in \mathcal{M}$. We say that an element c **satisfies** the $TBox$ Σ iff $\mathcal{M}^c$ is a model of Σ.

After setting the structure of the model, we define the necessary components of the formula comprising the axiomatization. These components encode the constrains and requirements w.r.t. to the $TBox$ and the production system. Informally, the three main concepts embedding each component are :

- *Time*: Each element in R represent a point in the execution, a state.
- *TBox*: We require that each element in R (each state) satisfies Σ
- *Run*: We constrain the relation between one element of the sequence and its successor depending if it is an intermediate state in the execution of the production system, or an element representing the end of the run.

A greatest fix point composed of these components restricts the models to the ones which represent a run of the Prod. Sys. .

A notable difference with the axiomatization of traditional Prod. Sys. presented in [14] is that the Frame axiom is split into two. One handles the Prod. Sys. facts, i.e., the atoms which predicate symbol is in P_{PS}. The other one handles DL facts. The main difference between these two is the DL frame axiom only keeps unchanged atoms which are true (and not removed) from one state to the next state, while it does not constrain the behavior of DL atoms which are false.

Before explaining the axiomatization, let us define a formula which encodes when a rule is fireable:

Let

$$Firable_r(y, \vec{x}) = \phi_r(y, \vec{x}) \wedge \neg(\psi_r^{add}(y, \vec{x}) \wedge \psi_r^{removePS}(y, \vec{x}))$$

Note that in this formula is when the conflict resolution strategy shows up in the logic embedding. It essentially encodes the conflict resolution strategy:

- the current state satisfies the condition ϕ_r,

- the rule's action must change the working memory, therefore we check that $\psi_r^{add}(y, \vec{x}) \wedge \psi_r^{removePS}(y, \vec{x})$ does not hold in the state where r is going to be applied, i.e. the predecessor of y.

Note that we ignore $\psi_r^{removeDL}(y, \vec{x})$. It is due the fact that $\psi_r^{removeDL}(y, \vec{x})$ dont impose any constrain but remove one. It does not state what should hold or not in a given working memory, but what should not necessarily be assumed to be true anymore.

In the remainder, let $PS = (\tau, \Sigma, L, R)$ be a generic production system and let WM_0 be a working memory. We first define the *foundational axioms*, which encode the basic structure of the models and the sequence shape of R.

We assume that the predicates B (signifying the start state), A and the binary predicate R are not in $P \cup L$.

Now we define the foundational axioms Γ_{found}, which enforce the sequence shape of R:

- There is a total order $<$ over the elements in A
- $\forall x, y : R(x, y) \rightarrow A(x) \wedge A(y)$
- $\forall x, y : R(x, y) \leftrightarrow x < y \wedge \nexists z : x < z < y$
- $\forall x, y : B(x) \rightarrow A(x) \wedge (x \neq y \rightarrow x < y)$

Now the components of the axiomatization:

Root

$$B(y) \wedge (\bigwedge_{r \in L} \forall \vec{x} : \neg r(y, \vec{x}))$$

TBox The *TBox* Σ holds in every state

$$(\bigwedge_{T \in \sigma} T(y))$$

RApp

$$(\bigwedge_{r \in L} \forall \vec{x} : r(y, \vec{x}) \rightarrow \psi_r^{add}(y, \vec{x}) \wedge \psi_r^{removePS}(y, \vec{x}))$$

Appl

$$(\bigwedge_{r \in L} \forall \vec{x} : \exists w(R(y, w) \wedge r(w, \vec{x})) \rightarrow Fireable_r(y, \vec{x}))$$

FrameDL

$$\bigwedge_{p \in P_{DL}} \forall x_1, \ldots, x_n([p(y, x_1, \ldots, x_n) \rightarrow (\forall w : R(y, w) \rightarrow p(w, x_1, \ldots, x_n) \vee (\bigvee_{r \in L. \psi_r^{removeDL}(\vec{z}) = \ldots \neg p(t_1, \ldots, t_n) \wedge \ldots} \exists \vec{z} : r(y, \vec{z}) \wedge x_1 = t_1 \wedge \cdots \wedge x_n = t_n))]$$

Frame $\bigwedge_{p\in P_{PS}} \forall x_1,\ldots,x_n([p(y,x_1,\ldots,x_n) \rightarrow (\forall w : R(y,w) \rightarrow$
$p(w,x_1,\ldots,x_n) \vee (\bigvee_{r\in L.\psi_r(\vec{z})=\ldots\neg p(t_1,\ldots,t_n)\wedge\ldots} \exists \vec{z} : r(y,\vec{z}) \wedge x_1 = t_1 \wedge \cdots \wedge x_n =$
$t_n))] \wedge [\neg p(y,x_1,\ldots,x_n) \rightarrow (\forall w : R(y,w) \rightarrow$
$\neg p(w,x_1,\ldots,x_n) \vee (\bigvee_{r\in L.\psi_r(\vec{z})=\ldots p(t_1,\ldots,t_n)\wedge\ldots} \exists \vec{z}.r(y,\vec{z}) \wedge x_1 = t_1 \wedge \cdots \wedge x_n = t_n))])$

NoFirable

$$(\bigwedge_{r\in L} \neg Fireable_r(y,\vec{x}))$$

Firable

$$\exists \vec{x} : \bigvee_{r\in L} (Fireable_r(y,\vec{x}))$$

1Rule

$$(\bigvee_{r\in L} \exists \vec{x} : r(y,\vec{x})) \wedge (\bigwedge_{r\in L} (\exists \vec{z} : r(y,\vec{z}) \rightarrow \neg \bigvee_{r'\in L \& r'\neq r} \exists \vec{x} : r'(y,\vec{x})))$$

Only A rule can not be applied twice in the same state.

$$(\bigwedge_{r\in L} \forall \vec{x} : r(y,\vec{x}) \rightarrow \exists^{=1} \vec{z} : (r(y,\vec{z}))$$

WM-PS

$$\bigwedge_{p\in P_{PS}} \forall x_1 \ldots x_n (p(y,x_1 \ldots x_n) \leftrightarrow$$
$$\bigvee \{x_1 = c_1 \wedge \cdots \wedge x_n = c_n \mid p(c_1,\ldots,c_n) \in WM_0\})$$
$$\bigwedge_{p\in P_{PS}} \forall x_1 \ldots x_n (\neg p(y,x_1 \ldots x_n) \leftrightarrow$$
$$\bigvee \{x_1 = c_1 \wedge \cdots \wedge x_n = c_n \mid \neg p(c_1,\ldots,c_n) \in WM_0\})$$

WM-DL

$$\bigwedge_{p(c_1,\ldots,c_n)\in WM_0 \text{ and } p\in P_{DL}} p(y,c_1,\ldots,c_n)$$
$$\bigwedge_{\neg p(c_1,\ldots,c_n)\in WM_0 \text{ and } p\in P_{DL}} \neg p(y,c_1,\ldots,c_n)$$

The main differences with the axiomatization presented in [14] are

1. **Appl** We ask every node resulting from a rule application, to be successor of a node consistent with Σ
2. **FrameDL** Only constrains the positive atoms which were not removed to stay over time
3. **TBox** To mark consistent nodes
4. **NoFirable** No consistent nodes are no candidates for fire a rule
5. **Firable** If a consistent node satisfies a rule, then some rule is fired and there is a unique successor which is consistent with the ontology.
6. **WM-DL** Which facts should hold in the initial WM, but not exclusively

$$\textbf{Intermediate} = \textbf{RApp} \wedge \textbf{TBox} \wedge \textbf{1Rule} \wedge \textbf{Only} \wedge \textbf{Appl} \wedge \textbf{Frame} \wedge \textbf{FrameDL} \wedge \textbf{Firable} \wedge \neg B(y)$$

$$\textbf{End} = \textbf{RApp} \wedge \textbf{TBox} \wedge \textbf{1Rule} \wedge \textbf{Only} \wedge \textbf{Frame} \wedge \textbf{FrameDL} \wedge \textbf{NoFirable} \wedge \neg B(y)$$

Analogous to the propositional case, we defined a formula that captures the behavior of PS:

$$\begin{aligned}\Phi_{PS} = (\exists y : (\textbf{Root} \wedge \textbf{NoFirable}) \wedge \textbf{TBox} \vee (\textbf{Root} \wedge \textbf{Frame} \wedge \textbf{FrameDL} \wedge \textbf{TBox} \wedge \textbf{Appl} \\ \wedge \textbf{Firable} \wedge \forall w(R(y,w) \rightarrow (\nu.X.y.(\textbf{Intermediate} \vee \textbf{End}) \\ \wedge \forall w(R(y,w) \rightarrow X(w)))(w))))\end{aligned}$$

The following notation will be useful. The formula ***Run***$(r_0 \dots r_n)$ denotes an order in the rules execution.

Definition 4.2.9. *Given a set of rules rules $r_1 \dots r_n$ of a Prod. Sys. , let* Run$(r_1 \dots r_n)$ *denote the following formula:*

$$B(y_1) \wedge \bigwedge_{i=1\dots n} R(y_i, y_{i+1}) \wedge r_i(y_i, \vec{x_i})$$

□

Given a rule r of a Prod. Sys. PS we say that r is **fireable** in the initial state, if r is fireable in the initial state of every model of the axiomatization of PS. That is

$$\Phi_{PS} \cup \Sigma_{found} \models \exists y_1, \vec{x} : B(y_1) \wedge Firable_r(y_1, \vec{x})$$

And sequence of rules $[r_1 \dots r_n, r]$ is fireable in the initial state (or just fireable), if for every model of the run $r_1 \dots r_n$, the rule r is fireable after r_n. That is, $[r_1 \dots r_n]$ is fireable and

$$\Phi_{PS} \cup \Sigma_{found} \models \exists y_1 \dots y_{n+1}, \vec{x} : Run(r_1 \dots r_n) \rightarrow Firable_r(y_{n+1}, \vec{x}_1)$$

We say that a formula in $\mathcal{L}_{\Phi_{PS}}$ is **timeless** if the predicate names R, B and A do not occur in α.

