EU-IST Integrated Project (IP) 2009-231875 ONTORULE

ONTORULE: ONTOlogies meet business RULEs

D3.4 Converged and Optimized Combinations of Rules and Ontologies

Cristina Feier (TUWIEN)

with contributions from: Thomas Eiter (TUWIEN), Michael Kifer (Stony Brook University, FUB), Alessandro Mosca (FUB), Martín Rezk (FUB), Riccardo Rosati (Università La Sapienza, FUB), Magdalena Ortiz Šimkiene (TUWIEN), Mantas Šimkus (TUWIEN), Trung-Kien Tran(TUWIEN), Guohui Xiao (TUWIEN)

Abstract.
The deliverable continues the work on optimized combinations of rules and ontologies, both on the logical programming side and on the production rule side: it presents new results concerning reasoning with Datalog-rewritable ontologies and Forest Logic Programs, and the combination of production rule systems and ontologies. Further on, it addresses issues concerning the integration of both types of combinations: based on logical rules and based on production rules.

Keyword list: Horn-$\mathcal{SHIQ}$, Datalog rewritability, KAOS, OASP, optimal reasoning with FoLPs, Transaction Logic with Partially Defined Actions, GOPS, ACTHEX, $\mathbb{FDNC}$

Work package number and name	WP3 Inference and Execution
Deliverable nature	Report
Dissemination level	Public/PU
Contractual date of delivery	M36
Actual date of delivery	December 20, 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 the theoretical results established during Year 3 of the project concerning combinations of logical rules, ontologies, and production rules.

The first two chapters of the deliverable take further the work described in Deliverable D3.3 [40] concerning combinations of logical rules and ontologies. First, we continued our work on Datalog rewritability by investigating reasoning with a more expressive ontology language, Horn-$\mathcal{SHIQ}$, which allows for both axioms with existentials on the right-hand side and inverse properties. A reasoning algorithm has been devised and a prototype reasoner, KAOS, has been implemented. The first experimental results for KAOS are promising. The work is reported in chapter 2.

Secondly, we improved the previous results concerning reasoning with Forest Logic Programs, by introducing a new worst-case optimal algorithm for reasoning with the fragment. The algorithm uses new termination conditions, in particular, a new redundancy rule and a caching rule. The worst case running time is exponential in the size of the input program, one exponential level lower than before. As reasoning with FoLPs was known to be an EXPTIME-hard problem, it follows that the problem is actually EXPTIME-complete. The work is reported in chapter 3.

Next two chapters present developments on combinations of production rules and ontologies. Chapter 4 introduces a new semantics for combinations of production rules systems and ontologies that includes looping production rules, and can handle inconsistency. A model theoretic semantics is provided, for the case where the ontology part is rule-based, by a sound embedding into *Transaction Logic with partially defined actions* (abbr., $\mathcal{TR}^{PAD}$) [89]. Also, $\mathcal{TR}^{PAD}$ is extended with default negation under a variant of the well-founded semantics [96].

A general framework for combining production rule systems and ontologies, Generalized Ontology-based Production Systems (GOPSs), is provided in Chapter 5 together with a μ-calculus based verification query language. The production rule part of a GOPS is as specified by the RIF-PRD dialect. The decidability of answering verification queries over GOPSs is investigated and a particularization of GOPSs, Lite-GOPSs, is introduced in the second part of the work. Lite-GOPSs employ the light-weight ontology language ($DL-Lite_A$), and the EQL-Lite(UCQ) ontology query language, as the ontology language, and the query verification language, respectively.

Some considerations regarding the integration of all three knowledge representation paradigms: logical rules, production rules, and ontologies, are also provided in chapter 6.

Table of Contents

Chapter 1

Introduction

In this deliverable we present some optimizations and refinements concerning combinations of rules and ontologies, where the languages are rich enough to meet the expressivity required by the case studies, as well as some generalizations of the combinations of production rules and ontologies described in Deliverable D3.3[40]. Finally, we sketch some ideas about how logical rules, production rules, and ontologies, could be combined altogether.

Combining Logical Rules and Ontologies. As part of *Task 3.2 Analysis of Issues in Case Studies*, two main requirements were identified for combinations of rules and ontologies: the possibility to express inverse properties and to create facts pertaining to new individuals (not existent in the universe of discourse). The latter feature is also known as allowing for ”existentials in the head” (in the case of rules) or ”existentials on the right-hand side of the axioms” (in the case of ontologies).

Chapter 2 describes a rewriting based method for answering conjunctive queries over Horn-$\mathcal{SHIQ}$ ontologies[1]. Horn-$\mathcal{SHIQ}$ is an expressive Datalog-rewritable subset of the DL $\mathcal{SHIQ}$ which allows for existentials on the right-hand side of the axioms, as well as for inverse properties, while retaining polynomial data complexity. A prototype reasoner for the fragment has been implemented: the reasoner is called KAOS and it is, to our knowledge, the first system offering conjunctive query answering services over ontologies in Horn-$\mathcal{SHIQ}$, that allows for unknown individuals in the queries. The chapter reports also on some promising experiments performed with KAOS.

Chapter 3 describes an optimized/worst-case optimal reasoning algorithm for Forest Logic Programs (FoLPs). FoLPs are a decidable fragment of Open Answer Set Programming (OASP), an extension of ASP with open domains. The fragment can be used for simulating reasoning with expressive $(SHOQ)$ ontologies, and as such it serves as the underlying fragment of f-hybrid knowledge bases, a tightly-coupled combination of FoLPs themselves and $(SHOQ)$ ontologies. It allows for a type of unsafe rules, feature which

[1]This work has been performed in collaboration with the FWF project “Reasoning in Hybrid Knowledge Bases (FWF P20840)”.

combined with the open domain of interpretation, allows for simulation of existentials in the head or rules. The new algorithm improves on previous algorithms by decreasing the running time one exponential level: this has been made possible by devising a new technique for reducing an infinite forest model to a model of finite bounded size. The algorithm is worst-case optimal and thus, it offers a tight complexity characterization for FoLPs. The knowledge compilation technique introduced during the second year of the project [40] is reused by the new algorithm.

Combining Production Rules and Ontologies. Deliverable D3.3 [40] introduced an operational and model-theoretic semantics to the combination of production rule systems and ontologies. The model-theoretic semantics was given by an embedding of production rule systems into fix-point logic. Chapter 4 describes a formalization of production rule systems based on an embedding of such systems augmented with rule-based DL ontologies in *Transaction Logic with Partially Defined Actions* (abbr., $\mathcal{TR}^{PAD}$) [89]. The combinations considered in this deliverable are significantly more general than the combinations studied last year or other existing formalizations of production rules, like RIF-PRD, in that they support wider ontology integration and cover important extensions that exist in commercial systems such as a *FOR*-loop. Unlike the previous semantics, the semantics for the new combination can also handle inconsistency. Finally, an extension of $\mathcal{TR}^{PAD}$ with default negation under a variant of the well-founded semantics [96] for $\mathcal{TR}^{PAD}$ is provided. Note that this work can also be seen as a way of achieving convergence of approaches based on logical rules, production rules, and ontologies, as it is trivial to embed logic programs in $\mathcal{TR}^{PAD}$.

Chapter 5 defines the class of generalized ontology-based production systems (GOPSs), which formalizes a general and powerful combination of ontologies and production systems together with a verification query language. GOPSs capture and generalize many existing formal notions of production systems. The verification query language is able to express the most relevant formal properties of production systems previously considered in the literature. A general sufficient condition for the decidability of answering verification queries over GOPSs is presented. The second part of the work describes a particularization of GOPSs, Lite-GOPSs, which uses the light-weight ontology language ($DL - Lite_A$), and the EQL-Lite(UCQ) ontology query language, as the ontology language, and the query verification language, resp.. Lite-GOPSs support also a tractable semantics for updates over DL ontologies. Decidability and tractability of several verification tasks over Lite-GOPSs has been investigated.

Convergence. Chapter 6 discusses some issues related to the convergence of approaches based on logical rules, production rules, and ontologies: two logic programming based formalisms are considered as possible integrative devices: ACTHEX, an extension of ASP with external atoms, and action atoms, and $\mathbb{FDNC}$, a decidable subset of ASP with function symbols. Pros and cons for using each approach are given and pointers to future work needed for achieving full convergence.

Finally, Chapter 7 draws some conclusions.

Chapter 2

Reasoning with Horn-$\mathcal{SHIQ}$

2.1 Introduction

Datalog-rewriting is an efficient method for reasoning over Description Logics (DLs). In the previous deliverable D3.3 [40], we developed a novel datalog-rewriting framework, called inline evaluation, for description logic programs (dl-programs), and implemented it in the DReW reasoner.

The ontology language targeted by DReW is $\mathcal{LDL}^+$, which is essentially an extension of OWL 2 RL with nominals, role conjunctions, and transitive closure. $\mathcal{LDL}^+$ enjoys a polynomial complexity, under both settings of data and combined complexity. However, in practice, sometimes $\mathcal{LDL}^+$ is not expressive enough to cover the real-life ontologies. For example, in the well known biomedical ontology Galen [86], they heavily use axioms in the form of (1) $\mathit{LeftEar} \equiv \mathit{Ear} \sqcap (\exists \mathit{hasLeftRightSelector.leftSelection})$, which has existential quantifier in the right hand side, and (2) functional roles like *hasDiameter*. Both of them can not be expressed in $\mathcal{LDL}^+$.

To capture more expressive DLs, in this work, we focus on Horn-$\mathcal{SHIQ}$ [81], which is very expressive, but still has a nice property of PTime completeness in data complexity. We developed a rewriting algorithm for answering conjunctive queries (CQs) over Horn-$\mathcal{SHIQ}$ ontology, in collaboration with FWF project “Reasoning in Hybrid Knowledge Bases (FWF P20840)”. The result is a novel ABox independent rewriting: given a Horn-$\mathcal{SHIQ}$ DL KB $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ and a CQ q, we can eliminate $\mathcal{T}$ from $\mathcal{K}$ by incorporating $\mathcal{T}$ into q. Thus we obtain a union of CQs (UCQs) $\mathcal{P}_{q,\mathcal{T}}$ such that $(\mathcal{T}, \mathcal{A}) \models q(\vec{t})$ iff $\mathcal{A} \models \mathcal{P}_{q,\mathcal{T}}(\vec{t})$. This is of course similar to the DL-Lite approach, but we have a more expressive DL Horn-$\mathcal{SHIQ}$.

We further show that this rewriting can be faithfully used for dl-programs. In the inline evaluation framework, different components of dl-programs are rewritten into some datalog rules in a modular way. The original inline evaluation also applies a polynomial rewriting time restriction to the DL component. If we drop this restriction, then we can inline evaluate dl-programs over Horn-$\mathcal{SHIQ}$ ontologies, by adopting the novel rewriting.

The rest of this chapter is structured as follows: In section 2.2, we recall the Description Logic Horn-$\mathcal{SHIQ}$ and conjunctive query. Section 2.3 presents the query rewriting algorithm for CQ over Horn-$\mathcal{SHIQ}$ ontologies. Section 2.4 presents the implementation of the prototype system KAOS and some experiment results. In section 2.5 we show that our results can be used for the inline evaluation of dl-programs over Horn-$\mathcal{SHIQ}$. We finally summarize in section 2.6.

2.2 Preliminaries

2.2.1 Description Logics Horn-$\mathcal{SHIQ}$ and Horn-$\mathcal{ALCHIQ}_{\sqcap}$

As usual, we assume countably infinite sets $\mathsf{N_C}$ and $\mathsf{N_R}$ of *concept names* and *role names* respectively; we also assume $\{\top, \bot\} \subset \mathsf{N_C}$. A *role* is a role name r, an expression r^-, or an expression $r_1 \sqcap r_2$, where r_1, r_2 are roles. If $r \in \mathsf{N_R}$, then $\mathsf{inv}(r) = r^-$ and $\mathsf{inv}(r^-) = r$. *Concepts* are inductively defined as follows: (a) each $A \in \mathsf{N_C}$ is a concept, and (b) if C, D are concepts and r is a role, then $C \sqcap D$, $C \sqcup D$, $\neg C$, $\forall r.C$, $\exists r.C$, $\geqslant n\, r.C$ and $\leqslant n\, r.C$, for $n \geqslant 1$, are concepts. An expression $C \sqsubseteq D$, where C, D are concepts, is a *general concept inclusion axiom (GCI)*. An expression $r \sqsubseteq s$, where r, s are roles, is a *role inclusion*.

The semantics of KBs is given by *interpretations* $\mathcal{I} = \langle \Delta^{\mathcal{I}}, \cdot^{\mathcal{I}} \rangle$ which map each $A \in \mathsf{N_C}$ to some $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$, and each $r \in \mathsf{N_R}$ to some $r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$, such that $\top^{\mathcal{I}} = \Delta^{\mathcal{I}}$ and $\bot^{\mathcal{I}} = \emptyset$. The interpretation of role conjunction $(r_1 \sqcap r_2)^{\mathcal{I}} = r_1^{\mathcal{I}} \cap r_2^{\mathcal{I}}$. The map $\cdot^{\mathcal{I}}$ is extended to all concepts and remaining roles as usual. We say that $\mathcal{I}$ is a *model* of $\mathcal{T}$, in symbols $\mathcal{I} \models \mathcal{T}$, if it satisfies all axioms in the TBox.

Then Horn-$\mathcal{ALCHIQ}_{\sqcap}$ is simply defined by restricting to the following axiom schema:

Definition 2.2.1. *Horn-$\mathcal{ALCHIQ}_{\sqcap}$ TBoxes contain only GCIs of the forms*

$$\begin{array}{rll} A \sqcap B \sqsubseteq C & A \sqsubseteq \forall r.B & A \sqsubseteq \geqslant m\, r.B \\ \exists r.A \sqsubseteq B & A \sqsubseteq \exists r.B & A \sqsubseteq \leqslant 1\, r.B \end{array}$$

where A, B, C are concept names, r is a role, S is a role conjunction and $m \geqslant 1$.

For a role conjunction $S = r_1 \sqcap \ldots \sqcap r_n$, we denotes $r_1^- \sqcap \ldots \sqcap r_n^-$ by S^- for simplicity. For any role inclusion $S \sqsubseteq r \in \mathcal{T}$, we also have $S^- \sqsubseteq r^- \in \mathcal{T}$.

The normal Horn-$\mathcal{ALCHIQ}$ is obtained from Horn-$\mathcal{ALCHIQ}_{\sqcap}$ by disallowing role conjunction ($\sqcap$). If we additionally allow transitive roles in normal Horn-$\mathcal{ALCHIQ}$, we get the normal Horn-$\mathcal{SHIQ}$.

In principle, Horn-$\mathcal{SHIQ}$ can be reduced to normal Horn-$\mathcal{ALCHIQ}$ while preserving satisfiability and answers of conjunctive queries [64]. To simplify presentation, we use role conjunctions; therefore, we focus on the DL Horn-$\mathcal{ALCHIQ}_\sqcap$.

Horn-$\mathcal{SHIQ}$ is a very expressive DL. Roughly speaking, Horn-$\mathcal{SHIQ}$ is a super set of all the OWL 2 lightweight fragments[1], depicted in Figure 2.1.

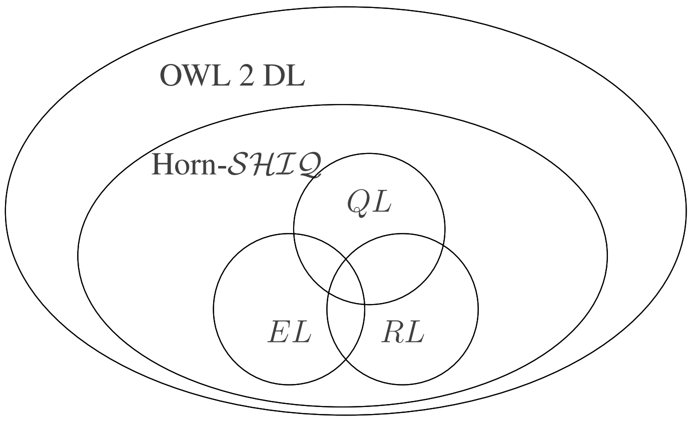


Figure 2.1: Horn-$\mathcal{SHIQ}$ and OWL 2 profiles

2.2.2 Conjunctive Query Answering in Description Logics

Conjunctive queries have been studied extensively in database theory. They are an important class of queries, which corresponds to SQL/algebra Select-Project-Join queries.

Definition 2.2.2 ([80] Query atom, Conjunctive query)**.** *Let N_V denote a countably infinite set of variables, and let $\mathcal{L}$ be a DL, v and v' either a variable from N_V or an individual from N_I , C a concept in $\mathcal{L}$, and R a role in $\mathcal{L}$.*

A query atom *in $\mathcal{L}$ is an expression of the form $C(v)$ (a concept atom) or $R(v, v')$ (a role atom).*

A conjunctive query (CQ) *in $\mathcal{L}$ is of the form $\exists\vec{v}.conj(\vec{x}, \vec{v})$, where $conj(\vec{x}, \vec{v})$ is a conjunction of query atoms. $\vec{x}$ are called* distinguished variables *or* answer variables. *Other variables are called* non-distinguished variables. *If $\vec{x} = x_1, \dots, x_n$ then we call this a query with n answer variables.*

A union of conjunctive queries is a list of conjunctive queries which have the same answer variables.

Definition 2.2.3 (Union of conjunctive queries (UCQ))**.** *UCQ is of the form*

$$\bigvee_{i=1,\dots,n} \exists\vec{v_i}.conj(\vec{x}, \vec{v_i})$$

[1]In this figure, for simplicity, we did not consider enumerations (nominals) involving a single individual (ObjectOneOf) in OWL 2 EL. Singleton nominals can be handled in Horn-$\mathcal{SHOIQ}$ [81], a slight extension of Horn-$\mathcal{SHIQ}$. Datatypes are not discussed either, as they can be similarly treated as singleton nominals.

where each $\exists\vec{v}.conj(\vec{x},\vec{v})$ is a conjunctive query.

Notation 2.2.4. *We often use Datalog notation for conjunctive queries.*

- DATALOG *notation of the conjunctive query $\exists\vec{v}.conj(\vec{x},\vec{v})$ as follows:*

$$q(\vec{x}) \leftarrow A_1(\vec{v_1}), \ldots, A_n(\vec{v_n}).$$

where $A_i(\vec{v_i})$ are query atoms; $\vec{x}$ and $\vec{v}$ are individuals or variables; q is not a description logic predicate and can has arbitrary name.

- DATALOG *notation of union of conjunctive queries $\bigvee_{i=1,\ldots,n} \exists\vec{v_i}.conj(\vec{x},\vec{v_i})$*

$$\begin{array}{rcl} q(\vec{x}) & \leftarrow & A_1(\vec{v_{1_1}}), \ldots, A_m(\vec{v_{1_m}}). \\ & \vdots & \\ q(\vec{x}) & \leftarrow & A_1(\vec{v_{n_1}}), \ldots, A_m(\vec{v_{n_m}}). \end{array}$$

Definition 2.2.5 (Boolean conjunctive queries)**.** *The conjunctive queries that do not have answer variables are called* boolean queries.

Example 2.2.6. *We consider the following queries:*

$$\begin{array}{rcl} q & \leftarrow & Professor(x), Student(y), hasFather(y,x). \\ sameDepartment(x,y) & \leftarrow & Professor(x), Professor(y), Department(z), \\ & & worksFor(x,z), worksFor(y,z). \end{array}$$

The first query is a boolean query because it does not have any answer variable. The second one is not a boolean query; it has two answer variables x and y.

Notation 2.2.7. *For a query $q = \exists\vec{v}.conj(\vec{x},\vec{v})$, we denote:*

- *Atoms(q) as the set of atoms occurring in q;*

- *VI(q) as the set of individuals and variables occurring in q;*

- *N_C(q), N_R(q), $N_I(q)$ are set of concept names, role names, and individual names, respectively in q.*

To give the meaning to a query, each variable and individual in this query must be mapped/bounded to elements of an interpretation domain.

Definition 2.2.8 (Binding, Query match)**.** *Let $\mathcal{I}$ an interpretation.*

A binding *for a query $q(\vec{x})$ in $\mathcal{I}$ is a total function π : VI$(q) \rightarrow \Delta^{\mathcal{I}}$ such that $\pi(d) = d^{\mathcal{I}}$ for each individual $d \in$ VI(q).*

We write $\mathcal{I}, \pi \models C(v)$ if $\pi(v) \in C^{\mathcal{I}}$; and $\mathcal{I}, \pi \models R(v, v')$ if $(\pi(v), \pi(v)) \in R^{\mathcal{I}}$.

A match *for $q(\vec{x})$ in $\mathcal{I}$ is a binding π such that $\mathcal{I}, \pi \models A(\vec{v_i})$ for all atoms $A(\vec{v_i})$ in $q(\vec{x})$.*

Intuitively, a match for a query in an interpretation $\mathcal{I}$ is a mapping from variables and individuals of this query to elements in the interpretation domain that makes this query true under $\mathcal{I}$.

Definition 2.2.9 (Query answer)**.** *The answer variables can only be mapped to individuals in the ABox.*

- *A tuple of domain elements $\langle d_1, \ldots, d_n \rangle$ is called an* answer *for query $q(x_1, \ldots, x_n)$ in an interpretation $\mathcal{I}$ if there is a match π for q in $\mathcal{I}$ such that: $\pi(x_i) = d_i$) for every i.*

- *A tuple of individuals $\langle a_1, \ldots, a_n \rangle$ is called an* answer *for query $q(x_1, \ldots, x_n)$ over the knowledge base $\mathcal{K}$ if it is an answer in all model $\mathcal{I}$ of $\mathcal{K}$.*

It is possible to have complex concepts in query atoms, but it does not make a query more expressive. We can reduce these complex concepts into atoms of the form $C(v)$ or $P(v, v')$, where C is a concept name and r is a role name. We call the resulting query *extensionally reduced query*. We consider the following example of query match and query answer.

Example 2.2.10. *Let $\mathcal{I} = \langle \Delta^{\mathcal{I}}, \cdot^{\mathcal{I}} \rangle$ an interpretation as follows*

$$\Delta^{\mathcal{I}} = \{p_1, p_2, p_3, p_4, d_1, d_2\}$$

$$\begin{aligned} \mathit{Professor}^{\mathcal{I}} &= \{p_1, p_2, p_3, p_4\} \quad \mathit{Department}^{\mathcal{I}} = \{d_1, d_2\} \\ \mathit{worksFor}^{\mathcal{I}} &= \{(p_1, d_1), (p_2, d_1), (p_3, d_2), (p_4, d_2)\} \end{aligned}$$

We consider a conjunctive query:

$$\begin{aligned} \mathit{sameDepartment}(x, y) \leftarrow\ & \mathit{Professor}(x), \mathit{Professor}(y), \mathit{Department}(z), \\ & worksFor(x, z), worksFor(y, z). \end{aligned}$$

The binding π such that $\pi(x) = p_1, \pi(y) = p_2, \pi(z) = d_1$ is a match for the above query in $\mathcal{I}$, and $\langle p_1, p_2 \rangle$ is an answer in the model $\mathcal{I}$.