Definition 4.2.10. *Given a Prod. Sys. , a timeless closed formula α in $\mathcal{L}_{\Phi_{PS}}$ and a set of rules $r_1 \dots r_n$, we say that α holds **after** $r_1 \dots r_n$ if and only if*

$$\Phi_{PS} \cup \Sigma_{found} \models \exists y_1 \dots y_{n+1}, \vec{x_1} \dots \vec{x_n} : \text{Run}(r_1 \dots r_n) \rightarrow \alpha(y_{n+1})$$

□

Theorem 4.2.11 states the correspondence between the runs of the production systems, and the models of our formalization. Informally, the soundness theorem says that given a production system, its formalization $\Phi_{PS} \cup \Sigma_{found}$, and the n first states of R, there is a run which contain the sequence $1 \dots n$ such that the models in the $i - th$ state, are the same as the ones in the sequence point i.

Theorem 4.2.11. *(Soundness) Let $PS = (\tau, \Sigma, L, R)$ be a production system and let (PS, WM_0) be a concrete production system. Then for any executable sequence of rules $[r_1 \dots r_n]$; there is a run $Run^{PS}_{WM_0} = (S, V)$ of the Prod. Sys. and valuations $\sigma_1 \dots \sigma_n$ such that*

1. *$S \supseteq 1 \dots n+1$*
2. *r_i is fireable in $V(i)$ with valuation σ_i*
3. *For any formula α in $\mathcal{L}_{PS}$, α holds after $[r_1 \dots r_n]$ iff $V(n+1) \models_\Sigma \alpha$ (modulo the state argument)*

Theorem 4.2.12. *(Completeness) Let $PS = (\tau, \Sigma, L, R)$ be a production system and let (PS, WM_0) be a concrete production system. Let $Run^{PS}_{WM_0} = (S, V)$ be a run of the Prod. Sys. Then for any finite sequence $1 \dots n+1 \subseteq S$ where r_i is fireable in $V(i)$ with valuation σ_i, the sequence $[r_1 \dots r_n]$ is executable and for any formula α in $\mathcal{L}_{PS}$, α holds after $[r_1 \dots r_n]$ iff $V(n+1) \models_\Sigma \alpha$ (modulo the state argument)*

4.3 Conclusions

We have presented two alternative approaches to combining production rules and OWL ontologies: a loose coupling and a tight coupling semantics.

In the loose coupling approach we augmented the production rule semantics in a minimal way: we only defined the notion of satisfaction of a rule condition, given a variable substitution, and we defined the effect actions have on the working memory. Condition

satisfaction was defined based on entailment from the ontology plus the working memory (viewed as an ABox). We considered two approaches for the semantics of actions: the formula-based approach, where actions effect addition or removal of facts from the working memory, and the model-based approach, which defines the effects on the model-level, which allows one to take the ontology into account when modifying the working memory. As we have seen, it is unfortunately not always possible to materialize the resulting working memory, but there are known ontology languages for which this is possible.

In the tight coupling approach, we defined an integrated semantics and an axiomatization of this semantics in fixpoint logic (FPL). Because of the integrated nature of this semantics, it encompasses more aspects of the production system: for example, the conflict-resolution strategy is an integral part of the definition of the semantics, as well as the axiomatization, although both have been constructed in such a way as to allow easy integration of other conflict-resolution strategies. The axiomatization may be used to analyze combinations and check certain properties, such as termination. Actual analysis of production rules and ontology combinations using the FPL axiomatization is future work.

Chapter 5

Production Rules over OWL Ontologies

5.1 Introduction

This chapter describes some issues encountered when trying to use production rules (PRs) over OWL ontologies.

Our main concern is to facilitate the use of PRs over ontologies expressed in OWL as opposed to traditional objects. After a quick review of today's state of the art in mixing production rules and ontologies, we focus on a "loose-coupling" approach that consists of implementing a new PR engine that delegates all ontological processing to an OWL engine.

Based on the theoretical framework for satisfaction of conditions and execution of actions introduced in Deliverable D3.2 [35], we studied the issues resulting from the combination of PRs and ontologies. The deep analysis of these issues allows us to enhance the implementation of XPR⟨OWL⟩, an execution engine for production rule over ontologies, and gives us a better comprehension of the impact of replacing a traditional object model with OWL in a PR engine, as well as the limitations of the "loose-coupling" approach.

The identified issues belong to two categories. On one hand, those on the **condition part** of a PR, such as matching sets, counting of property values, or user predicates (or connectives) in conditions. On the other hand, the issues on the **action part** of a PR, such as retracting individuals, or maintanining consistency. As our analysis will clearly show, many typical constructs in the object world can be captured precisely and formally as OWL statements. It is also clear that there are many object-oriented assumptions and features that are at odds with the assumptions or expressive power of OWL. Hence, our general approach has been to identify a *core* sublanguage encompassing all XPR⟨OWL⟩ constructs that can be delegated safely and correctly to an OWL reasoner. For the parts of XPR⟨OWL⟩ not fitting this core sublanguage, specific compromise solutions addressing the identified issue are proposed.

The prototype implementation, based on OWL-API[1] and Jena [20] (see [40] for the details of the implementation), was driven by two aims: (a) bring the theoretical framework and the implementation in line as much as possible, and (b) make sure that the implementation respects the OWL semantics, while preserving the peculiarities of the PR engine. It must be noticed that we have already seen that using the Jena engine does not guarantee faithfulness to the OWL semantics. In the loose-coupling approach, production rules and OWL only touch each other when (1) checking satisfaction of conditions and (2) updating the working memory (in our case, the individuals of an OWL ontology). But we noticed that the use of OWL ontologies also impacted other components of the PR engine, such us pattern matching, navigation or assertions.

This chapter is organized as follows. The theoretical rule language is first briefly recalled. Next, the issues that arise up when trying to adapt a practical production rule language initially dedicated to objects to OWL are detailed. The issues related to the condition part of the production rules are isolated from the issues related to the action part. A discussion finally follows analyzing the impact of replacing the object model of a PR engine with OWL, and explicating some of the limitations of the "loose-coupling" approach.

5.2 Theoretical framework

In the rule language we are currently considering in the theoretical framework, we distinguish between conditions and actions. A rule may be fired for a particular variable substitution S if the condition is satisfied (defined below) for this substitution. In case the rule is fired, we say how the working memory is updated, based on the action. We do not say anything about the conflict resolution strategy.

We assume the following:

- A description logic knowledge base K (*i.e.*, the ontology). This is immutable.
- A working memory WM, which is a set of variable-free atomic formulae; *i.e.*, atomic class and property membership statements.

5.2.1 Conditions

We consider here only positive patterns, and we consider only predicates that are used in the DL knowledge base.

A rule has a set of positive patterns p that are constant-free atomic formulae of the forms $A(x)$ or $R(x, y)$, where A is the name of an OWL class (*i.e.*, a *concept*) and R is an OWL property (*i.e.*, a *role*). A rule also has a condition c, which is a first-order logic formula with free variables among the variables appearing in p. When writing the condition, the

[1] http://owlapi.sourceforge.net/

set of patterns can be viewed as a conjunction and can, in turn, be conjoined with the condition. We can thus write conditions of the form:

$$\texttt{Person}(x) \ \wedge \ \exists y(\texttt{hasChild}(x, y))$$

which matches with all persons who have at least some child.

A rule is σ-fireable if, for some variable substitution σ:

$$\texttt{K} \cup \texttt{WM} \ \models \ \sigma(p), \sigma(c)$$

Where, `K` can be viewed as the ontology, with `WM` being the instance data of the ontology. The symbol $\models$ denotes the logical entailment relation.

Consider the example ontology above and the variable substitution $\sigma \ = \ \{\texttt{John}/x\}$. We have:

$$\texttt{K} \cup \texttt{WM} \ \models \ \texttt{Person}(\texttt{John}) \ \wedge \ \exists y(\texttt{hasChild}(\texttt{John}, y))$$

and thus our rule is σ-firable if and only it can be inferred from the `WM`'s `OWL` ABox data that an individual `John` exists who has at least one child.

5.2.2 Actions

The action of a rule consists of:

```
remove r
add a
```

where `r` and `a` are sets of atomic formulas whose variables appear free in the condition.

Consider, for example, an "aging" rule that makes someone old. The variable `x` is assumed to appear in the condition, and denotes an aging person:

```
remove hasAge(x,young)
add hasAge(x,old)
```

Note here that we add and remove facts; *i.e.*, *statements*. We do not add or remove *objects*.

Note also that adding statements to the working memory may make the working memory inconsistent with respect to the ontology. For example, one could do:

```
add owl:Nothing(a)
```

Thus making the ontology inconsistent by definition. The problem of having an inconsistency is that everything follows, and thus every rule becomes firable for every possible variable substitution, making them meaningless.

There are three possible ways to deal with such inconsistencies; we may:

1. stop execution as soon as an inconsistency is reached;
2. make firable only those rules that do not introduce inconsistency (this would require runtime reasoning); or,
3. fix inconsistencies on the fly (this is a really hard problem, and most methods for repairing inconsistencies require user intervention).

Where adding facts may introduce inconsistency, removing facts may lead to unexpected situations when removing is done naively, particularly when removing implicit (*i.e.*, implied) facts.

As an illustration, consider the following ontology `K`:

```
SubClassOf (:Student :Person)
```

and the working memory `WM`:

```
ClassAssertion(:Student :Kate)
```

We have that, combined, they imply that `Kate` is a `Person`:

$$\texttt{K} \cup \texttt{WM} \models \texttt{Person(Kate)}$$

Let's say, we want to have a rule that removes all persons:

```
IF Person(x) THEN remove Person(x)
```

Clearly, this rule is triggered for the person `Kate`. However, executing this rule does not result in removing anything about `Kate` from the working memory, since the fact `Person(Kate)` is not explicitly present in the `WM`; it is only implied by it.

5.2.3 OWL

The syntax of `OWL` is described in the `OWL2` specification. What is important for us is that `OWL` statements essentially correspond to description logic statements. In turn, description logic statements correspond to statements in first-order logic.