Definition 2.2.11 (Query answering)**.** *Let $\mathcal{K}$ a knowledge base.* Query answering *is to compute all answers for given query q in $\mathcal{K}$.*

Example 2.2.12. *Let $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$ a knowledge base with:*

- *TBox $\mathcal{T}$:*

$$\begin{aligned} Professor &\sqsubseteq \mathit{Employee} \\ Professor &\sqsubseteq \exists worksFor.\mathit{Department} \\ WorkingGroup &\sqsubseteq \mathit{Department} \end{aligned}$$

- *ABox $\mathcal{A}$*

$$\begin{array}{lll} Professor(p_1) & Professor(p_2) & Employee(e_1) \\ Employee(e_2) & CSDepartment(d_1) & Department(d_2) \\ worksFor(p_1, d_1) & worksFor(p_2, d_1) & worksFor(e_1, d_2) \\ worksFor(e_2, d_2) & & \end{array}$$

- *Query q*

$$\begin{aligned} sameDepartment(x, y) \leftarrow\ & Professor(x), Professor(y), Department(z), \\ & worksFor(x, z), worksFor(y, z). \end{aligned}$$

We consider two models:

- *$\mathcal{I} = \langle \Delta^\mathcal{I}, \cdot^\mathcal{I} \rangle$ maps $a^\mathcal{I}$ to $a \in \Delta^\mathcal{I}$ and:*

$$\Delta^\mathcal{I} = \{p_1, p_2, d_1, d_2, e_1, e_2\}$$

$$\begin{array}{rclrcl} Professor^\mathcal{I} & = & \{p_1, p_2, e_1, e_2\} & Employee^\mathcal{I} & = & \{p_1, p_2, e_1, e_2\} \\ WorkingGroup^\mathcal{I} & = & \{d_1\} & Department^\mathcal{I} & = & \{d_1, d_2\} \end{array}$$

$$worksFor^\mathcal{I} = \{(p_1, d_1), (p_1, d_1), (e_1, d_2), (e_2, d_2)\}$$

- *$\mathcal{J} = \langle \Delta^\mathcal{J}, \cdot^\mathcal{J} \rangle$ maps $a^\mathcal{J}$ to $a \in \Delta^\mathcal{J}$ and:*

$$\Delta^\mathcal{J} = \{p_1, p_2, d_1, d_2, e_1, e_2\}$$

$$\begin{array}{rclrcl} Professor^\mathcal{J} & = & \{p_1, p_2\} & Employee^\mathcal{J} & = & \{p_1, p_2, e_1, e_2\} \\ WorkingGroup^\mathcal{I} & = & \{d_1\} & Department^\mathcal{I} & = & \{d_1, d_2\} \end{array}$$

$$worksFor^\mathcal{J} = \{(p_1, d_1), (p_1, d_1), (e_1, d_2), (e_2, d_2)\}$$

It is easy to check that (p_1, p_2) and (e_1, e_2) are answers for query q in $\mathcal{I}$. However, only (p_1, p_2) is a answer for q over $\mathcal{K}$, because every model $\mathcal{K}$ must satisfy $\mathcal{A}$, thus there always exists a query match $\pi : x \mapsto p_1; y \mapsto p_2; z \mapsto d_1$. (e_1, e_2) is not a answer for q over $\mathcal{K}$, since there is another model $\mathcal{J}$ of $\mathcal{K}$ such that $\mathcal{K}, \mathcal{J} \not\models q$.

Definition 2.2.13 (Query entailment)**.** *Given a Boolean query q and a knowledge base $\mathcal{K}$, we say that $\mathcal{K}$ entails q, in symbols $\mathcal{K} \models q$, if $\mathcal{I} \models q$ for every model $\mathcal{I}$ of $\mathcal{K}$. The* query entailment problem *is to decide , given a knowledge base $\mathcal{K}$ and a Boolean query q, whether $\mathcal{K} \models q$.*

We consider again the knowledge base $\mathcal{K}$ in Example 2.2.12. The difference is that the query q does not contain answer variables.

Example 2.2.14.

$$\begin{aligned} q \leftarrow\ & Professor(x), Professor(y), WorkingGroup(z), \\ & worksFor(x, z), worksFor(y, z). \end{aligned}$$

This query asks whether there exists two professors working in the same working group. We can claim that $\mathcal{K} \models q$ by the explanation as in Example 2.2.12.

2.3 Conjunctive Query Answering in Horn-$\mathcal{SHIQ}$

We present a rewriting-based method for answering conjunctive queries over Horn-$\mathcal{SHIQ}$ ontology [2]. In principle, Horn-$\mathcal{SHIQ}$ can be reduced to normal Horn-$\mathcal{ALCHIQ}$ while preserving satisfiability and answers of conjunctive queries. To simplify presentation, we use role conjunctions; therefore, we work on Horn-$\mathcal{ALCHIQ}_{\sqcap}$. Our approach is similar to the combined-approach in [61], but instead of completely building the compact canonical model, we apply rules to simulate the extended ABox and derive all existential restrictions needed for query rewriting. This approach contains two main steps:

(1) We apply a specially tailored resolution calculus on an input TBox $\mathcal{T}$ to obtain a saturated set of axioms $\mathsf{sat}(\mathcal{T})$. This not only provides consequences of $\mathcal{T}$, but also enables construction of canonical models. For any ABox $\mathcal{A}$, if $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ is consistent, we can build from $\mathcal{A}$ and $\mathsf{sat}(\mathcal{T})$ an interpretation $\mathcal{I}_{\mathcal{K}}$ such that $\mathcal{I}_{\mathcal{K}}$ is a model of $\mathcal{K}$, and $\mathcal{I}_{\mathcal{K}}$ can be homomorphically embedded in any other model of $\mathcal{K}$. The calculus is a nontrivial modification of the calculus in [59] and constructions in [81] .

(2) Each (possibly infinite) model $\mathcal{I}_{\mathcal{K}}$ can be seen as a forest where roots are constants and the remaining tree nodes are anonymous elements implied by existential axioms. Assume $\mathcal{I}_{\mathcal{K}} \downarrow$ is the restriction of $\mathcal{I}_{\mathcal{K}}$ to the interpretation of constants, i.e. the "graph-part" of $\mathcal{I}_{\mathcal{K}}$. Assume also an arbitrary conjunctive query q. We show that using $\mathsf{sat}(\mathcal{T})$ we can rewrite q into a UCQ $\mathsf{rew}_{\mathcal{T}}(q)$ in such a way that the answer to q over $\mathcal{I}_{\mathcal{K}}$ equals the answer to $\mathsf{rew}_{\mathcal{T}}(q)$ over $\mathcal{I}_{\mathcal{K}} \downarrow$. We also show that $\mathcal{I}_{\mathcal{K}} \downarrow$ can be generated by a plain DATALOG program. Thus we reduce the problem of answering a query q over a KB $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ to evaluating a DATALOG query $\mathcal{P}_{q,\mathcal{T}}$ over a database instance (i.e. an ABox $\mathcal{A}$).

2.3.1 Saturation

We recall that before doing reasoning in a Horn-$\mathcal{SHIQ}$ KB, we normalize it. Thus, from now on we only consider KBs in the normal form of Horn-$\mathcal{ALCHIQ}_{\sqcap}$.

Definition 2.3.1. *The TBox of the normal form of a Horn-$\mathcal{ALCHIQ}_{\sqcap}$ contains only axioms of the forms*

$$A \sqcap B \sqsubseteq C \quad A \sqsubseteq \exists r.B \quad A \sqsubseteq \forall r.B \quad A \sqsubseteq \leqslant 1 r.B \quad S \sqsubseteq r$$

where A, B, C are concept names; r, s are role names; S is the conjunction of role names. The semantics of role conjunction $r_1 \sqcap \ldots \sqcap r_n$ is defined as follows: $(r_1 \sqcap \ldots \sqcap r_n)^{\mathcal{I}} = r_1^{\mathcal{I}} \cap \ldots \cap r_n^{\mathcal{I}}$, with $n > 0$, for all roles $r_1 \ldots r_n$, and every interpretation $\mathcal{I}$.

[2]The query rewriting algorithm here is done in collaboration with FWF project "Reasoning in Hybrid Knowledge Bases (FWF P20840)" and mostly done in the FWF project. The results have been published in the master thesis of Trung-Kien Tran [95] with detailed proofs.

$$\frac{M \sqsubseteq \exists(S \sqcap S').N \quad S \sqsubseteq r}{M \sqsubseteq \exists(S \sqcap S' \sqcap r).N}\ (\sqsubseteq_1) \qquad \frac{M \sqsubseteq \exists S.N \sqcap N' \quad N \sqsubseteq A}{M \sqsubseteq \exists S.N \sqcap N' \sqcap A}\ (\sqsubseteq_2)$$

$$\frac{M \sqsubseteq \exists S.N \sqcap \bot}{M \sqsubseteq \bot}\ (\bot) \qquad \frac{M \sqsubseteq \exists S.N \sqcap N' \quad N' \sqsubseteq \exists S'.N''}{N \sqcap N' \sqsubseteq \exists S'.N''}\ (\exists_1)$$

$$\frac{M \sqcap A \sqsubseteq \exists S \sqcap r.N \quad A \sqsubseteq \forall r.B}{M \sqcap A \sqsubseteq \exists S \sqcap r.N \sqcap B}\ (\forall_1) \qquad \frac{M \sqsubseteq \exists(S \sqcap r^-).N \sqcap A \quad A \sqsubseteq \forall r.B}{M \sqsubseteq B}\ (\forall_2)$$

$$\frac{M \sqsubseteq \exists S \sqcap r.N \quad A \sqsubseteq \forall r.B \quad \{M, A\} \subseteq \Sigma_{\mathcal{A}}}{M \sqcap A \sqsubseteq \exists S \sqcap r.N \sqcap B}\ (\forall_3)$$

$$\frac{\begin{array}{l} M \sqsubseteq \exists(S \sqcap r).N \sqcap B \quad A \sqsubseteq\ \leqslant 1\, r.B \\ M' \sqsubseteq \exists(S' \sqcap r).N' \sqcap B \end{array}}{M \sqcap M' \sqcap A \sqsubseteq \exists(S \sqcap S' \sqcap r).N \sqcap N'}\ (\leqslant_1)$$

$$\frac{\begin{array}{l} M \sqsubseteq \exists(S \sqcap r^-).N_1 \sqcap N_2 \sqcap A \quad A \sqsubseteq\ \leqslant 1\, r.B \\ N_1 \sqcap A \sqsubseteq \exists(S' \sqcap r).N' \sqcap B \sqcap C \end{array}}{M \sqcap B \sqsubseteq C \quad M \sqcap B \sqsubseteq \exists(S \sqcap (S')^-).N_1 \sqcap N_2 \sqcap A}\ (\leqslant_2)$$

Table 2.1: Saturation rules

The ABox axioms in normal Horn-$\mathcal{ALCHIQ}_\sqcap$ are in the form of $A(a)$, where $A \in \Sigma_{\mathcal{A}}$, and $\Sigma_{\mathcal{A}}$ is a set of concept names possibly occur in Abox $\mathcal{A}$. For efficiency reason, we do not use all the concepts in $\mathcal{K}$.

We saturate $\mathcal{T}$ using rules triggered by axioms that appear in a Horn-$\mathcal{ALCHIQ}_\sqcap$ TBox. To simplify notation, we denote by M, N, M', N' (resp., S, S') the conjunctions of atomic concepts (resp., roles) For a role conjunction $S = r_1 \sqcap \ldots \sqcap r_n$, we denotes $r_1^- \sqcap \ldots \sqcap r_n^-$ by S^- for simplicity. We write $M \subseteq M'$ if the concept names of M also occur in M'. Similarly, we write $S \subseteq S'$ if the roles of S also occur in S'. We similarly define $A \in M$ and $r \in S$ in the obvious way.

Definition 2.3.2 (Saturation). *Let* $\mathsf{sat}(\mathcal{T})$ *denote the TBox obtained from a TBox* $\mathcal{T}$ *by exhaustively applying the inference rules in Table 2.1.*

Lemma 2.3.3. *The set of axioms* $\mathsf{sat}(\mathcal{T})$ *obtained from the saturation on a Horn-*$\mathcal{ALCHIQ}_\sqcap$ *TBox* $\mathcal{T}$ *are logical consequences of* $\mathcal{T}$.

We explain the saturation rules in the following two examples.

Example 2.3.4. *TBox* $\mathcal{T} = \{A \sqsubseteq \exists R.B \sqcap C; B \sqsubseteq \forall S.D; C \sqsubseteq \exists S.D'; D \sqcap D' \sqsubseteq \bot;\}$.

We assume that $\{B, C\} \notin \Sigma_{\mathcal{A}}$, so rule $\forall_3$ is not applicable.

$$\begin{array}{rcll}
C & \sqsubseteq & \exists S.D' & \\
 & \Downarrow & & \textit{by rule } (\exists_1) \\
B \sqcap C & \sqsubseteq & \exists S.D' & \\
 & \Downarrow & & \textit{by rule } (\forall_1) \\
B \sqcap C & \sqsubseteq & \exists S.D \sqcap D' & \\
 & \Downarrow & & \textit{by rule } (\sqsubseteq_2) \\
B \sqcap C & \sqsubseteq & D \sqcap D' \sqcap \bot & \\
 & \Downarrow & & \textit{by rule } (\bot) \\
B \sqcap C & \sqsubseteq & \bot & \\
 & \Downarrow & & \textit{by rule } (\sqsubseteq_2) \\
A & \sqsubseteq & \exists R.B \sqcap C \sqcap \bot & \\
 & \Downarrow & & \textit{by rule } (\bot) \\
A & \sqsubseteq & \bot &
\end{array}$$

Without ($\exists_1$) we never get $A \sqsubseteq \bot$, and $B \sqcap C \sqsubseteq \bot$, which should be derivable.

Example 2.3.5. *Let $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$, where $\mathcal{T} = \{C \sqsubseteq \exists r.D;\ A \sqsubseteq \forall r.B;\ D \sqcap B \sqsubseteq \bot\}$, $\mathcal{A} = \{C(a), A(a)\}$.*

$$\begin{array}{rcll}
C & \sqsubseteq & \exists r.D & \\
 & \Downarrow & & \textit{by rule } \forall_3 \\
C \sqcap A & \sqsubseteq & \exists r.D \sqcap B & \\
 & \Downarrow & & \textit{by rule } \exists_2 \\
C \sqcap A & \sqsubseteq & \exists r.D \sqcap B \sqcap \bot & \\
 & \Downarrow & & \textit{by rule } \bot \\
C \sqcap A & \sqsubseteq & \bot &
\end{array}$$

However this contradicts to assertions in ABox $\mathcal{A}$, thus $\mathcal{K}$ is unsatisfiable. We should remind that if $\mathcal{K}$ contains only $\mathcal{T}$, then $\mathcal{K}$ is satisfiable. Hence the rule $\forall_3$ does take into account ABox. We can compare the result of this process with the result of Tableau-based algorithm in Figure 2.2

Figure 2.2: Model building of $\mathcal{K}$

$$\begin{tikzcd}
\textcircled{a}\ \{C, A, \exists r.D\} \arrow[r, Rightarrow] & \textcircled{a}\ \{C, A, \exists r.D\} \arrow[d, "r"'] \arrow[r, Rightarrow] & \textcircled{a}\ \{C, A, \exists r.D\} \arrow[d, "r"'] \\
 & \textcircled{b}\ \{D\} & \textcircled{b}\ \{D, B, (Clash)\}
\end{tikzcd}$$

The main idea of saturation is to derive all relevant axioms necessary for query rewriting part. Next, we will present the completion rules for ABox and show that using these rules and axioms from saturation we can build a canonical model of $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$.

2.3.2 Canonical Models

A canonical model for a consistent KB $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ is the model that can be homomorphically embedded into any model of $\mathcal{K}$. In other words, it is sufficient to answer queries based on such model. A canonical model can be seen as two parts: a graph part and a tree part. The graph part is constructed based on the following completion rules.

Definition 2.3.6 (Completion rules)**.** *Let $r[x, y] = r(x, y)$ if $r \in \mathsf{N_R}$, and $r[x, y] = s(y, x)$ if $r = s^-$ of some $s \in \mathsf{N_R}$. Let $\mathsf{cr}(\mathcal{T})$ be the* DATALOG *program consisting of the following rules:*

(a) $B(y) \leftarrow A(x), r[x, y]$ *for all* $A \sqsubseteq \forall r.B \in \mathcal{T}$*;*

(b) $B(x) \leftarrow A_1(x), \ldots, A_n(x)$ *for all* $A_1 \sqcap \ldots \sqcap A_n \sqsubseteq B \in \mathsf{sat}(\mathcal{T})$*);*

(c) $\bot(x) \leftarrow A(x), r[x, y_1], r[x, y_2], B(y_1), B(y_2), y_1 \neq y_2$ *for all* $A \sqsubseteq\ \leqslant 1\, r.B \in \mathcal{T}$*;*

(d) $r[x, y] \leftarrow r_1[x, y], \ldots, r_n[x, y]$ *for all* $r_1 \sqcap \ldots \sqcap r_n \sqsubseteq r \in \mathcal{T}$*;*

(e) for all $A \sqsubseteq\ \leqslant 1\, r.B \in \mathcal{T}$ *and all axioms* $A_1 \sqcap \ldots \sqcap A_n \sqsubseteq \exists r_1 \sqcap \ldots \sqcap r_m.B_1 \sqcap \ldots \sqcap B_k$ *of* $\mathsf{sat}(\mathcal{T})$ *such that* $r = r_i$ *and* $B = B_j$ *for some* i, j*, add rules:*

$$B_1(y) \wedge \ldots \wedge B_k(y) \wedge r_1[x, y] \wedge \ldots \wedge r_k[x, y] \leftarrow A(x), A_1(x), \ldots, A_n(x), r[x, y], B(y).$$

A model of $\mathcal{A} \cup \mathsf{cr}(\mathcal{T})$ is "almost" a model of $\mathcal{T}$: existential axioms may be violated. To deal with this, we extend such a model with new domain elements, following the existential axioms $A \sqsubseteq \exists r.N$ in $\mathsf{sat}(\mathcal{T})$. The domain of canonical model contains elements as words of the form $w = a \cdot (\exists S_1.N_1) \cdot \cdots \cdot (\exists S_n.N_n)$, where $a \in \mathsf{N_I}$. We define the *type* of an element w as follows:

Definition 2.3.7. *For an interpretation $\mathcal{I}$ and individual w, we define the* type *of w w.r.t. $\mathcal{I}$ as $type(w) = \{A \mid w \in A^{\mathcal{I}}\}$. For a word $w = w'.\exists S.N$, we define $type(w) = N$.*

Definition 2.3.8. *Let $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ a KB such that the program $\mathcal{A} \cup \mathsf{cr}(\mathcal{T})$ is consistent and assume $\mathcal{I}_{\mathcal{A}}$ is the least model of $\mathcal{A} \cup \mathsf{cr}(\mathcal{T})$. We define $\mathsf{paths}(\mathcal{K})$ as the smallest set of words of the form $a \cdot (\exists S_1.N_1) \cdot \cdots \cdot (\exists S_n.N_n)$ such that:*

(P1) $a \in \mathsf{paths}(\mathcal{K})$ *for all individuals* a *in* $\mathcal{A}$*;*

(P2) if $a \in \mathsf{paths}(\mathcal{K})$ *and there is an axiom* $M \sqsubseteq \exists S.N \in \mathsf{sat}(\mathcal{T})$ *that satisfies*

- *(a)* $M \subseteq type(a)$*,*
- *(b) there is no* $b \in \mathsf{N_I}$ *such that* $(a, b) \in \bigcap_{r \in S} r^{\mathcal{I}_{\mathcal{A}}}$ *and* $b \in \bigcap_{A \in N} A^{\mathcal{I}_{\mathcal{A}}}$*, and*
- *(c) there is no* $M' \sqsubseteq \exists S'.N' \in \mathsf{sat}(\mathcal{T})$ *such that* $a \in \bigcap_{A \in M'} A^{\mathcal{I}_{\mathcal{K}}}$*, and ((*$S \subseteq S'$*,* $N \subset N'$*) or (*$S \subset S'$*,* $N \subseteq N'$*)).*

then $a \cdot (\exists S.N) \in \mathsf{paths}(\mathcal{K})$*;*

(P3) if $w = w' \cdot (\exists S_0.N_0) \in \mathsf{paths}(\mathcal{K})$ *and an axiom* $M \sqsubseteq \exists S.N \in \mathsf{sat}(\mathcal{T})$ *satisfies*

- *(a)* $M \subseteq type(w)$*,*
- *(b)* $(S)^- \not\subseteq S_0$ *or* $N \not\subseteq type(w')$*, and*
- *(c) there is no* $M' \sqsubseteq \exists S'.N' \in \mathsf{sat}(\mathcal{T})$ *such that* $N_0 \subseteq M'$*, and ((*$S \subseteq S'$*,* $N \subset N'$*) or (*$S \subset S'$*,* $N \subseteq N'$*)),*

then $w \cdot (\exists S_0.N_0) \cdot (\exists S.N) \in \mathsf{paths}(\mathcal{K})$.

The intuition of generating $\mathsf{paths}(\mathcal{K})$ is to create new elements to witness existential axioms in a way that a new elements is created whenever it is really needed, and there is no two different elements generated to witness the same existential axiom. Let's consider one example to demonstrate the process of building $\mathsf{paths}(\mathcal{K})$.

Example 2.3.9. *Let* $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$*, where*

- $\mathcal{T}$ *contains*

$$\begin{array}{rclcrcl} A & \sqsubseteq & \exists r.B & \quad & B & \sqsubseteq & C \\ A_1 & \sqsubseteq & \leqslant 1 r_1.B_1 & & D & \sqsubseteq & \exists r_1^-.A_1 \\ A_1 & \sqsubseteq & \exists r_1.(B_1 \sqcap C_1) & & & & \end{array}$$

- $\mathcal{A}$ *contains*

$$A(d), A_1(d_1), B(d_2), D(d_2)$$

- *Apply inference rule in 2.1:*

  By $A \sqsubseteq \exists r.B$*,* $B \sqsubseteq C$*, and rule* $(\sqsubseteq_2)$ *we get* $A \sqsubseteq \exists r.(B \sqcap C)$*,*

  By $A_1 \sqsubseteq \leqslant 1 r_1.B_1$*,* $D \sqsubseteq \exists r_1^-.A_1$*,* $A_1 \sqsubseteq \exists r_1.(B_1 \sqcap C_1)$*, and rule* $\leqslant_2$ *we get* $D \sqcap B_1 \sqsubseteq C$*, and* $D \sqcap B_1 \sqsubseteq \exists r_1^-.A_1$*.*

- *Thus, we have* $\mathsf{sat}(\mathcal{T}) = \mathcal{T} \cup \{A \sqsubseteq \exists r.(B \sqcap C),\ D \sqcap B_1 \sqsubseteq C,\ D \sqcap B_1 \sqsubseteq \exists r_1^-.A_1\}$*, and* $\mathsf{cr}(\mathcal{T})$ *contains:*

$$\begin{array}{l} C(x) \leftarrow B(x). \\ C(x) \leftarrow B_1(x), C_1(x). \\ \bot(x) \leftarrow A_1(x), r_1(x, y_1), B_1(y_1), r_1(x, y_2), B_2(y_2), y_1 \neq y_2. \\ C_1(y) \leftarrow A_1(x), r(x, y), B_1(y) \end{array}$$

- *We now compute* $\mathsf{paths}(\mathcal{K})$*:*

  (1) $\{d, d_1, d_2\} \subseteq \mathsf{paths}(\mathcal{K})$*,*

(2) $d \in \mathsf{paths}(\mathcal{K})$ *and* $A \in type(d)$*, we have to consider two related axioms* $A \sqsubseteq \exists r.B$, $A \sqsubseteq \exists r.B \sqcap C$*. In this case, we do not generate two different new elements, it is enough to generate only one element according to* $A \sqsubseteq \exists r.B \sqcap C$*. So we get* $d \cdot (\exists r.B \sqcap C) \in \mathsf{paths}(\mathcal{K})$*. There is no existential axioms needed to be considered w.r.t.* $d \cdot (\exists r.B \sqcap C)$*.*

(3) $A_1 \in type(d_1)$*, we consider axiom* $A_1 \sqsubseteq \exists r_1.(B_1 \sqcap C_1)$ *and add* $d_1 \cdot (\exists r_1.(B_1 \sqcap C_1))$ *to* $\mathsf{paths}(\mathcal{K})$*. There is no existential axioms needed to be considered w.r.t.* $d_1 \cdot (\exists r_1.(B_1 \sqcap C_1))$*.*

(4) $\{B_1, D\} \subseteq type(d_2)$*, we consider axiom* $D \sqcap B_1 \sqsubseteq \exists r_1^-.A_1$ *and add* $d_2 \cdot (\exists r_1^-.A_1)$ *to* $\mathsf{paths}(\mathcal{K})$*.*

(5) Since $A_1 \in type(d_2 \cdot (\exists r_1^-.A_1))$*, we need to consider axiom* $A_1 \sqsubseteq \exists r_1.(B_1 \sqcap C_1)$*. Normally we generate a new element, however in this case we do not. The reason is that the condition in (P3) are violated. In other words, there already exists* $d_2 \in \mathsf{paths}(\mathcal{K})$ *such that* d_2 *is witness element for* $A_1 \sqsubseteq \exists r_1.(B_1 \sqcap C_1)$*.*

(6) Finally we have

$$\mathsf{paths}(\mathcal{K}) = \{d,\ d_1,\ d_2,\ d \cdot (\exists r.(B \sqcap C)),\ d_1 \cdot (\exists r_1.(B_1 \sqcap C_1)),\ d_2 \cdot (\exists r_1^-.A_1)\}$$

We now define the canonical model $\mathcal{I}_\mathcal{K}$ which has $\mathsf{paths}(\mathcal{K})$ as its domain. In principle, $\mathcal{I}_\mathcal{K}$ can be viewed as *chase* [1] of $\mathcal{A}$ with respect to axioms in $\mathsf{sat}(\mathcal{T})$.