As an example, consider the following expressions stating that the concept of `parent` is equivalent to (defined as) the union of `mother` and `father` (*i.e.*, every parent is a mother or father, and vice versa), `John` is a `father`, and `John` has two children: `Mary` and `Kate`. Namely,

```
EquivalentClasses       ( :Parent ObjectUnionOf ( :Mother :Father ) )
ClassAssertion          ( :Father :John )
ObjectPropertyAssertion ( :hasChild :John :Mary )
ObjectPropertyAssertion ( :hasChild :John :Kate )
```

In DL syntax, this is written as:

$$\begin{aligned}&\texttt{Parent} = \texttt{Mother} \sqcup \texttt{Father}\\&\texttt{Father}(\texttt{John})\\&\texttt{hasChild}(\texttt{John}, \texttt{Mary})\\&\texttt{hasChild}(\texttt{John}, \texttt{Kate})\end{aligned}$$

This, again, corresponds to the following first-order logic theory:

$$\begin{aligned}&\forall x(\texttt{Parent}(x) \leftrightarrow \texttt{Mother}(x) \vee \texttt{Father}(x))\\&\texttt{Father}(\texttt{John})\\&\texttt{hasChild}(\texttt{John}, \texttt{Mary})\\&\texttt{hasChild}(\texttt{John}, \texttt{Kate})\end{aligned}$$

Since `John` has two children, `Mary` and `Kate`, one may be tempted to view the value of the `hasChild` predicate as a set `{Mary, Kate}`. Indeed, the `OWL-API` suggests that `hasChild` is a *set-valued property*, and returns the set `{Mary, Kate}` as the value of the property. However, one must keep in mind that, from the `OWL` language point of view, we have two separate statements, and thus we have a property that has two values.

5.3 Issues

We have identified a number of discrepancies between the proposed theoretical framework and the current `XPR⟨OWL⟩` implementation. These discrepancies are described below as issues on the condition part and the action part of the rule.

We have recognized that, in practice, it may be the case that users will desire certain functionality that is not easily captured in a nice theoretical framework. We can thus think of a core production rule language that has proper theoretical foundations in `OWL` semantics. This core may be extended with certain (*e.g.*, scripting) functionality as per request by the users. Of course, we strive to capture as much as possible of the useful functionality of the current JRules system in the theoretical framework. The solutions to the issues described below have in mind that we stay in the core fragment of the language that has solid foundations in the `OWL` semantics.

5.3.1 Condition Part

5.3.1.1 Matching Sets

The implementation of XPR⟨OWL⟩ currently allows matching sets. From a semantic point of view it is not always clear what this should mean, in particular when matching sets of statements. Also sets of values may be a problem, or at least a discrepancy with the theoretical framework. However, in some cases such conditions can be reduced to conditions without sets, in particular when the set is only used for enumeration. For example, the following condition will match with all children of any parent, and thus the rule is fired for every child y of every parent x:

```
Parent(x) and hasChild(x,y)
```

Note that sets (and other aggregates) are just examples of monoids. A set is just a value, and any basic calculus (say, the λ-Calculus or the Relational Calculus) can accommodate them without complication. See for example the Fegaras-Maier SIGMOD'95 paper [19]. However it is simpler to start without sets or aggregates over collections.

Incidentally, the above "special case" of use of collections is deemed useful as it "happens" to match the semantics of:

$$\forall x\ \exists y.\ Parent(x)\ \wedge\ hasChild(x, y).$$

All such ("operational") iterations over collections may be thus formally interpreted using the paradigm of monoid comprehensions, which too have a simple set-theoretic (and thus FOL) semantics.

The issue is one of understanding what is meant. Let's take as an example the condition `Parent(x)`. If variable `x` is matched with individual `John`, we can check whether `Parent(John)` is entailed. However, if `x` is matched with a set such as `{John,Jack}`, it is unclear what to do, since `Parent({John,Jack})` is not a well-formed FOL expression. This means we need to figure out what to do in such cases.

Here is a more detailed example with sets to help our understanding.

```
rule MyRule1 {
  when {
    foreach person as myonto:Person {
      myonto:friends friends;
    };
    foreach friend from friends;
    if (friend.age > 24);
  }
  then {
    insert friend;
  }
}
```

In the condition part of Rule `MyRule1`, it is assumed that:

1. the range of the property `myonto:friends` is `myonto:Person` and the property is not functional; and,
2. the variable `friends` has the type `set<myonto:Person>` because operationally `OWL-API` will return a set for the objects when passed a subject and a property.

Therefore, it should be possible to use it as a set in the rule and in function calls. For instance here, it is used as the source collection value of a `foreach` pattern in the second part of the condition. In the nested `when` condition, only the friends of the persons of the WM whose `age` is greater than 24 are relevant.

```
rule MyRule11 {
  when {
    foreach person as myonto:Person {
      myonto:friends friends;
    };
  }
  then {
    insert friends;
  }
}
```

The action part of `MyRule11` is syntactic sugar to add all the elements of the set to the working memory (an `OWL` ontology) one after the other (thus saving an interation loop). It will not add the set itself as a single individual as it has no meaning in `OWL`. Only values with an `OWL` type (*a.k.a.*, *description*) can be added to the WM. Thus a value with a type in the rule type system built over the `OWL`'s own type system will be refused and a value with an `OWL` `DataRange` type will also be refused as compile-time errors.

```
rule MyRule2 {
  when {
    let friend_collector = new set<myonto:Person>();
    do {
      aggregate friend_collector(friend);
    }
    when {
      foreach as myonto:Person {
        myonto:friends friends;
      };
      foreach friend from friends;
      if (friend.age > 24);
    }
  }
  then {
    doSomething(friend_collector);
  }
}
```

Rule `MyRule2`'s condition illustrates how sets may be combined with a `do` pattern. This is useful to collect only relevant data from patterns. In the nested `do` condition part, the relevant friends are collected.

In the condition part of `MyRule2`, the pattern is used to build a set. In the action part, a *single instance* of the rule is created and not as many as the number of `Person` instances times the number of instances in the set `friends`.

5.3.1.2 Counting of property values

5.3.1.2.1 Issue The current implementation relies on the set of values for a particular property returned by the `OWL-API`. The problem is that, in the `OWL` world, several of these property values may actually be identifiers of the same object and there may be values of the property that do not have an associated identifier, and are thus not returned by the `OWL-API`.

For example, to test if `John` has more than one child, implementation will ask the `OWL-API` for all known children of `John`, in this case the `OWL-API` will return the set `{Mary, Kate}`. From this it is not valid to conclude that `John` has two children, since `Mary` and `Kate` may actually be the same person *e.g.*, `Mary` might have changed her name to `Kate` at some point during her life).

5.3.1.2.2 Solution The solution would be to query the `OWL` ontology when checking the condition. For example, the following may be part of the condition:

```
x.hasChildren.size > n
```

where x and n are variables. At the time of execution, these variables will be instantiated; *e.g.*, x is instantiated with John and n is instantiated with the number 1. The resulting condition is thus:

```
John.hasChildren.size > 1
```

which corresponds to the OWL statement:

```
ClassAssertion(ObjectMinCardinality(2 :hasChild) :John)
```

It is thus possible to check entailment of this statement from K ∪ WM. If this statement is entailed, it is then indeed true that John has more than one child.

Note that the above is just an example where the solution is to generate an appropriate OWL statement that expresses exactly the needed semantics. Indeed, it is quite possible to find examples for which generating an OWL expression with equivalent semantics is non-trivial or simply impossible. For instance, if a rule's condition requires to compare the size of two sets in the PR language, this cannot be reduced to checking an appropriate OWL entailment.

The general solution, then, would be to restrict the core sublanguage only to those expressions that can be translated into equivalent OWL. A more systematic generic solution would thus require to be able to identify which expressions belong in the core and which do not. For the former an appropriate equivalent OWL can be generated as required; as for the latter, such expressions are to be flagged as problematic (*i.e.*, reported as exceptions).

5.3.1.3 User predicates in condition

5.3.1.3.1 Issue In the implementation, it is possible to include user-defined functions in a rule condition. Of course, we cannot specify the semantics of such functions completely, but such functions would simply reduce to user-defined predicates in the theoretical framework. The only remaining problem is that user defined functions may take sets as arguments.

5.3.1.3.2 Solution For now, disallow the users to define functions that takes sets as arguments.

5.3.1.4 Connectives in conditions

The use of certain logical connectives in the conditions (disjunction, negation, universal quantification) raise problems[2], because they operate only on the variable substitutions that have been matched, and not on the OWL models.

Consider, for example, the condition:

```
not(Person(Jake))
```

Arguably, this condition should only be satisfied if:

$$\mathtt{K} \cup \mathtt{WM} \models \neg\mathtt{Person}(\mathtt{Jake})$$

i.e., if it is known that Jake is not a person. However, according to the current mechanism, the condition is satisfied if it is not known that Jake is a person:

$$\mathtt{K} \cup \mathtt{WM} \not\models \mathtt{Person}(\mathtt{Jake})$$

So it may still be the case that there are some models in which Jake is a person, and it is thus arguably not reasonable to assume that he is not.

For a disjunction example, let us consider an OWL ontology K with the following statements:

$$\mathtt{Person} \sqsubseteq (\mathtt{Man} \sqcup \mathtt{Woman})$$
$$\mathtt{Person}(\mathtt{Jane})$$

which state that every Person is a Man or a Woman and that Jane is a Person. Let us then consider the disjunctive condition :

$$\mathtt{Man}(\mathtt{Jane}) \vee \mathtt{Woman}(\mathtt{Jane}).$$

Using the current (closed-world) mechanism in the XPR⟨OWL⟩ implementation, this condition is not satisfied, because :

$$\mathtt{K} \cup \mathtt{WM} \not\models \mathtt{Man}(\mathtt{Jane})$$
$$\mathtt{K} \cup \mathtt{WM} \not\models \mathtt{Woman}(\mathtt{Jane})$$

which states that neither Man(Jane) nor Woman(Jane) is entailed by the ontology. However, because of the axiom:

$$\mathtt{Person} \sqsubseteq (\mathtt{Man} \sqcup \mathtt{Woman})$$

it comes that Jane is either a Man or a Woman in every model of the ontology and thus Man(Jane) ∨ Woman(Jane) is entailed:

$$\mathtt{K} \cup \mathtt{WM} \models \mathtt{Man}(\mathtt{Jane}) \vee \mathtt{Woman}(\mathtt{Jane}).$$

[2]Many of these problems derive from the discrepancies between ontologies and rules, due to open vs. close world as introduced in D3.2 [35, 5.4.1]

Here is an example of the issue with using "for all" quantification in a PR condition.

Consider an ontology K:

```
Person(Jane)
Woman(Jane)
```

stating that Jane is a Person and a Woman, and a condition:

$$\forall x \ \Big(\texttt{Person}(x) \implies \texttt{Woman}(x)\Big)$$

that is satisfied if every Person is a Woman. Now, the ontology does not entail that every Person is a Woman; indeed:

$$\texttt{K} \cup \texttt{WM} \not\models \texttt{Person} \sqsubseteq \texttt{Woman}.$$

However, the condition is satisfied using the current XPR⟨OWL⟩ mechanism, since every Person that is explicitly mentioned in the ontology is also a Woman.

From a semantic point of view, to consider conditions with arbitrary logical connectives as queries on the DL knowledge base poses no problem. However, it is known that answering such queries is a hard problem, and not decidable in general. From a practical point of view, it is also hard to get reasoning support for such arbitrary queries. Many DL reasoners only support tree-shaped queries; *i.e.*, queries that correspond to concept expressions.

A possible solution would be to see which kind of queries can be reduced to tree-shaped queries, and in which cases it is possible to split the matching of the condition in various parts. In particular, this can always be done in the case of a conjunction between two formulas that do not share any quantified variables.

Yet another solution would be to view OWL statements as *oracle* calls separate from the PR world, and view the Boolean connectives as an algebra that just does simple syntactic manipulation.

5.3.1.5 A note on iterations

In the XPR⟨OWL⟩ language, it is possible to express iterations over matching variable substitutions in the conditions. This means that the rule is applied to all matching substitutions, rather than just one. In the theoretical framework, we currently only consider matching individual variable substitutions.

Consider the rule:[3]

```
foreach x: Person(x) do remove(x)
```

[3]This is informal syntax.

meaning to remove all `Person` objects from the working memory. This rule is essentially the same as the rule:

```
if Person(x) then remove(x)
```

along with a conflict resolution strategy that says that when this rule is applied, it is applied for all matching substitutions before any other rule is applied.

There are various possible ways of dealing with this mismatch, including:

- Saying that the theoretical framework is only concerned with matching individual substitutions, and thus not say anything about the order of rule applications.
- Including a special kind of rule in the theoretical framework that is fired for all matching substitutions, rather than just one.

For now, we choose the former.

5.3.2 Action Part

5.3.2.1 Retracting individuals

5.3.2.1.1 Issue In the object world of production rules it is typical to retract individuals. The current implementation tries to mimic this behavior by retracting all statements found using the `OWL-API` involving the particular individual. Indeed, actual retraction of an individual would mean changing the signature of the language, which is a problem from a semantic point of view. The current `XPR⟨OWL⟩` implementation using `OWL-API` mimics this behavior, since `OWL-API` only allows querying objects that are present in the ontology, and removing all statements involving objects, including membership in `owl:Thing`, would mean the object is no longer present in the ontology.

Consider, for example, the action `remove(a)` removing the individual `a`. Consider now a rule with a condition `owl:Thing(x)`.

This condition matches all possible variable substitutions;*i.e.*, all possible substitutions of variables with constant symbols in the signature of the language.

Should this condition match with all individuals, including `a`? For instance, if some rule now removes all statements about some individual `a`:

```
remove(a)
```

the constant symbol a is still in the signature of the language and clearly

$$\mathtt{K} \cup \mathtt{WM} \models \mathtt{owl:Thing(a)}$$

and thus the condition `owl:Thing(x)` should match with the substitution $\theta = \{x \mapsto$ `a`$\}$. However, the current implementation will not match the condition with the substitution θ, because the statement `owl:Thing(a)`, which the `OWL-API` uses to keep track of the signature, is removed from the ontology.

5.3.2.1.2 Solution When removing statements about an individual `a`, the statement `owl:Thing(a)` should not be removed.

5.3.2.2 Inconsistency

5.3.2.2.1 Issue There are actually two issues here regarding inconsistency of the WM as the result of a rule application: (1) type satisfaction may be violated; and (2) some arbitrary logical insonsistency is introduced.

As an example of changing type resulting from an action, consider the ontology `K`:

$$\texttt{Woman}(\texttt{Jane})$$
$$\texttt{Woman} \sqsubseteq \texttt{Person}$$

and consider the rule:

```
if
  Person(x)
then
  remove(Woman(x));
  myPersonFunction(x);
```

where `myPersonFunction` is a user-defined function expecting a `Person` as parameter. Now, `Person(Jane)` is entailed by `K` and thus the condition of the rule matches with substitution $\{x \mapsto$ `a`$\}$. After `Woman(Jane)` is removed from the ontology, `Person(Jane)` is no longer entailed, and thus the argument of `myPersonFunction` is no longer a `Person`.

For an example of logical inconsistency, consider the following ontology `K`:

$$\texttt{Man} \not\sqsubseteq \texttt{Woman}$$
$$\texttt{Man}(\texttt{Jack})$$
$$\texttt{WantsSexChange}(\texttt{Jack})$$

stating that the classes `Man` and `Woman` are disjoint; *i.e.*, there may be no individual that is both a `Man` and a `Woman`, and `Jack` is a man who wants a sex change. Now consider the rule (in informal syntax):

```
if
  Man(x), WantsSexChange(x)
then
  performSexChange(x);
  add(Woman(x));
```

where performSexChange is a user-defined function. When this rule is executed, the statement Woman(Jack) is added to the ontology, making the ontology inconsistent, since Man(Jack) is in the ontology while Man and Woman are disjoint.

5.3.2.2.2 Solution

- Enforce that all user-defined functions in the action part *must* be executed before the working memory is changed.
- The second problem is an update in the working memory causing an inconsistency with respect to the ontology. The considerations here are exactly the same as the considerations in the action part in the theoretical framework (see Section 5.2.2 on Page 117).[4]

5.4 Practical impacts on the rule engine

From a practical point of view, the impact of moving from OM to OWL is related to the ability to build, introspect and reason about descriptions. The OWL reasoning involved here is T-Box reasoning. With the production rule engine the impact is moving on the ability to reason about the content of the working memory. The OWL reasoning now involved is A-Box reasoning.

The following sections will discuss in more details the impact of moving from objects to OWL assertions for:

- the working memory.
- the pattern matching.
- the navigation from a subject individual to related individuals.
- the assertions.

5.4.1 Impact on the working memory

In a PR engine over OM, the working memory is a set of objects managed internally by the rule engine. The objects are added to, removed from the working memory or updated because one of their attributes has changed. The objects are the instances of the classes defined in the OM. Contrariwise with OWL, the working memory is made of the individuals

[4]Note that in this case it may be possible to perform static analysis to discover that the rule is problematic since the concept: Man⊓WantsSexChange⊓Woman is unsatisfiable. For example, if there are only insertions in the action, it is possible simply to conjoin the facts added by the action with the condition and check for satisfiability.

of the OWL ontology who are at least the members of the owl:Thing class. XPR⟨OWL⟩ is still providing means to add, remove and update the individuals themselves. A working memory is still available. This is the object-oriented approach. But XPR⟨OWL⟩ is also providing means to add, remove and in some limited cases update OWL statements about the individuals. The OWL assertions will be discussed in more details below.

While in OM, there is a built-in mecanism to define a custom object identity: an instance comparator that is provided for each class of the OM. In OWL, there is no unique name assumption for the individuals. The reasoner should always be invoked to check whether an individual is identical to another one in particular each time an individual is added to a set. In XPR⟨OWL⟩, only a representative individual is asserted in the working memory and propagated in the Rete network. This representative can evolve as individuals are removed or new identity assertions are added. The working memory and the whole state of the Rete network need to be updated each time a representative is changing. XPR⟨OWL⟩is also providing the set type as a built-in collection type. Each instance of set created from XPR⟨OWL⟩ is using representative individuals.

5.4.2 Impact on pattern matching

The main task of the pattern matching is to check whether a runtime value is of the expected type. If it is not the case, the pattern matching will fail. If it is the case, the pattern matching will succeed and it is safe to consider the value as an instance of the type in the remaining patterns.

With an OM, each time an object is reaching a Rete class node, the OM is invoked to check whether it is an instance of the class.

With OWL, each time an OWL individual is reaching a Rete class node, the OWLreasoner is invoked to check whether it is an instance of an OWL class expression.

5.4.3 Impact on navigation

Navigation is a classical mecanism in PRs that is used within object patterns and when the dot notation is used in a traditional expression.

Classical Rete implementations are based on class conditions. In the body of a class condition, the matched object is becoming the this object. The attribute names that are specified are automatically translated to this.name. But no nested patterns can be introduced for the attribute values. In XPR⟨OWL⟩, the provided OWL individual patterns are similar to class patterns. The matched object is becoming the "subject". But nested patterns can be specified for the “objects” that are connected to the “subject” thanks to a property.

In classical PR over OM, each time a navigation is encountered at runtime, the subject value is known and the OM is asked to give the value of the attribute. But when using

OWL, each time a navigation is encountered at runtime, the subject OWLindividual is know and the OWL reasoner is asked to provide the objects that are related to this subject. Curiously, the OWL-API is returning a java.util.Set of OWL individuals, assuming here that an object property is involved and not a data property. In other words, it is only returning what is materialized in the OWL ontology. One would have expected an OWL implementation to return a more sophisticated kind of set description adapted to the open world assumption that would summarize what is kwnown about the object values. For instance, if an onto:Car is defined to be equivalent to an onto:Vehicle == 4 onto:wheels and only 2 distinct wheels are materialized in the OWL ontology for a Car, it is interesting to know that 2 other distinct wheels are expected to exist somewhere but are not yet known. With OWL-API, what you get is the 2 available wheels. Well, most people who are familiar with the closed world object-oriented approach will think "it's a bike, not a car, this rule about car should not have matched". Here, the distance that exists between logical descriptions and the object-oriented mechanisms should become a bit more clear. It is the same with Jena except that the returned type is a Jena NodeIterator which does not help a lot here when compared to java.util.Set.

For functional properties, a tricky mechanism has been implemented in XPR⟨OWL⟩ to be able to keep the object-oriented navigation, the so called "dot notation" as a production rule language construct. The result here is still a set of values. But this set may be empty in the case where the single expected value is not yet known. This is not really compliant with traditional object-oriented navigation expressions that are not expected to fail on unknown values. In this case the XPR⟨OWL⟩ engine will propagate a local pattern matching failure if the navigation expression is involved in a pattern and throw a global runtime exception if it is involved in an action. This illustrates once again the difference that may exist between the OWL logical statement manipulations and the traditional object-oriented mechanisms.

5.4.4 Understanding production rule semantics for assertions

The action part of the production rules may cause assumptions that were used in the condition part to fail. While in logical rules, this is considered as an inconsistency, in production rules this behaviour is perfectly acceptable. The meaning of a change in the action part is that a new state needs to be considered for the condition parts of all the rules of the ruleset under processing. We must notice here that such changes are manageable when only the production rule engine side is considered. In the past there used to be complex instructions in the condition part of the production rules that matched different states, the so called « *watchdogs* ». Those complex instructions are not used anymore in the modern PR engine and have not been implemented in XPR⟨OWL⟩.