Definition 2.3.10 (Canonical models)**.** *The canonical model $\mathcal{I}_\mathcal{K}$ is defined as follows:*

(I_1) $\Delta^{\mathcal{I}_\mathcal{K}} = \mathsf{paths}(\mathcal{K})$*;*

(I_2) $a^{\mathcal{I}_\mathcal{K}} = a$ *for all individuals* $a \in \mathsf{paths}(\mathcal{K})$*;*

(I_3) $A^{\mathcal{I}_\mathcal{K}} = \{e \in \mathsf{paths}(\mathcal{K}) \mid A \in type(e)\}$*, for all atomic concepts A;*

(I_4) *For all role names r,* $r^{\mathcal{I}_\mathcal{K}}$ *is the set pairs* (e_1, e_2) *such that*

- *i)* $e_1, e_2 \in \mathsf{N_I}$ *and* $(e_1, e_2) \in r^{\mathcal{I}_\mathcal{A}}$*,*
- *ii)* $e_2 = e_1 \cdot (\exists S.N)$ *and* $r \in S$*, or*
- *iii)* $e_1 = e_2 \cdot (\exists S.N)$ *and* $r^- \in S$*.*

We now show that if $\mathcal{I}_\mathcal{K}$ exists then it indeed is the model of $\mathcal{K}$, and it can be embedded homomorphically into arbitrary model of $\mathcal{K}$. Thus in the next step, it is enough to do query answering in $\mathcal{I}_\mathcal{K}$. In addition, we show that a KB $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$ is consistent iff $\mathcal{A} \cup \mathsf{cr}(\mathcal{T})$ has a model. This property allows us to reduce consistency checking of a KB to finding a model of a DATALOG program.

Lemma 2.3.11. *If a KB* $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ *is consistent, then* $\mathcal{A} \cup \mathsf{cr}(\mathcal{T})$ *is consistent.*

We next show that if a KB $\mathcal{K}$ is consistent, which implies $\mathcal{I}_\mathcal{K}$ is defined, then $\mathcal{I}_\mathcal{K}$ is the canonical model of $\mathcal{K}$.

Theorem 2.3.12. *If $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$ is an consistent KB, then*

(1) $\mathcal{I}_\mathcal{K}$ is a model of $\mathcal{K}$, and

(2) $\mathcal{I}_\mathcal{K}$ can be homomorphically embedded into any model of $\mathcal{K}$.

Corollary 2.3.13. *$\mathcal{K} = (\mathcal{T}, \mathcal{A})$ is consistent iff $\mathcal{A} \cup \mathsf{cr}(\mathcal{T})$ is consistent.*

The above corollary already shows a good result: in database terms, checking consistency over Horn-$\mathcal{ALCHIQ}_\sqcap$ is reduced to evaluating the (plain) DATALOG query $\mathsf{cr}(\mathcal{T})$ over the database $\mathcal{A}$. In addition, the existence of a homomorphism from $\mathcal{I}_\mathcal{K}$ to any other model of a consistent $\mathcal{K}$ also yields:

Corollary 2.3.14. *If $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ is consistent and q is a query, then $ans(\mathcal{K}, q) = ans(\mathcal{I}_\mathcal{K}, q)$.*

2.3.3 Query Rewriting

Assume a consistent $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ and a query q. By the above corollary, $ans(\mathcal{K}, q) = ans(\mathcal{I}_\mathcal{K}, q)$ and thus for answering q over $\mathcal{K}$ it suffices to concentrate on a single model of $\mathcal{K}$. However, $\mathcal{I}_\mathcal{K}$ may be infinite and thus cannot be explicitly constructed for the query answering purpose. Our goal next is to rewrite q into a finite set of rules Q such that $ans(\mathcal{I}_\mathcal{K}, q) = ans(\mathcal{I}_\mathcal{A}, Q)$, where $\mathcal{I}_\mathcal{A}$ is the least model of the DATALOG program $\mathcal{A} \cup \mathsf{cr}(\mathcal{T})$. This yields an algorithm for answering any q over $\mathcal{K}$. Importantly, the rewriting is only dependent on the TBox, and thus Q can be used to answer q over any $\mathcal{K}' = (\mathcal{T}, \mathcal{A}')$.

Definition 2.3.15 (Query rewriting). *Let q a conjunctive query and $\mathcal{T}$ a Horn-$\mathcal{ALCHIQ}_\sqcap$ TBox. We write $q \rightarrow_\mathcal{T} q'$ if q' can be obtained from q by the following steps:*

(S1) Select a non-distinguished variable x in q such that

- *$r(x, x) \notin q'$ for all r, and*
- *$r_1(d_1, x), r_2(d_2, x) \notin q$ for $d_1, d_2 \in N_I$ and $d_1 \neq d_2$.*

(S2) Let

(i) $S = \{r \mid \exists u : r(u, x) \in q\} \cup \{r^- \mid \exists u : r(x, u) \in q\}$,

(ii) $N = \{A \mid A(x) \in q\}$, and

(iii) $T = \{y \mid \exists r : r(y, x) \in q \vee r(x, y) \in q\}$. Note that $x \notin T$.

(S3) Select some M such that $M \sqsubseteq \exists S'.N' \in \mathsf{sat}(\mathcal{T})$ with $S \subseteq S'$ and $N \subseteq N'$.

(S4) Remove all atoms of q where x occurs.

(S5) If there is an individual $d \in T$, then rename all $y \in T$ of q by d, otherwise rename $y \in T$ of q by x.

(S6) Add the atoms $\{A(x) \mid A \in M\}$ to q.

We write $q \rightarrow^*_{\mathcal{T}} q'$ if $q = q'$ or there is $n > 0$ such that $q_0 \rightarrow_{\mathcal{T}} q_1, \cdots, q_{n-1} \rightarrow_{\mathcal{T}} q_n$ with $q_0 = q$ and $q_n = q'$. That is, $q \rightarrow^*_{\mathcal{T}} q'$ means that q' can be obtained from q by making finitely many rewrite iterations. For a query q, we let $\mathsf{rew}_{\mathcal{T}}(q) = \{q' \mid q \rightarrow^*_{\mathcal{T}} q'\}$ be the set all rewritings of q w.r.t. $\mathcal{T}$.

We now look at a specific example.

Example 2.3.16. *Let $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$, where*

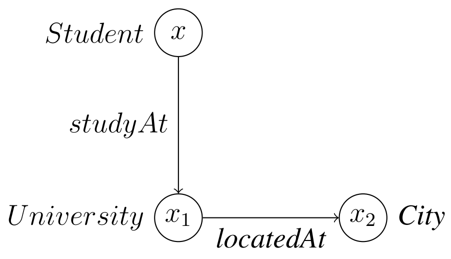
- *The TBox $\mathcal{T}$ contains*

$$\begin{array}{rcl} MasterStudent & \sqsubseteq & \exists studyAt.University \\ University & \sqsubseteq & \exists locatedAt.City \end{array}$$

- *The ABox $\mathcal{A} = \{Student(d), MasterStudent(d)\}$*

- *The query Q:*

$$q(x) \leftarrow Student(x), studyAt(x, x_1), University(x_1), locatedAt(x_1, x_2), City(x_2).$$



- *We compute $\mathsf{sat}(\mathcal{T})$ and see that $\mathsf{sat}(\mathcal{T}) = \mathcal{T}$.*

- *We first choose non-deterministically a non-distinguished variable x_1, and compute:*

$$\begin{array}{l} S = \{studyAt, locatedAt^-\} \\ N = \{University\} \\ T = \{x, x_2\} \end{array}$$

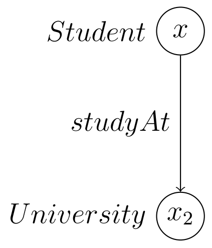
Since there is no axiom in $\mathsf{sat}(\mathcal{T})$ satisfies conditions in (S3), we stop.

- *We choose another non-distinguished variable x_2, and compute:*

$$\begin{aligned} S &= \{locatedAt\} \\ N &= \{City\} \\ T &= \{x_1\} \end{aligned}$$

We find that axiom $University \sqsubseteq \exists locatedAt.City$ satisfies conditions in (S3), so we get the rewritten query Q_1:

$$q(x) \leftarrow Student(x), studyAt(x, x_2), University(x_2).$$



- *Regarding query Q_1, we choose the only non-distinguished variable x_2, and compute:*

$$\begin{aligned} S &= \{studyAt\} \\ N &= \{University\} \\ T &= \{x\} \end{aligned}$$

There is an axiom $Student \sqsubseteq \exists studyAt.University$ satisfies condition in (S3), so we get the rewritten query Q_2:

$$q(x_2) \leftarrow Student(x_2), MasterStudent(x_2).$$

- *Finally, we get $REW_{\mathcal{T}}(Q) = \{Q, Q_1, Q_2\}$, and a DATALOG program $\mathcal{A} \cup \mathsf{cr}(\mathcal{T}) \cup REW_{\mathcal{T}}(Q)$=*

$$\begin{array}{rcl} & & Student(d). \\ & & MasterStudent(d). \\ q(x) & \leftarrow & Student(x), studyAt(x, x_1), University(x_1), locatedAt(x_1, x_2), City(x_2). \\ q(x) & \leftarrow & Student(x), studyAt(x, x_2), University(x_2). \\ q(x_2) & \leftarrow & Student(x_2), MasterStudent(x_2). \end{array}$$

This program has the least model $\mathcal{I}_{\mathcal{A}} = \{MasterStudent(d), Student(d), q(d)\}$, thus d is answer for the original query Q.

We can check this answer again by building model $\mathcal{I}_{\mathcal{K}}$ of $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$, and match query Q to $\mathcal{I}_{\mathcal{K}}$. The query match is depicted in Figure 2.3.

Proposition 2.3.17. *Assume a consistent $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ and a query q. Let $\mathcal{I}_{\mathcal{A}}$ be the least model $\mathcal{A} \cup \mathsf{cr}(\mathcal{T})$. Then $ans(\mathcal{I}_{\mathcal{K}}, q) = ans(\mathcal{I}_{\mathcal{A}}, \mathsf{rew}_{\mathcal{T}}(q))$.*

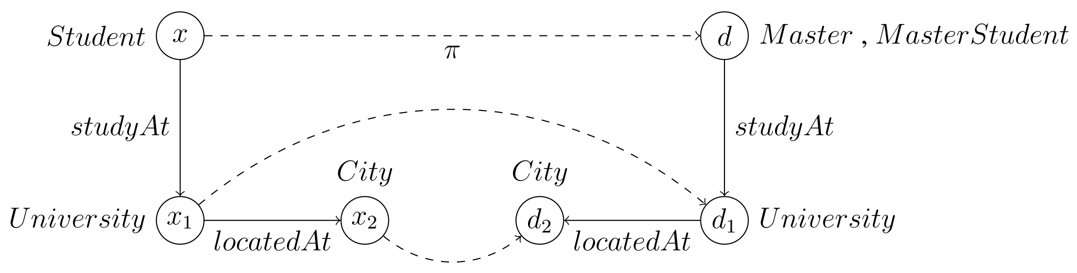


Figure 2.3: A map π for Q to the canonical model $\mathcal{I}_{\mathcal{K}}$

By the above reduction, we can answer q over $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ by posing $\mathsf{rew}_{\mathcal{T}}(q)$ over the DATALOG program $\mathcal{A} \cup \mathsf{cr}(\mathcal{T})$. However, computer $\mathsf{sat}(\mathcal{T})$ could be exponentially hard.