But in the context of XPR⟨OWL⟩ a change is not as manageable on the OWL reasoner side. XPR⟨OWL⟩ is doing its best to avoid introducing inconsistencies in the OWL ontology due to changes. The best XPR⟨OWL⟩ can do to properly manage the changes of the OWL

ontology is first to introspect the OWL ontology and second to use OWL queries to extract the precious derived knowledge from the hands of the OWL reasoner.

All those control steps and round-trips between the production rule engine and the OWL reasoner are obviously extremely costly in time and in transient memory consumption.

5.4.5 Pre-requisites for assertions

The first mandatory pre-requisite is that the OWL manager which is used must accept ontology changes. This is the case for Jena where the concept of mutable OWL ontology is supported.

The second pre-requisite is not mandatory and is related to incrementality. The OWL manager which is used should provide support for incremental changes. In our experiments with OWL-API, we noticed that OWL-API is providing pretty good support for such changes, even if the quality of all this ultimately depends on the OWL reasoner implementation. With OWL-API, a change is defined as an object. Several changes can be considered at once. Listeners can be specified in order to be notified of OWL ontology changes. Jenais providing less sophisticated goodies of this kind. However in both cases, it is not clear at all when an inconsistency due to a bad change is really detected: is it precisely when the change is done or is it at the next reasoning step?

5.4.6 Impact on assertions

In classical PR engines, only the capability to add to, remove from objects of the working memory is provided. An update action is also provided to notify the Rete that some attribute value of an object has changed and that it should be checked again whether the object is still compliant with the patterns it has already matched.

In XPR⟨OWL⟩ this capability is maintained because our working memory is similar to the object-oriented one. The object-oriented "new" instruction of XPR⟨OWL⟩ is able to create new OWL individuals. Those OWL individuals are not automatically added to the working memory. They need to be added to the working memory with an add object action explictly. When they are created, they are nothing else than constant objects. But OWL individuals created from the rules are automatically associated their OWL instantiation class descriptor which is statically available as part of the object-oriented "new" instruction. This OWL class descriptor is used for strong type checking whenever the new OWL individual is used in the rest of the rule. It is also used to automatically convert an add object action involving this individual to an OWL add class assertion.

In addition, the following action are provided in XPR⟨OWL⟩ :

Class assertion updates The OWL add class assertion is available as an XPR⟨OWL⟩ action to assert the fact that an OWL individual is an instance of an OWL class. In OWL,

individuals do not have a single instantiation class as it is the case for objects. Many assertions can be added this way to state all the OWL classes an OWL individual is an instance of. When this assertion is interpreted, the XPR⟨OWL⟩ rule engine will first check whether this direct assertion is not already in the ontology and will only add it to the mutable OWL ontology if it is not the case.

The OWL retract class assertion is available as an XPR⟨OWL⟩ action to retract any similar class assertion already in the OWL ontology. Any attempt to retract the fact that an OWL individual is an owl:Thing is silently discarded. In OWL, an individual is always at least an owl:Thing and can be referenced in many other OWL model elements, OWL class expressions and enumerations in particular. In XPR⟨OWL⟩, an OWL individual can be removed from the working memory but cannot be removed as a whole from the OWL model. Provided the OWL class specified is not owl:Thing, the OWL individual is also automatically removed from the members of all the sub classes of the class to maintain the consistency. The OWLreasoner is used to collect the set of sub classes. Only the direct assertions involving those sub classes that exist in the OWL ontology are removed from the OWL ontology.

Negative OWL class assertions are also supported in XPR⟨OWL⟩ but have not been really used so far.

Property assertion updates The OWL add property assertion is available as an XPR⟨OWL⟩ action to assert that an OWL individual is related to a value thanks to a property. When this assertion is interpreted, the XPR⟨OWL⟩ rule engine will first check whether an equivalent direct assertion is not already in the ontology and will only add it to the OWL ontology if it is not the case. For functional properties, only a unique value should be kept. Any assertion already in the OWL ontology which involves the same subject but a different value is automatically removed.

The OWL retract property assertion is available as an XPR⟨OWL⟩ action to retract any similar property assertion already in the OWL ontology. Assertions already in the OWL ontology that involve the same subject but that are related to sub-properties are automatically removed from the OWL ontology to maintain the consistency. The OWL reasoner is used to collect the set of sub-properties of a property.

The OWL update property assertion is also available as an XPR⟨OWL⟩ action. When this assertion is interpreted, the XPR⟨OWL⟩ rule engine will first check whether an equivalent assertion cannot be derived from the OWL ontology and will only add an assertion for the new object value to the OWL ontology if it not the case. The OWL reasoner is used to perform this check. For functional properties, only a unique value should be kept. Any assertion already in the OWL ontology which involves the same subject but a different value is automatically removed.

Negative OWL property assertions are also supported in XPR⟨OWL⟩ but have not been really used so far.

Individual assertion updates The OWL same individuals assertion is available as an XPR⟨OWL⟩

action to state that two individuals are equivalent. It means that the two OWL individuals will have the same representative.

The OWL different individuals assertion is available as an XPR⟨OWL⟩ action to state that two individuals are distinct. It means that the two OWL individuals will not have the same representative.

Each time an OWL individual is involved in an assertion whatever it is, an automatic update is automatically propagated through the Rete network to check that this OWL individual is still matching the patterns it has already matched.

XPR⟨OWL⟩ is also providing an auto-update feature. Each time an OWL individual is modified, all the OWL individuals which are related to this individual as subject of a property assertion are automatically updated. The inverse of the property is used to collect all the subjects given a particular object. In order to limit the scope of the auto-update feature, only the properties that are really used in the set of rules are considered. The OWL reasoner is used to collect the subjects and to automatically consider the sub-properties.

We must keep in mind here that modifying an OWL ontology with production rules is not really an easy task. The assertion available in an OWL ontology are only the materialized level. The OWL reasoner should systematically be used to also consider all the derived knowledge.

5.4.7 Strong typing versus dynamic classification

The rule checker is expected to implement strong typing. This strong typing is possible in an object-oriented context because an object do not change its class. But with production rules that manage OWL assertion changes as described in the previous section, the OWL individuals may definitely change class. For instance an onto:GoodCustomer can become an onto:BadCustomer even if the two classes have been declared as disjoint. It corresponds to two different states for the same person. Production rules are mainly used to do that kind of changes in a business context.

Let us consider the following XPR⟨OWL⟩ program:

```
rule Good2Bad {
  when {
    foreach x as onto:GoodCustomer;
  }
  then {
    retract x as onto:GoodCustomer;
    insert x as onto:BadCustomer;
    notifyChange(x);
  }
```

```
}

function owl:Nothing notifyChange(onto:GoodCustomer x) {
  // not relevant here
}
```

The variable x is always matching an onto:GoodCustomer. This is compatible with the signature of the utility function « *notifyChange* » which is precisely expecting an argument of this class. The problem is that at runtime, as the action part is executed, the individual x is bound to, has become an onto:BadCustomer thanks to the class assertions just before the function call. With the assumption that onto:GoodCustomer and onto:BadCustomer have been declared as disjoint in the OWL ontology, there is a mismatch between the static type checking and the dynamic classification. This is an odd behaviour. It seems that any PR system tightly connected to OWL should either drop strong static type checking in favor of dynamic type checking or limit the expressiveness of the action language so that this kind of situation cannot occur.

The example above can also be used to show how much easy it is to introduce an inconsistency in an OWL ontology while modifying it:

```
rule Good2Bad {
  when {
    foreach x as onto:GoodCustomer;
  }
  then {
    insert x as onto:BadCustomer;
  }
}
```

Adding the fact that x is an onto:BadCustomer in the action part without removing the fact x is an onto:GoodCustomer before is introducing an inconsistency with the assumption that the two classes are disjoint. This assumption is typically stated as an owl:disjointClasses axiom in the OWLontology. The individual x cannot be an onto:BadCustomer and an onto:GoodCustomer at the same time.

Even if XPR⟨OWL⟩ will do its best to avoid introducing inconsistencies, it is still possible to do this. The reason is that XPR⟨OWL⟩ is not taking into account all the axioms of the OWL ontology. XPR⟨OWL⟩ is implementing some changes in a consistent way, but not all. XPR⟨OWL⟩ is not an OWL reasoner able to manage consistent changes on the derived knowledge. XPR⟨OWL⟩ is simply using an OWL reasoner that has not been designed to accept changes. An OWL reasoner is designed to answer deductive queries on a consistent OWL ontology that does not change. Since OWL components are not ready to deal with changes, XPR⟨OWL⟩ has to manage the changes from its side. But the more XPR⟨OWL⟩ is doing to avoid inconsistencies and the more it digs into derived knowledge, the more it is behaving itself as an OWL reasoner.

This simple remark is illustrating why the next step is to build a new production rule engine which is able to manage changes in an OWL ontology in a consistent way, including the derived knowledge.

5.5 Discussion

This section discusses the work accomplished, does a general recapitulation toward the objectives of XPR⟨OWL⟩. We review the steps taken in the process of combining PRs with OWL: the limitations of our current settings, what the recognized issues are and how to fix them. This is strongly connected to the theoretical part of Deliverable D3.2 [35] but we introduce examples written in the new XPR⟨OWL⟩ syntax to illustrate issues and potential solutions.

In our approach proposing an environment for PRs with the objective of staying perfectly synchronized with an OWL ontology management environment, IBM has kept a rather conservative approach.

Clearly, Rete-based PR systems do not activate rules based on general *logical entailment*. They do so based on clever *pattern matching* of explicit objects populating a working memory. The whole benefit of such systems is the efficiency of the procedure for deciding what objects in the working memory are matched thanks to the Rete Algorithm [21]. The Rete method's benefits rely crucially on the fact that (1) objects are structured records, and (2) all objects reside explicitly in the working memory.

In XPR⟨OWL⟩, the entirety of the logical inference task is delegated to the OWL reasoner. Our experience has shown that, in practice, even though such reasoners do seem to carry out useful bits of inference rather nicely, many times they unfortunately stay rather evasive and inconclusive. This is essentially due to the assumptions made by OWL to live in an open world by default, as opposed to object-oriented or constraint-oriented systems [1] that rely on a closed-world assumption.