Proposition 2.3.18. $\mathsf{sat}(\mathcal{T})$ *can be computed in exponential time in* $|\mathcal{T}|$.

Proposition 2.3.19. $|\mathsf{rew}_{\mathcal{T}}(q)|$ *is exponential in* $|\mathcal{T}| + |q|$.

2.4 KAOS Reasoner

In this part, we introduce the prototype system KAOS which provides query answering services for Horn-$\mathcal{SHIQ}$. To the best of our knowledge, this is the first reasoner which supports conjunctive queries (without DL-safeness restriction) over expressive DLs. To show the efficiency, we will test KAOS on several benchmarks.

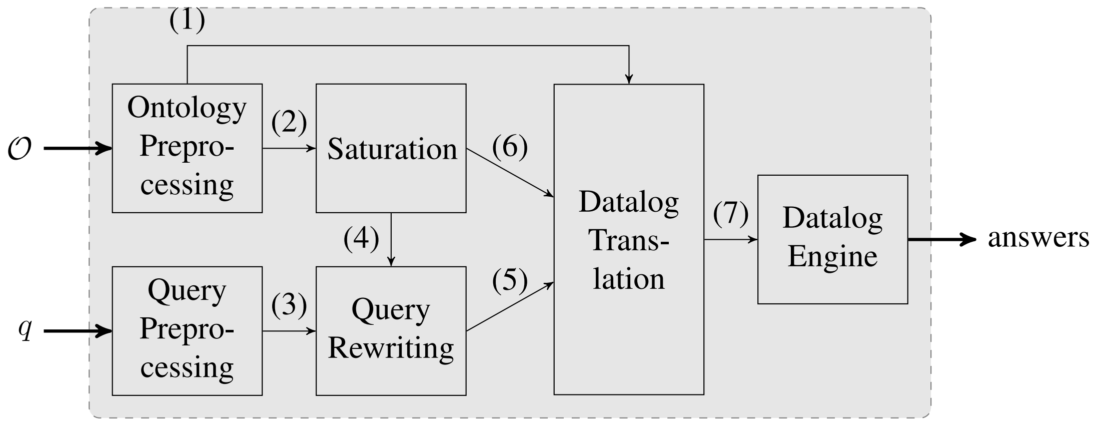
2.4.1 System Architecture

The system consists of the main components depicted in Figure 2.4.

2.4.1.1 Ontology preprocessing

This component is responsible for ontology parsing, ontology normalization, profile checking, and ontology encoding.

- Ontology parsing: In our implementation, we use the OWL API library [53] to manage ontologies. This enables the system to manipulate ontologies in different formats such as RDF/XML, OWL/XML, OWL Functional Syntax.
- Profile checking: After doing the Negation Normal Form (NNF) normalization, we are able to check whether the ontology is in Horn-$\mathcal{SHIQ}$ profile based on the grammar presented in Table 2.2. It was shown in [63] that an axiom $C \sqsubseteq D$ is in Horn-$\mathcal{SHIQ}$ if the concept expression $\neg C \sqcup D$ has the form C_1^+.



(1)	ABox assertions	(4)	Existential axioms	(7)	Datalog rules
(2)	TBox axioms	(5)	Rewritten queries		
(3)	Conjunctive queries	(6)	Axioms		

Figure 2.4: KAOS architecture

$$\begin{array}{lcl}
C_1^+ & ::= & \top \mid \bot \mid \neg C_1^- \mid C_1^+ \sqcap C_1^+ \mid C_0^+ \sqcup C_1^+ \mid \exists R.C_1^+ \mid \forall S.C_1^+ \mid \forall R.C_0^+ \mid \geqslant nR.C_1^+ \mid \leqslant 1R.C_0^- \mid A \\
C_1^- & ::= & \top \mid \bot \mid \neg C_1^+ \mid C_0^- \sqcap C_1^- \mid C_1^- \sqcup C_1^- \mid \exists S.C_1^- \mid \exists R.C_0^- \mid \forall R.C_1^- \mid \geqslant 2R.C_0^- \mid \leqslant nR.C_1^+ \mid A \\
C_0^+ & ::= & \top \mid \bot \mid \neg C_0^- \mid C_0^+ \sqcap C_0^+ \mid C_0^+ \sqcup C_0^+ \mid \forall R.C_0^+ \\
C_0^- & ::= & \top \mid \bot \mid \neg C_0^+ \mid C_0^- \sqcap C_0^- \mid C_0^- \sqcup C_0^- \mid \exists R.C_0^- \mid A
\end{array}$$

A, R and S denote the set of all atomic concepts, roles, and simple role, respectively.

Table 2.2: A grammar defining Horn-$\mathcal{SHIQ}$ axioms.

- Ontology normalization: TBox axioms are transformed into normal form close to the one presented in [65]. To do so, we first exhaustively apply the rules P1 to the knowledge base, and then exhaustively apply the rules P2. Rules P1 and P2 are described in Table 2.3. The last two translations in P2 are used to eliminate qualified number restrictions and transitive roles.

- Ontology encoding: In the saturation and query rewriting part we often operate on a huge number of sets of concept and role names. Therefore, we encode ontology concept names, role names into binary numbers, and use bitsets to represent sets of concept and role names. The current version of our system uses Trove[3] library to manipulate integer hash sets.

[3] `http://trove.starlight-systems.com/`

$$
\begin{array}{rcll}
P1: & & & \\
\hat{A} & \sqsubseteq & \hat{C} & \mapsto \{\hat{A} \sqsubseteq D, D \sqsubseteq \hat{C}\} \\
\hat{A} \sqcap B & \sqsubseteq & C & \mapsto \{\hat{A} \sqsubseteq D, D \sqcap B \sqsubseteq C\} \\
B \sqcap \hat{A} & \sqsubseteq & C & \mapsto \{\hat{A} \sqsubseteq D, D \sqcap B \sqsubseteq C\} \\
A & \sqsubseteq & B \sqcup C & \mapsto \{A \sqsubseteq D, D \sqcap NNF(\neg B) \sqsubseteq C\} \text{ if } B \in C_0^+ \\
 & & & \mapsto \{A \sqsubseteq D, D \sqcap NNF(\neg C) \sqsubseteq D\} \text{ otherwise} \\
\exists R.\hat{A} & \sqsubseteq & B & \mapsto \{\hat{A} \sqsubseteq D, \exists R.D \sqsubseteq B\} \\
A & \sqsubseteq & \exists R.\hat{C} & \mapsto \{A \sqsubseteq \exists R.D, D \sqsubseteq \hat{C}\} \\
A & \sqsubseteq & \forall R.\hat{C} & \mapsto \{A \sqsubseteq \forall R.D, D \sqsubseteq \hat{C}\} \\
A & \sqsubseteq & \geqslant nR.\hat{B} & \mapsto \{A \sqsubseteq \geqslant nR.D, D \sqsubseteq \hat{B}\} \\
A & \sqsubseteq & \leqslant 1R.\hat{B} & \mapsto \{A \sqsubseteq \leqslant nR.D, D \sqsubseteq \hat{B}\} \\
P2: & & & \\
A & \sqsubseteq & B \sqcap C & \mapsto \{A \sqsubseteq B, A \sqsubseteq C\} \\
\hat{A} \sqcup B & \sqsubseteq & C & \mapsto \{\hat{A} \sqsubseteq C, B \sqsubseteq C\} \\
B \sqcup \hat{A} & \sqsubseteq & C & \mapsto \{\hat{A} \sqsubseteq C, B \sqsubseteq C\} \\
A & \sqsubseteq & \neg B & \mapsto \{A \sqsubseteq D, D \sqcap B \sqsubseteq \bot\} \\
\exists R.A & \sqsubseteq & B & \mapsto \{A \sqsubseteq \forall \bar{R}.B\}; \text{ Where } \bar{R} \text{ is inverse of } R \\
A & \sqsubseteq & \forall R.B & \mapsto \{A \sqsubseteq \forall R_T.B_T, B_T \sqsubseteq \forall R_T.B_T, B_T \sqsubseteq B\}; \\
 & & & \quad \text{for transitive subrole } R_1 \text{ of } R; B_T \text{ is a fresh concept name.} \\
A & \sqsubseteq & \geqslant nS.C & \mapsto \{A \sqsubseteq \exists S.B_i, B_i \sqsubseteq C, B_i \sqcap B_j \sqsubseteq \bot\}; 1 \leqslant i \leqslant j \leqslant n
\end{array}
$$

Table 2.3: Normal form translation for Horn-$\mathcal{SHIQ}$. A, B, C are concept names. $\hat{A}$,$\hat{C}$ are concept expressions. D is a fresh concept name. R is a role, and S is a simple role.

2.4.1.2 Query preprocessing

This component is simply responsible for parsing queries in SPARQL syntax. It takes a conjunctive query in SPARQL syntax and returns the corresponding conjunctive query in an internal data structure.

2.4.1.3 Saturation

The responsibility of this component is to exhaustively apply saturation rules presented in section 2.3 . The input of this component is TBox axioms and the output consists of: (1) existential axioms and (2) other type of axioms. Existential axioms are then used by *query rewriting* component to rewrite the input query. The rest of axioms are translated into Datalog rules.

2.4.1.4 Query rewriting

This component rewrites the input query. The procedure for query rewriting is described in Algorithm 2.1. Procedure $rewrite(q)$ is called recursively to compute the set of rewritten queries $\mathsf{rew}_{\mathcal{T},q}$. The set $\mathsf{rew}_{\mathcal{T},q}$ is initialized to contain only the original query q. For each non-distinguished variable x in query q that does not contain a loop at x i.e. $r(x, x)$, we check if there is an applicable existential axioms in $\mathsf{sat}(\mathcal{T})$ to rewrite q. If there is such axiom, we rewrite the query q and add new rewritten queries to $\mathsf{rew}_{\mathcal{T},q}$. The rewriting procedure is repeated with the new added queries. By recursively applying $rewrite(q)$, we assure that every query is checked to be rewritten. The algorithm terminates when all non-distinguished variables in all queries are chosen and processed.

In many cases we have to operate on a huge number of axioms, thus we use *inverted index* which is normally used in information retrieval systems. This index data structure allows the system efficiently search through the set of existential axioms while rewrite queries.

Algorithm 2.1: ComputeREW

```
Data: program $q$; TBox $\mathcal{T}$
Result: UCQ $\mathsf{rew}_{\mathcal{T},q}$
$\mathcal{T}' \longleftarrow \mathsf{sat}(\mathcal{T})$
static $\mathsf{rew}_{\mathcal{T},q} \longleftarrow \emptyset$
$rewrite(q)$
return $\mathsf{rew}_{\mathcal{T},q}$

Procedure rewrite(q)
Data: rule $q$
$\mathsf{rew}_{\mathcal{T},q} \longleftarrow \mathsf{rew}_{\mathcal{T},q} \cup \{q\}$
forall the non-distinguished variables $x$ of $q$ do
    if there is no $r$ with $r(x,x) \in q$ then
        $S \longleftarrow \{r \mid \exists u : r(u,x) \in q\} \cup \{r^- \mid \exists u : r(x,u) \in q\}$
        $N \longleftarrow \{A \mid A(x) \in q\}$
        forall the $M \sqsubseteq \exists S'N' \in \mathsf{sat}(\mathcal{T})$ do
            if $S \subseteq S'$ and $N \subseteq N'$ then
                Let $q'$ be the rewriting of $q$ w.r.t. $x$ and $M$
                if $q' \notin \mathcal{P}'$ then
                    $rewrite(q')$
                end
            end
        end
    end
end
```

2.4.1.5 Datalog translation

This component translates rewritten queries and the rest of axioms into Datalog rules. The translation for conjunctive queries is straightforward, and the translation for the rest of axioms follows the rules presented in section 2.3.3 . This component also takes as the input ABox assertions and translates them to Datalog facts. In this way, we have a program constituted by Datalog translation of the rewritten queries, TBox axioms and ABox assertions.