In a loosely-coupled design such as XPR⟨OWL⟩'s, because the rule engine's world and the ontology reasoner's world are orthogonal, the system takes care of maintaining a Business Ontology, which may then be indexed efficiently thanks to the closed-world assumption required by the Rete-method. Synchronization problems that arise in the course of this consistency-maintenance between the two worlds are essentially due to the fact that, on one hand, a PR system deals with a world on individual objects defined in *extension* while, on the other hand, OWL reasons over models defined in *intension*.

From a PR's perspective when dealing with rule conditions, the knowledge that is implicit in an OWL knowledge base is not readily available. So the PR system must constantly probe the OWL reasoner for known facts. To this end, there is only one means: sending explicit queries. Even so doing, the only facts that will materialize in the working memory as a result (provided the query actually gets a meaningful reply) are those relevant to answering the query—not all that are *implied* by them. As a consequence, a Rete-style

method only has partial information—it doesn't know as much as could be inferred by the reasoner.

As for the effect of a rule's actions, a PR system must also be careful making modifications in the working memory since they cannot be only extensional (*i.e.*, adding, removing, or updating existing objects) since the OWL reasoner will not take care of keeping its intensions consistent with the working memory's extension. It is therefore up to the PR system to manage such discrepancies. However, the PR system "knows" only partially what the OWL reasoner "knows." Missing in between the two systems is a component that would "patch this mismatch" by maintaining a dual intensional/extensional view of the working memory.

In the current XPR⟨OWL⟩ design, such (incomplete) maintenance is diffused all over the place in the structure of the software. However, it is our observation that a great majority of actual use cases of rules over ontologies encountered in practice, as opposed to fictitious ones designed for academic pursuits, face similar situations. Therefore, designing such a generic interfacing component makes great sense as this would be of great practical benefit.

It must also be recognized that it may not be a good idea to flesh out the explicit extension of all intensional knowledge. Such sets may indeed become rather large. The solution may then be to materialize such extensions by need as they are required for making the Rete-method effective. For example, if a rule set matching only on partial aspects of objects, and not others, it is then sufficient to flesh out only those parts of the extensional representation of the KB relevant to the indexing. If such a dynamic consistency-maintenance interface module were available, the PR system would no longer need to worry about the OWL reasoner—it would only need an OWL Interface Format to communicate with it.

On the other hand, when one sees actual applications of rules over ontologies as done in practice in the Arcelor-Mittal and AUDI use cases [2], such problems often do not arise since a closed-world assumption comes handy for the largest part, if not all, of the KB. This does not mean that these use cases may not evolve to need more than simply using an OWL KB as a data base. It simply means that it would be most useful to try and characterize precisely the Design Pattern proper to what kinds of rules and what kinds of ontologies appear in actual situations.

5.6 Conclusion

The limitations of a combination where a production rule engine delegates reasoning tasks and ontology management to a separate OWL reasoner are now identified. But does it mean that such a weakly coupled combination is not valuable? The answer is no. Many practical use cases are actually based on compliance rules that are not there to modify the ontology but are rather there to react to relevant ontology configurations. In this context, the production rules will not introduce inconsistencies. The XPR⟨OWL⟩ implementation

has been designed for maximum expressivity in this context without too much a focus on inference. The XPR⟨OWL⟩ production rules are likely to introduce inconsistencies when they are used to perform stateful inference on the ontology with actions that modify the ontology while the rules are still in the process of matching this same ontology.

Another discrepency is related to open world vs closed world assumption. The current production rules technlogy is strongly connected to the classical data base technology where all the data is ground and extensively available. A migration from data to logically organized knowledge, in other words from data base to knowledge base, means building a brand new production rule technology. Building a brand new technology from scratch is very risky. It is always better to proceed step by step. The XPR⟨OWL⟩ implementation is the first step. The next step is to combine more tightly production rules and ontology reasoning. The issues listed in this document are clearly pointing out that the production rule engine is not fully aware of what the reasoner has inferred and is not necessarilly properely notified of the inconsistencies. Moreover, the production rules are modifying the ontology, at least the values of the ontology, something logical reasoning hardly appreciates.

A first step might be to improve the communication of the reasoner with the outside world. Typically, a finer grained event based interface w.r.t. queries woud be welcome to be aware of all the intermediate knowledge a reasoner is inferring while answering a query. But reasoners are not likely to provide this kind of service in the timeframe of this project.

Hence, the next step will be to build a brand new production rule technology that is also able to perform reasoning without dropping the main feature of the production rules : the ability to incrementally modify the values of the ontology as the rules are matching it. Some compromise will certainly need to be done, once again to proceed step by step toward a tight combination of production rules and ontologies. Here, closed world assumption and unique name assumption seem to be good candidates. But when taking a look at practical use cases, it does not seem odd to select those two simplifying assumptions. Ultimately, an even tighter combination that fully complies with the standard OWL semantics could be derived.

We have an example here on how much it is difficult to design and implement practical programmatic interfaces to a complex standard like OWL. With the available interfaces it is easy to build OWL ontologies but it is difficult for a software component to use them and in particular to update them once they have been built. OWL is looking very intrusive on the environment it is expected to live in. Production rules is not the only technology which is impacted here, traditional programming languages which are not logic aware are impacted in the same way. But the introduction of OWL is also creating a challenging context to build new technologies.

XPR⟨OWL⟩ has been extremely useful to assess the impact of OWL on the current production rule technology. Not only combinations of OWL with production rules have been tried but also combinations of OWL with all the artefacts that are connected to production rules.

The discussions with FUB have pointed out that the main discrepancy between objects and OWL is that the last is manipulating knowledge made of statements about individuals and is not directly dealing with the individuals as it is the case in an object-oriented approach. A definitive executable production rule suite tightly combined to OWL must provide means to manage OWL assertions at runtime and not directly OWL individuals. It is also mandatory that all the possible derived knowledge is kept consistent each time an OWL assertion is added, removed or updated from the production rule engine. On this particular subject, XPR⟨OWL⟩ has reached the limits of the coupling between an OWL reasoner and a production rule engine. Hopefully this document is explaining why a new single production rule engine able both to manage production rules and to perform OWL reasoning on ever changing knowledge needs to be built and why it cannot be derived from the available production rule technology. This last remark is both annoying and exciting. It will certainly be annoying for customers and sales people who will have to manage a change of tools just because OWL is coming. A very good story will have to be told about the benefits of OWL when compared to OM to convince the ever growing population of cost killers who is ultimately taking decisions. But in the case this story is convincing, this is also exciting for tool builders because they have a rare opportunity to build refreshing new software.

Appendix A

Tree-shaped queries

We give here the description of tree-shaped queries; *i.e.*, queries that can be rewritten to OWL class expressions, so that standard dl reasoners (*e.g.*, Pellet) can be used for evaluating the conditions.

Formulas in XPR⟨OWL⟩ rule conditions are essentially queries. We make a distinction between "*distinguished*" and "*non-distinguished*" variables. Distinguished variables are the free variables in the query; *i.e.*, they do not appear in the scope of a quantifier ($\forall, \exists$). Non-distinguished variables are those variables that are not distinguished; *i.e.*, they appear in the scope of a quantifier.

Below is a description of queries that can be evaluated using standard dl reasoning. We call these queries "*tree-shaped*" because the graph of the non-distinguished variables has a tree shape; *i.e.*, they are rooted, contain no cycles, and all their nodes have at most one immediate ancestor. Tree-shaped queries can be written as dl class expressions, and thus checking tree-shaped conditions can be reduced to checking membership of individuals in dl classes. Checking a conjunction of tree-shaped conditions can be reduced to a number of individual class membership checks, one for each conjunct.

Atomic queries are atomic formulas with unary or binary predicates that do not contain non-distinguished variables.[1] Therefore, they are a special case *tree-shaped* queries. Examples of atomic queries are:

$$\texttt{person}(X), \texttt{person}(\texttt{john}), \texttt{hasChild}(X, Y), \texttt{hasChild}(\texttt{john}, \texttt{mary}).$$

Atomic queries can be answered by checking entailment of class membership or property value assertions. Every atomic query is a tree-shaped query.

Non-atomic tree-shaped queries have a *root variable* or a constant—say, "X." Other tree-shaped queries are Boolean combinations of atomic formulas that use "X" (*e.g.*, $\texttt{person}(X), \texttt{hasChild}(\texttt{john}, X)$) and do not contain non-distinguished variables, with the exception of binary atomic formulas that have a data value as the first argument.

[1] Unary predicates correspond to classes, binary predicates to properties in OWL.

When rewriting tree-shaped queries to class membership assertions, we rewrite:

- conjunction "$A \wedge B$" as "`objectIntersectionOf` $(A\ B)$,"
- disjunction "$A \vee B$" as "`objectUnionOf` $(A\ B)$," and
- negation "$\neg A$" as "`objectComplementOf` (A)."

Class expressions of the form "`class`(X)" are simply written as "`class`," object property expressions of the form "`property`(X, Y)" as "`objectHasValue` (`property` Y)" and object property expressions of the form "`property`(Y, X)" as "`objectHasValue` (`objectInverseOf` (`property`) Y)." Data property expressions (*i.e.*, where the value is a data value rather than a logical constant; *e.g.*, "`hasAge`$(X, 32)$") are denoted using "`dataHasValue` " as in, *e.g.*:

```
dataHasValue (hasAge, "32"^^xs:integer).
```

For example, the query:

$$\texttt{person}(X) \ \wedge \ \texttt{hasChild}(Y, X) \ \vee \ \texttt{car}(X)$$

is written as the assertion:

```
classAssertion (
    objectUnionOf (
        objectIntersectionOf (
            person
            objectHasValue (
                objectInverseOf (hasChild)
                Y
            )
        car
        )
        X
    )
).
```

In place of atomic formulas, one can also use quantified formulas of a specific shape. In particular, the following six forms may be used:

- $\forall Y.\ \big(\texttt{property}(X, Y) \implies \texttt{class}(Y)\big)$;
- $\forall Y.\ \big(\texttt{property}(Y, X) \implies \texttt{class}(Y)\big)$;
- $\exists Y.\ \big(\texttt{property}(X, Y) \wedge \texttt{class}(Y)\big)$;
- $\exists Y.\ \big(\texttt{property}(Y, X) \wedge \texttt{class}(Y)\big)$;
- $\exists Y.\ \big(\texttt{property}(X, Y)\big)$;

- $\exists Y.\ \big(\texttt{property}(Y, X)\big)$.

Here, "$\texttt{class}(Y)$" is a tree-shaped query with "Y" being the root or a datatype.