2.4.1.6 Datalog engine

The Datalog program obtained from the *Datalog translation* component is evaluated by a Datalog engine such as DLV [69] and Clingo [44]. The answers for the input query are the evaluation of corresponding query predicate in the Datalog program.

2.4.2 System Usage

KAOS is implemented in Java version 1.6 and can be run in Windows and Unix-based operating systems as a stand-alone java jar file. KAOS supports ontologies in different formats: RDF/XML, OWL/XML, OWL Functional Syntax, Turtle syntax, the Manchester OWL syntax, and KRSS syntax. The supported input query is in SPARQL syntax. To run KAOS we use command line in the following format:

```
java -jar kaos.jar -ontology <file1> -sparql <file2> -dlv <path>

<file1> the ontology file to be read
<file2> the sparql file to be query
<dlv_path> the path of dlv datalog engine
Option:
[-verbose <verbose_level ] specifies verbose category
                           the default level is  0

Example: java -jar kaos.jar -ontology university.owl
                            -sparql q1.sparql
                            -dlv /usr/bin/dlv
```

Example 2.4.1. *Ontology file* university.owl *is an ontology* $\mathcal{O} = (\mathcal{T}, \mathcal{A})$

- *TBox* $\mathcal{T}$:

$$\begin{aligned} Student &\sqsubseteq \exists attends.Course \\ Student &\sqsubseteq \forall attends.Course \end{aligned}$$

- *ABox $\mathcal{A}$:*

$$\begin{array}{lll} Student(john), & attends(john, computer\ network), & Course(machine\ learning), \\ Student(paul), & attends(peter, database\ systems), & Course(complexity\ theory), \\ Student(peter), & & Course(computer\ network) \end{array}$$

- *SPARQL query file* query.sparql *contains the query to get all students who attend at least one course:*

```
PREFIX uri:<http://www.semanticweb.org/testontologies.owl#>
SELECT ?student
WHERE {
?student a uri:Student ;
           uri:attends ?course .
?course a  uri:Course . }
```

- *We run KAOS by the following command:*

```
java -jar kaos.jar -ontology university.owl
                   -sparql query.sparql
                   -dlv /usr/bin/dlv
```

- *The generated Datalog program is:*

$$\begin{array}{rcl} q(X) & \leftarrow & Student(X), attends(X, Y), Course(Y). \\ q(X) & \leftarrow & Student(X). \\ Course(Y) & \leftarrow & Student(X), attends(X, Y). \end{array}$$

$$\begin{array}{lll} Student(john). & attends(john, computer\ network). & Course(machine\ learning). \\ Student(paul). & attends(peter, database\ systems). & Course(complexity\ theory). \\ Student(peter). & & Course(computer\ network). \end{array}$$

- *The answer of the query:*

$$john, paul, peter$$

2.4.3 Experiments

In our experiments, we carried out two type of tests:

1. Downscaling test: to see the performance of the system when it runs with ontologies in less expressive languages than Horn-$\mathcal{SHIQ}$.
2. Scalability test: to see the performance of the system when is runs with ontologies in "true" Horn-$\mathcal{SHIQ}$.

The experiments were done on a Pentium Core2 Duo 2.00GHZ machine with 2GB RAM running Ubuntu 10.04. The Java virtual machine is set with 500MB heap memory. The running time is in milliseconds and we stop running if the running time is over five minutes. Our system supports queries in SPARQL syntax, but to simplify notion we present queries in Datalog notation.

We briefly introduce the ontologies in our tests. Adolena [60] is an ontology about people with disabilities and used to enhance web portals with ontology-based data access. Stock is an ontology that presents the application domain of financial institutions. Vicody [4] is an ontology about European history. Its TBox is relatively simple and consists of role inclusions, concept inclusion, and domain and range specifications. LUBM [47] is a benchmark for testing performance of ontology management and reasoning systems. It describes the domain of organizational structure of universities. PATH5 is a synthetic ontology created in the test suite of REQUIEM, a tool for query rewriting developed at the University of Oxford. Its concepts represent the length from one to five of the paths in a graph. PATH8 ontology is an extension of PATH5.

For readability, we put the tables into the appendix of this chapter.

2.4.3.1 Downscaling Test

In this test, we run KAOS on ontologies in less expressive languages than Horn-$\mathcal{SHIQ}$ and compare with REQUIEM [83]. Two types of experiments are distinguished in the comparison. The first one is to evaluate the query rewriting process, the second one is to evaluate the Datalog programs obtained from the rewritten queries and ABox assertions.

Query Rewriting Evaluation

First, we make the comparison with REQUIEM on ontologies provided in the test suite of REQUIEM. This test suite contains fives ontologies: Vicodi, StockExchange, University, Adolena, and PathX. The queries are presented in Table 2.4, and the experiment results are in Table 2.5.

In principle, the original query will be rewritten based on axioms in TBox. Each approach has different ways to use these TBox axioms. In our approach we use Datalog rules to capture the meaning of these axioms except for existential axioms. Thus, when counting the number of rewritten queries, we take into account the Datalog rules that relate to rewritten queries. On the column for rewritten queries/rules by KAOS in the Table 2.5, we distinguish two sub columns: the first column contains the number of queries generated by our rewriting algorithm, the second one is the total number of generated queries and the Datalog rules related to these queries. From the experiment results we observe that:

- In all cases, the time for rewriting a query by KAOS is much shorter than the time used by REQUIEM.

[4]http://www.vicodi.org/

- KAOS often generates fewer rewritten queries/rules than REQUIEM-N, and REQUIEM-G generates fewer rewritten queries than KAOS in average. The number of rewritten queries generated by REQUIEM-F is comparable with those generated by KAOS.

The number of rewritten queries depends on the insight of the rewriting algorithm, but the running time much depends on implementation techniques. Commonly, before doing reasoning, the ontology is normalized, and this may affect the running time of other processes. We do not know how long does it take for REQUIEM to normalize ontologies, thus we cannot take this time into account while comparing running time between REQUIEM and our system. Nevertheless, we provide the normalization time used by our system and see that even when we count the normalization time as part of rewriting time, this total time is comparable with the rewriting time of REQUIEM. Table 2.6 provides the time used to normalize ontologies in the test suite.

Query Answering Evaluation

We make the comparison between KAOS and REQUIEM on LUBM ontologies with different ABoxes. The rewritten queries/rules are put together with facts corresponding to ABox assertions to constitute Datalog programs. These Datalog programs are then evaluated by Datalog engines: DLV and Clingo. The tested ontologies have the same TBox but different ABoxes ranging from 10MB to 40MB. They are downloadable from KAON2's website [5]. The queries are given in the Table 2.7. The first fives queries in this table are get from test suite of REQUIEM. We notice that the university ontology in the test suite of REQUIEM is not exactly the same as original LUBM ontology, so the rewritten results can be different with those in Table 2.5. We run Datalog programs with both DLV and Clingo, the running time are presented in Table 2.8 and Table 2.9. For readability, we mark the best total running time with a blue color.

According to these results, we have some observations as follows:

- Regarding query rewriting, KAOS always runs faster than REQUIEM, especially than REQUIEM-G. However, REQUIEM-G produces fewer rewritten queries than KAOS w.r.t. $Q_1, \ldots, Q_5$. For the rest, KAOS generates fewer written queries.
- Regarding the Datalog running time, it is not always true that the more rules Datalog program has, the slower the Datalog engine solves it.
- In almost all queries, Datalog programs obtained from rewritten queries by KAOS take less time to be evaluated than those by REQUIEM-N. However, Datalog program obtained from rewritten queries by REQUIEM-F and REQUIEM-G often takes less times to be solved than those by KAOS, except for queries Q7, Q8.
- In most of the tests, with respect to the total time of query rewriting and Datalog evaluation time, KAOS outperforms REQUIEM-N, F, and G.

[5]`http://kaon2.semanticweb.org/`

2.4.3.2 Scalability Test

Finally, we test our system with ontologies in full Horn-$\mathcal{SHIQ}$. To create an ontology having full features of Horn-$\mathcal{SHIQ}$, we customize UOBM benchmark [77], which is an extension of LUBM. In this test, we do not run KAOS with different ABoxes, but we try to run it with queries having different topologies and constructors disallowed in other less expressive languages. These queries are presented in Table 2.10.

The experiment results are presented in Table 2.11. Because there are no available systems supporting query answering over ontologies in Horn-$\mathcal{SHIQ}$, we cannot compare KAOS with others. The experiment results show that our system scales well.

2.5 Inline Evaluation of DL-Programs over Horn-$\mathcal{SHIQ}$

Conjunctive queries over Horn-$\mathcal{SHIQ}$ ontologies can be seen as a special case of a tight coupling of a single rule and a Horn-$\mathcal{SHIQ}$ ontology. Actually the datalog rewriting of Horn-$\mathcal{SHIQ}$ can be further employed in the other combination of rules and ontologies for efficient reasoning.

The datalog encoding of Horn-$\mathcal{SHIQ}$ can also be used for the inline evaluation [101] of dl-programs. DL-programs are a loosely coupled approach for the combination of rules and ontologies [33]. The rule components can access the ontology, and vice versa. A prominent feature is that the rules can be very expressive using default negations and disjunction in the head. The traditional way of reasoning over dl-programs is using a rule reasoner and an ontology reasoner accessing the two components, which is not very efficient. In D3.3 [40] we showed that for certain fragments of DLs which enjoys the datalog-rewritablity property, the reasoning of dl-programs can be reduced to reasoning over a DATALOG$^{\neg}$ program, thus can be implemented using only one datalog reasoner. With a small extension of the definition of datalog-rewritablity (removing the polynomial rewriting time restriction), the datalog rewriting of Horn-$\mathcal{SHIQ}$ is well suited in the framework of inline evaluation. We briefly show this by the following examples.

We first recall how inline evaluation works for dl-programs by the following example [101].

Example 2.5.1. *Let $\mathcal{KB} = (\Sigma, P)$ be a dl-program, where $\Sigma = \{\ C \sqsubseteq D\ \}$ and $P = \{p(a) \leftarrow;\quad s(a) \leftarrow;\quad s(b) \leftarrow;$
$q \leftarrow DL[C \uplus s; D](a), not\ DL[C \uplus p; D](b)\ \}$.*

Each dl-atom sends up a different input/hypothesis to Σ and that entailments for different inputs might be different. To this purpose, we copy Σ to new disjoint equivalent versions for each dl-atom, i.e., for each distinct dl-atom λ, we define a new knowledge base Σ_λ that results from replacing all concept and role names by a λ-subscripted version. Thus, for the set $\Lambda_P = \{\lambda_1 \triangleq C \uplus s, \lambda_2 \triangleq C \uplus p\}$ of dl-atoms, we have $\Sigma_{\lambda_i} = \{\ C_{\lambda_i} \sqsubseteq D_{\lambda_i}\ \}$, $i = 1, 2$.

We translate these disjoint ontologies to a DATALOG *program, resulting in the rules* $\Phi(\Sigma_{\lambda_i}) = \{D_{\lambda_i}(X) \leftarrow C_{\lambda_i}(X)\ \}$, $i = 1, 2$.