Here are the rewritings for these six forms, for "$\texttt{class}(Y)$" being a tree-shaped query:[2]

- `objectAllValuesFrom (property class)`;
- `objectAllValuesFrom (objectInverseOf (property) class)`;
- `objectSomeValuesFrom (property class)`;
- `objectSomeValuesFrom (objectInverseOf (property) class)`;
- `objectSomeValuesFrom (property owl:Thing)`;
- `objectSomeValuesFrom (objectInverseOf (property) owl:Thing)`.

Here, "`class`" is the rewriting of "$\texttt{class}(Y)$."

Here is a more involved example of a tree-shaped query and its rewriting:

$$\Big(\texttt{person}(X) \wedge \forall Y.\ \Big(\texttt{hasChild}(Y, X) \implies \texttt{person}(Y) \wedge \texttt{hasAge}(Y, 6)\Big)\Big) \vee \texttt{car}(X)$$

is rewritten as:

```
classAssertion (
    objectUnionOf (
        objectIntersectionOf (
            person
            objectAllValuesFrom (
                objectInverseOf (hasChild)
                objectIntersectionOf (
                    person
                    dataHasValue (
                        hasAge,
                        "6"^^xs:integer
                    )
                )
            )
            car
        )
        X
    )
).
```

[2] If "`class`" is a datatype, we replace "`object...`" with "`data...`."

Appendix B

Analysis of Issues in Use Cases

B.1 Introduction

Task 3.2 is an activity that monitors the issues arising in combinations of rules and ontologies in the case studies, and analyzes and categorizes them according to the survey of Deliverable D3.1 [13]. Additionally, the survey will be updated if necessary in subsequent deliverables in WP3.

B.2 Analysis of the “Steel Industry Use Case"

This analysis is based on the description of the use case in D5.2 *Business Layer of the Steel Domain*, D5.3 *Ontology for the Steel Domain*, and D5.4 *First steel industry internal demonstrators*.

B.2.1 Analysis of Steel Industry Use Case Ontology

We analyze the ontology from D5.3 focused on expressiveness and complexity, taking into account the technologies available in WP3.

The ontology is classified by Protégé 4.1 in the DL fragment $\mathcal{ALCHIF}(\mathbf{D})$. In other words, it extends the basic DL $\mathcal{ALC}$ with role hierarchies, inverse roles, functional roles, and datatypes. In general, checking concept satisfiability as well as checking ABox consistency is EXPTIME-complete. We investigate in the subsequent points whether the actual fragment is less expressive, in particular whether it falls in one of the tractable OWL 2 Profiles or the new Datalog-rewritable DL $\mathcal{DL}^{+}$ introduced in chapter 2.

- No ranges were set for the data type properties, such that a restriction to the OWL 2 EL, OWL 2 QL, and OWL 2 RL data types is from that perspective not a problem.

- Inverse properties are not allowed in OWL 2 EL, they are however present in the ontology, e.g., *location is the inverse of locationOf*. Thus, the ontology does **not** fall in the **OWL 2 EL profile** of OWL.
- The only explicit subclass axioms are of the form $A \sqsubseteq B$ where A and B are concept names. Indeed no expressive class expressions such as union or disjunction are used. The only expressive class expressions occur in *ObjectPropertyDomain* and *ObjectPropertyRange* axioms (which are the only subclass axioms that implicitly occur with more expressive left and right hand sides than just concept names). Both *ObjectPropertyDomain* and *ObjectPropertyRange* axioms are supported by OWL 2 EL, QL, and RL.
- The ontology contains functional roles, which is **not supported by OWL 2 EL and OWL 2 QL**. Functional roles are supported by OWL 2 RL though.
- The ontology contains disjoint classes axioms, functional data properties, and data properties inclusion axioms, which are **not supported by** $\mathcal{LDL}^+$.
- All constructs appearing in the ontology are supported by OWL 2 RL.

In summary, the ontology for the Steel use case is falling in the tractable OWL 2 RL profile.

B.2.2 Analysis of Steel Industry Use Case Rules

This analysis is based on the description of the use case in D5.2 *Business Layer of the Steel Domain*, D5.3 *Ontology for the Steel Domain*, and D5.4 *First steel industry internal demonstrators*.

Rules for the Steel Industry Use Case are described in D5.4 *First steel industry internal demonstrators*.

The rules are described using the Production Rule dialect of RIF (RIF-PRD) and have an equivalent translation in JRules.

The Production Rule dialect was chosen for modeling the rules for the Steel Industry Use Case, as during the modeling process, as mentioned in Deliverable D5.4, the need of rules which allow for existentials in the head appeared. While the tractable approaches studied in the context of Ontorule do now allow for this feature, the formalism studied in the context of tightly-coupled approaches, Forest Logic Programs, adopts the Open World Assumption and allows for unsafe rules, thus offering the possibility of modeling such rules. However this comes at a high computational cost: reasoning with FoLPs takes in the worst case double exponential time. Also, at this stage, FoLPs do not offer support for datatypes and aggregates.

For an in-depth discussion on the usage of these rules, we refer the reader to Section 3.3 of D5.4. In summary, RIF-PRD showed to be expressive enough to capture the intended business meaning of the rules, even though lacking native support for aggregates. Another disadvantage of RIF-PRD is its lack of editing and validation tools.

B.3 Analysis of the Automotive Use Case

This analysis is based on the description of the use case from D4.2 *Semantic integration of BOMs and public demonstrator* [50]. We also consider the extract of the M32 use-case which was provided by Audi for the purpose of the integrated M24 demonstrator [56].

The Automotive M18 Use Case described in D4.2 is concerned with sharing, interchanging, and consolidating Bills of Materials (BOMs) specific to different Computer Aided Methods used in the development of a car. In particular, the Digital MockUp BOM and the AVx BOM have been considered: two ontologies corresponding to each BOM have been generated and then translated to F-Logic. Rules which interlink these two BOMs were also created using F-Logic. The rules corresponding to the BOMs together with the rules for the mapping were stored in a single F-Logic file. This rule repository falls into the stratified fragment of F-Logic. Aggregates and data-types are extensively used. Recursivity is not employed by the rules. However, it might make sense to introduce recursivity for modeling knowledge about the 'parent' of rang k of a part (for now, this is modelled up to depth 3 using non-recurvise rules). Such a modification would not render the KB unstratifiable and it will be considered for the future version of the use case.

The acquisition part of the M24 demonstrator generated an ontology which is used then by the other components of the demonstrator. IBM extended this ontology by adding some concepts and properties contained in the regulation reg16 document, which is a document that explains the seat belt domain including tests and concept descriptions issued by the United Nations Economic Commission for Europe. Both these ontologies fall within every OWL 2 profile and also within the new Datalog-rewritable DL introduced in this deliverable: $\mathcal{LDL}^{+}$. As part of the demonstrator, some Object Logic Rules and XPR-OWL rules have been created. Note that, neither these rules, nor the OWL ontologies available at this stage, capture the full knowledge requirements for the AUDI use case which is due only at M32 in the project. We expect new requirements regarding data modeling to arise during the subsequent development of the ontology and rules. We will monitor these requirements and discuss them in further reports of task T3.2.

Bibliography

[1] Hassan Aït-Kaci. Data models as constraint systems—a key to the semantic web. *Constraint Processing Letters*, 1(1):33–88, November 2007. `http://www.cs.brown.edu/people/pvh/CPL/Papers/v1/hak.pdf`.

[2] Hassan Aït-Kaci, Hugues Citeau, and Roman Korf. Processing of initial combinations of rules and ontologies. Ontorule Project Deliverable D3.5, December 2009. `http://ontorule-project.eu/wiki/InitialCombinationDemos`.

[3] Jürgen Angele, Michael Kifer, and Georg Lausen. Ontologies in f-logic. In *Handbook on Ontologies*. 2009.

[4] The OWL API. Internet Web Page. `http://owlapi.sourceforge.net/`.

[5] F. Baader, S. Brandt, and C. Lutz. Pushing the $\mathcal{EL}$ envelope. In *Proc. IJCAI*, pages 364–369. Morgan-Kaufmann Publishers, 2005.

[6] F. Baader, S. Brandt, and C. Lutz. Pushing the $\mathcal{EL}$ envelope further. In *Proc. OWLED08DC*, 2008. `http://ceur-ws.org/Vol-496`.

[7] F. Baader and U. Sattler. Number restrictions on complex roles in DLs: A preliminary report. In *Proc. KR*, pages 328–339, 1996.

[8] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider (eds). The description logic handbook: Theory, implementation, and applications. In *Description Logic Handbook*. Cambridge University Press, 2003.

[9] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *PODS '86: Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 1–15, New York, NY, USA, 1986. ACM.

[10] C. Baral and V. S. Subrahmanian. Dualities between alternative semantics for logic programming and nonmonotonic reasoning. *JAR*, 10(3):399–420, 1993.

[11] D. Calvanese, G. de Giacomo, D. Lembo, M. Lenzerini, and Riccardo Rosati. Tractable reasoning and efficient query answering in description logics: The DL-Lite family. *JAR*, 39(3):385–429, 2007.

[12] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. In *CCC '97: Proceedings of the 12th Annual IEEE Conference on Computational Complexity*, page 82, Washington, DC, USA, 1997. IEEE Computer Society.

[13] Jos de Bruijn. D3.1 - state-of-the-art survey of issues. Technical report, ONTORULE IST-2009-231875 Project, 2009.

[14] Jos de Bruijn and Martín Rezk. A logic based approach to the static analysis of production systems. In *3rd International Conference on Web Reasoning and Rule Systems (RR 2009)*, pages 254–268, Chantilly, VA, USA, 2009.

[15] T. Eiter, G. Ianni, T. Lukasiewicz, R. Schindlauer, and H. Tompits. Combining answer set programming with description logics for the Semantic Web. *Artificial Intelligence*, 172(12-13):1495–1539, 2008.

[16] T. Eiter, T. Lukasiewicz, R. Schindlauer, and H. Tompits. Well-founded semantics for description logic programs in the Semantic Web. In *Proc. RuleML*, pages 81–97, 2004. Full paper *ACM TOCL*, (to appear).

[17] Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. Effective integration of declarative rules with external evaluations for semantic-web reasoning. In York Sure and John Domingue, editors, *ESWC*, volume 4011 of *Lecture Notes in Computer Science*, pages 273–287. Springer, 2006.

[18] F. Fages. A new fix point semantics for generalized logic programs compared with the wellfounded and the stable model semantics. *New Generation Computing*, 9(4), 1991.

[19] Leonidas Fegaras and David Maier. Optimizing object queries using an effective calculus. *ACM Transactions on Database Systems*, 25(4):457–516, December 2000. `http://lambda.uta.edu/tods00.ps.gz`.

[20] Jena A Semantic Web Framework for Java. Internet Web Page. `http://jena.sourceforge.net/`.

[21] Charles Forgy. Rete: A fast algorithm for the many patterns/many objects match problem. *Artif. Intell.*, 19(1):17–37, 1982.

[22] U. Furbach, H. Günther, and C. Obermaier. A Knowledge Compilation Technique for ALC TBoxes. In *Proc. of the Twenty-Second International Florida Artificial Intelligence Research Society Conference, May 19-21, 2009, Sanibel Island, Florida, USA*, 2009.

[23] Allen Van Gelder, Kenneth Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

[24] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. ICLP*, pages 1070–1080. The MIT Press, 1988.

[25] Giuseppe De Giacomo, Maurizio Lenzerini, Antonella Poggi, and Riccardo Rosati. On the update of description logic ontologies at the instance level. In *AAAI*. AAAI Press, 2006.

[26] Giuseppe De Giacomo, Maurizio Lenzerini, Antonella Poggi, and Riccardo Rosati. On the approximation of instance level update and erasure in description logics. In *AAAI*, pages 403–408. AAAI Press, 2007.

[27] B. N. Grosof, I. Horrocks, R. Volz, and S. Decker. Description logic programs: Combining logic programs with description logic. In *Proc. WWW 2003*, pages 48–57. ACM, 2003.

[28] Y. Guo, Z. Pan, and J. Heflin. An evaluation of knowledge base systems for large owl datasets. Technical report, CSE Department, Lehigh University, 2004. Y. Guo, Z. Pan, and J. Heflin. An Evaluation of Knowledge Base Systems for Large OWL Datasets. Technical Report LU-CSE-04-012, CSE Department, Lehigh University, 2004.

[29] Volker Haarslev and Ralf Möller. Racer system description. In Rajeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *IJCAR*, volume 2083 of *Lecture Notes in Computer Science*, pages 701–706. Springer, 2001.

[30] Harry Halpin and Patrick J. Hayes. When owl:sameAs isn't the same: An analysis of identity links on the semantic web. April 2010.

[31] Ivan Herman. OWLRL - proof of concept implementation. http://www.ivan-herman.net/Misc/2008/owlrl/, 2008.

[32] S. Heymans, D. Van Nieuwenborgh, and D. Vermeir. Conceptual Logic Programs. *Annals of Mathematics and Artificial Intelligence (Special Issue on Answer Set Programming)*, 47(1–2):103–137, 2006.

[33] S. Heymans, D. Van Nieuwenborgh, and D. Vermeir. Open Answer Set Programming for the Semantic Web. *Journal of Applied Logic*, 5(1):144–169, 2007.

[34] S. Heymans, D. Van Nieuwenborgh, and D. Vermeir. Open answer set programming with guarded programs. *Transactions on Computational Logic*, 9(4):1–53, August 2008.

[35] Stijn Heymans, Jos de Bruijn, Martín Rezk, Hassan Aït-Kaci, Hugues Citeau, Roman Korf, Jörg Pührer, Cristina Feier, and Thomas Eiter. D3.2 - Initial combinations of rules and ontologies. Technical report, ONTORULE IST-2009-231875 Project, 2009.

[36] I. Horrocks. Implementation and optimisation techniques. In F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors, *The Description Logic Handbook: Theory, Implementation, and Applications*, pages 306–346. Cambridge University Press, 2003.

[37] I. Horrocks and P. F. Patel-Schneider. Reducing OWL entailment to description logic satisfiability. *Journal of Web Semantics*, 1(4):345–357, 2004.

[38] I. Horrocks, P. F. Patel-Schneider, and Frank van Harmelen. From $\mathcal{SHIQ}$ and RDF to OWL: The making of a Web ontology language. *Journal of Web Semantics*, 1(1):7–26, 2003.

[39] U. Hustadt, B. Motik, and U. Sattler. Reducing $\mathcal{SHIQ}^-$ description logic to disjunctive datalog programs. In *Proc. of KR*, pages 152–162. AAAI Press, 2004.

[40] Eva Maria Kiss, Hugues Citeau, Adil El Ghali, Thomas Krekeler, Roman Korf, Antonia Schwichtenberg, and Jürgen Angele. D3.6 - Efficient processing of expressive combinations. Technical report, ONTORULE IST-2009-231875 Project, 2010.

[41] Hongkai Liu, Carsten Lutz, Maja Milicic, and Frank Wolter. Updating description logic ABoxes. In Patrick Doherty, John Mylopoulos, and Christopher A. Welty, editors, *KR*, pages 46–56. AAAI Press, 2006.

[42] John W. Lloyd. *Foundations of logic programming*. Springer-Verlag New York, Inc., 1987.

[43] B. Motik, B. Cuenca Grau, I. Horrocks, Z. Wu, A. Fokoue, and C. Lutz, editors. *OWL 2 Web Ontology Profiles*. 2008. W3C Rec. 27 Oct. 2009.

[44] B. Motik, P. F. Patel-Schneider, and B. Parsia, editors. *OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax*. 2008. W3C Working Draft April 2009.

[45] B. Motik, R.Shearer, and I. Horrocks. Optimized reasoning in description logics using hypertableaux. In *CADE'07*, volume 4603 of *LNCS*, pages 67–83. Springer, 2007.

[46] Boris Motik. *Reasoning in Description Logics using Resolution and Deductive Databases*. PhD thesis, University of Karlsruhe, Karlsruhe, Germany, January 2006.

[47] OWL 2 web ontology language structural specification and functional-style syntax. Recommendation 27 October 2009, W3C, 2009.

[48] OWLIM. Semantic repository. http://ontotext.com/owlim/index.html, 2008.

[49] R. Rosati. On the decidability and complexity of integrating ontologies and rules. *Journal of Web Semantics*, 3(1):41–60, 2005.

[50] Peter Rosina and Thomas Syldatke. Semantic integration of boms – public demonstrator. Ontorule Project Deliverable D4.2, June 2010. `http://ontorule-project.eu/wiki/M18_CAx_Demonstrator`.

[51] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical OWL-DL reasoner. *J. Web Sem.*, 5(2):51–53, 2007.

[52] M. K. Smith, C. Welty, and D. L. McGuinness, editors. *OWL Web Ontology Language Guide*. 2004. W3C Recommendation 10 February 2004.

[53] Michael Smith, Ian Horrocks, Markus Krötzsch, and Birte Glimm, editors. *OWL 2 Conformance and Test Cases*. 2009. W3C Rec. 27 Oct. 2009.

[54] T. Swift. Deduction in ontologies via ASP. In *Proc. of LPNMR*, pages 275–288, 2004.

[55] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[56] Roman Korf *et al*. Ontorule M24 show-case demonstrator. Technical report, 2011. `http://ontorule-project.eu/wiki/M24_Demonstrators`.

[57] M. Y. Vardi. Reasoning about the Past with Two-way Automata. In *Proc. 25th Int. Colloquium on Automata, Languages and Programming*, pages 628–641. Springer, 1998.

[58] George Eadon Vladimir Kolovski, Zhe Wu. Optimizing enterprise-scale owl 2 rl reasoning in a relational database system. In P.F. Patel-Schneider, Y. Pan, P. Hitzler, P.Mika, L. Zhang, J.Z. Pan, I. Horrocks, and B. Glimm, editors, *Proceedings of the 9th International Semantic Web Conference*, volume 6497 of *Lecture Notes in Computer Science*, pages 436–452. ISWC 2010, Springer-Verlag, Heidelberg, Germany, November 2010.

[59] M. Winslett. Reasoning about action using a possible models approach. In *aaai88*, pages 89–93, 1988.

[60] M. Winslett. *Updating Logical Databases*. Cambridge University Press, 1990.

Glossary

Assertion Box The Assertion Box is the population of assertions. Another way of looking at an assertion is to consider it as a fact. The Assertion box in OWL is restricted to unary and binary facts., 128

Datalog Datalog is a query and rule language for deductive databases that syntactically is a subset of Prolog., 7

Description Logics Description Logics (DLs) are a family of knowledge representation languages. The modeling primitives in most DLs are classes, which represent sets of objects, properties, which are relations between classes, and individuals. Constants may be defined using logical axioms. The language constructs available for writing such axioms depends on the DL at hand. Typical language constructs include class intersection, union, and complement, as well as universal and existential property restrictions., 4, 6, 81, 93

DL-Programs DL-Programs is a loosely coupled approach of integration of Ontology and Rules., 6, 10

DReW DReW (Datalog ReWriter) (http://www.kr.tuwien.ac.at/research/systems/drew/) is a solver which can either be used as a prototype DL reasoner over LDL+ ontologies or as a prototype reasoner for DL-Programs over LDL+ ontologies under well-founded semantics, 6, 28

Fixpoint Logics Fixpoint logics (or fixed point logics) are regarded as logics with a fixpoint operator. Typical examples of fixpoint logics are propositional mu-calculus, which is more expressive then temporal logics, and common knowledge logic, which is an extension of multi-agent logics., 2, 81

Forest Logic Programs (FoLPs) FoLPs is a decidable subset of OASP which has the *forest-model property*., 1, 5, 35

ObjectLogic ObjectLogic is a newly developed ontology language which is based on the development of F-logic F-logic, et al., 1995 and it's new development et al., 2008, F-logic forum. It is developed at ontoprise GmbH., 57

Open Answer Set Programming (OASP) OASP is an extension of (unsafe) function-free Answer Set Programming with open domains: while the syntax is unchanged, and the semantics is still stable-model based, programs are interpreted w.r.t. open domains, i.e., non-empty arbitrary domains which extend the Herbrand universe. OASP is undecidable., 1, 5

Open-World-Assumption In the Open World Assuption one considers that not all facts are known., 82

OWL The Web Ontology Language OWL is an ontology language for the Semantic Web that extends Description Logics. To RDFS it adds features such as class intersection, union and complement, local property restrictions, cardinality restrictions, and reflexive, symmetric, functional, transitive and inverse properties., 114

Rule Engine A Rule Engine is a generic activity which is responsible for applying a set of rules (derivation rules in SBVR and part of the domain specific Terminology box (T-Box) in OWL) to a set of values (called ground facts in SBVR and assertions in OWL)., 129