The inputs in the dl-atoms Λ_P *can then be encoded as rules* $\rho(\Lambda_P)$*:*

$$\{C_{\lambda_1}(X) \leftarrow s(X); \quad C_{\lambda_2}(X) \leftarrow p(X)\}.$$

It remains to replace the original dl-rules with rules not containing dl-atoms: P^{ord} *results from replacing each dl-atom* $DL[\lambda; Q](\mathbf{t})$ *in* P *with a new atom* $Q_\lambda(\mathbf{t})$*, such that* P^o*is the* DATALOG$^\neg$ *program*

$$P^o \triangleq \{p(a) \leftarrow; \quad s(a) \leftarrow; \quad s(b) \leftarrow; \quad q \leftarrow D_{\lambda_1}(a), not\ D_{\lambda_2}(b)\}.$$

One can see that indeed $\mathcal{KB} \models q$ *and* $\Phi(\Sigma_{\lambda_1}) \cup \Phi(\Sigma_{\lambda_2}) \cup P^o \cup \rho(\Lambda_P) \models q$*, effectively reducing reasoning w.r.t. the dl-program to a* DATALOG$^\neg$ *program.*

When the ontology component is in Horn-$\mathcal{SHIQ}$, we can apply the new rewriting in this report.

Example 2.5.2. *Let* $\mathcal{KB}' = (\Sigma', P)$ *be a dl-program, where* $\Sigma' = \{\ C \sqsubseteq \exists r.A,\ A \sqsubseteq B,\ \exists r.B \sqsubseteq D\ \}$ *and* P *is same with the previous example.*

Normalization of Σ'*:*

$$\Sigma'_N = \{\ C \sqsubseteq \exists r.A,\ A \sqsubseteq B,\ B \sqsubseteq \forall r^-.D\ \}.$$

After saturation, we have $\mathsf{sat}(\Sigma'_N) = \Sigma'_N \cup \{C \sqsubseteq \exists r.(A \sqcap B),\ C \sqsubseteq D\}$.

The completion rules $\mathsf{cr}(\Sigma'_N)$ *for* Σ'_N *are :*

$$\begin{aligned}
&B(X) \leftarrow A(X).\\
&D(X) \leftarrow C(X).\\
&D(X) \leftarrow r(X, Y), B(Y).
\end{aligned}$$

In dl-programs, we only need to deal with instance query, in which all variables involved are distinguished. Therefore, query rewriting will not generate new rules.

The rest rules of datalog rewriting for $\mathcal{KB}'$ *are same with those in the previous example. Now we are done.*

To summarize, for the inline evaluation of dl-programs over Horn-$\mathcal{SHIQ}$ ontology, the completion rules are enough. Note that there is an extension of dl-programs, called cq-programs [32], which applies CQ over the ontology. If we want to inline evaluate cq-programs, the query rewriting part is also needed.

2.6 Summary

Finding an efficient solution for query answering in expressive DLs is important. The existing practical approaches are only for query answering in lightweight languages such as DL-Lite and EL. This raises the need for more expressive languages and Horn-$\mathcal{SHIQ}$ is an attractive candidate. As there have been no efficient techniques for query answering over ontologies in Horn-$\mathcal{SHIQ}$, we proposed in this work a practical query rewriting approach for this issue.

We also provided a prototype implementation and primary experimental results. To the best of our knowledge, this is the first system offering conjunctive query answering services over ontologies in Horn-$\mathcal{SHIQ}$, that allows for unknown individuals in the queries. Although query answering in Horn-$\mathcal{SHIQ}$ is challenging, KAOS has shown promising results.

We showed that the datalog rewriting of Horn-$\mathcal{SHIQ}$ is well suited in the framework of inline evaluation for dl-programs. A new version of DReW (targeting Horn-$\mathcal{SHIQ}$) can be implemented based on the results of this work.

2.7 Appendix

2.7.1 Tables for downscaling test

Ontology		Queries
Adolena	Q1	Q(?0) ← Device(?0), assistsWith(?0,?1)
	Q2	Q(?0) ← Device(?0), assistsWith(?0,?1), UpperLimbMobility(?1)
	Q3	Q(?0) ← Device(?0), assistsWith(?0,?1), Hear(?1), affects(?2,?1), Autism(?2)
	Q4	Q(?0) ← Device(?0), assistsWith(?0,?1), PhysicalAbility(?1)
	Q5	Q(?0) ← Device(?0), assistsWith(?0,?1), PhysicalAbility(?1), affects(?2,?1), Quadriplegia(?2)
	Q6	Q(?0) ← Device(?0), assistsWith(?0,?1), ReadingDevice(?1)
	Q7	Q(?0) ← Device(?0), assistsWith(?0,?1), ReadingDevice(?1), assistsWith(?1,?2), SpeechAbility(?2)
	Q8	Q(?0) ← Device(?0), assistsWith(?0,?1), UpperLimbMobility(?1), assistsWith(?1,?2), MovementAbility(?2)
Path5	Q1	Q(?0) ← edge(?0,?1)
	Q2	Q(?0) ← edge(?0,?1), edge(?1,?2)
	Q3	Q(?0) ← edge(?0,?1), edge(?1,?2), edge(?2,?3)
	Q4	Q(?0) ← edge(?0,?1), edge(?1,?2), edge(?2,?3), edge(?3,?4)
	Q5	Q(?0) ← edge(?0,?1), edge(?1,?2), edge(?2,?3), edge(?3,?4), edge(?4,?5)
Path8	Q1	Q(?0) ← edge(?0,?1)
	Q2	Q(?0) ← edge(?0,?1), edge(?1,?2)
	Q3	Q(?0) ← edge(?0,?1), edge(?1,?2), edge(?2,?3)
	Q4	Q(?0) ← edge(?0,?1), edge(?1,?2), edge(?2,?3), edge(?3,?4)
	Q5	Q(?0) ← edge(?0,?1), edge(?1,?2), edge(?2,?3), edge(?3,?4), edge(?4,?5)
	Q6	Q(?0) ← edge(?0,?1), edge(?1,?2), edge(?2,?3), edge(?3,?4), edge(?4,?5), edge(?5,?6)
	Q7	Q(?0) ← edge(?0,?1), edge(?1,?2), edge(?2,?3), edge(?3,?4), edge(?4,?5), edge(?5,?6), edge(?6,?7)
	Q8	Q(?0) ← edge(?0,?1), edge(?1,?2), edge(?2,?3), edge(?3,?4), edge(?4,?5), edge(?5,?6), edge(?6,?7, edge(?7,?8)
Stock	Q1	Q(?0) ← StockExchangeMember(?0)
	Q2	Q(?0,?1) ← Person(?0), hasStock(?0,?1), Stock(?1)
	Q3	Q(?0,?1,?2) ← FinantialInstrument(?0), belongsToCompany(?0,?1), Company(?1), hasStock(?1,?2), Stock(?2)
	Q4	Q(?0,?1,?2) ← Person(?0), hasStock(?0,?1), Stock(?1), isListedIn(?1,?2), StockExchangeList(?2)
	Q5	Q(?0,?1,?2,?3) ← FinantialInstrument(?0), belongsToCompany(?0,?1), Company(?1), hasStock(?1,?2), Stock(?2), isListedIn(?1,?3), StockExchangeList(?3)
University	Q1	Q(?0)← worksFor(?0,?1), affiliatedOrganizationOf(?1,?2)
	Q2	Q(?0,?1) ← Person(?0), teacherOf(?0,?1), Course(?1)
	Q3	Q(?0,?1,?2) ← Student(?0), advisor(?0,?1), FacultyStaff(?1), takesCourse(?0,?2), teacherOf(?1,?2), Course(?2)
	Q4	Q(?0,?1) ← Person(?0), worksFor(?0,?1), Organization(?1)
	Q5	Q(?0) ← Person(?0), worksFor(?0,?1), University(?1), hasAlumnus(?1,?0)
	Q6	Q(?0,?1) ← Professor(?0), teacherOf(?0,?1), GraduateCourse(?1)
	Q7	Q(?0,?2) ← Department(?1), Professor(?2), Student(?0), memberOf(?0,?1), worksFor(?2,?1)
	Q8	Q(?0,?2) ← Student(?0), Course(?1), takesCourse(?0,?1), Professor(?2), teacherOf(?2,?1)
Vicodi	Q1	Q(?0) ← Location(?0)
	Q2	Q(?0,?1) ← Military-Person(?0), hasRole(?1,?0), related(?0,?2)
	Q3	Q(?0,?1) ← Time-Dependant-Relation(?0), hasRelationMember(?0,?1), Event(?1)
	Q4	Q(?0,?1) ← Object(?0), hasRole(?0,?1), Symbol(?1)
	Q5	Q(?0) ← Individual(?0), hasRole(?0,?1), Scientist(?1), hasRole(?0,?2), Discoverer(?2), hasRole(?0,?3), Inventor(?3)

Table 2.4: Queries for rewriting evaluation

Ontology	Query	Number of generated rules/CQs					Time (msec)			
		REQUIEM	RF	RG	Kaos		REQUIEM	RF	RG	Kaos
					Queries	Q & R				
adolena	Q1	402	27	27	13	42	226	647	226	6
adolena	Q2	103	50	50	2	31	111	162	240	5
adolena	Q3	104	104	104	1	31	236	289	376	6
adolena	Q4	492	224	224	5	39	354	628	475	6
adolena	Q5	624	624	624	1	36	750	1288	1092	3
adolena	Q6	364	364	364	1	30	334	596	512	4
adolena	Q7	2548	2548	2548	3	32	1903	7256	7547	6
adolena	Q8	936	936	936	5	38	772	2408	2561	7
path5	Q1	6	6	6	6	6	6	9	8	1
path5	Q2	10	10	10	10	10	28	37	37	2
path5	Q3	13	13	13	13	13	247	323	275	5
path5	Q4	15	15	15	15	15	1018	2173	2349	5
path5	Q5	16	16	16	16	16	6047	17064	17505	6
path8	Q1	9	9	9	9	9	8	14	10	3
path8	Q2	16	16	16	16	16	66	96	88	4
path8	Q3	22	22	22	22	22	753	1625	1900	6
path8	Q4	27	27	27	27	27	3935	13954	14290	9
path8	Q5	-	-	-	31	31	-	-	-	12
path8	Q6	-	-	-	34	34	-	-	-	19
path8	Q7	-	-	-	36	36	-	-	-	21
path8	Q8	-	-	-	37	37	-	-	-	24
stock	Q1	6	6	6	1	10	8	9	13	5
stock	Q2	160	2	2	1	25	243	452	292	3
stock	Q3	480	4	4	1	10	1119	1670	1461	2
stock	Q4	960	4	4	1	27	1069	1848	1805	3
stock	Q5	2880	8	8	1	12	5361	18472	18045	2
university	Q1	2	2	2	1	2	6	9	13	1
university	Q2	148	1	1	1	48	165	317	183	2
university	Q3	224	4	4	1	20	257	551	431	2
university	Q4	1628	2	2	1	63	790	2791	2489	1
university	Q5	2960	10	10	1	52	1821	8308	7469	2
university	Q6	10	10	10	1	10	8	12	17	3
university	Q7	320	320	320	1	16	413	672	590	2
university	Q8	160	40	40	1	16	177	414	194	1
vicodi	Q1	15	15	15	1	15	8	12	13	0
vicodi	Q2	10	10	10	1	10	9	14	16	1
vicodi	Q3	72	72	72	1	26	48	136	68	0
vicodi	Q4	185	185	185	1	41	123	339	262	0
vicodi	Q5	30	30	30	1	8	34	102	94	0

Table 2.5: Query rewriting evaluation (RF=REQUIEM-F, RG=REQUIEM-G)

Ontology	Query	Generated rules/CQs	Normalization time	Rewriting time time
adolena	Q1	42	390	6
adolena	Q2	31	390	4
adolena	Q3	31	379	5
adolena	Q4	39	388	7
adolena	Q5	36	382	4
adolena	Q6	30	401	4
adolena	Q7	32	379	6
adolena	Q8	38	382	8
path5	Q1	6	270	1
path5	Q2	10	257	4
path5	Q3	13	254	3
path5	Q4	15	255	5
path5	Q5	16	256	8
path8	Q1	9	269	3
path8	Q2	16	270	5
path8	Q3	22	271	7
path8	Q4	27	271	9
path8	Q5	31	264	12
path8	Q6	34	273	17
path8	Q7	36	272	22
path8	Q8	37	262	23
stock	Q1	10	341	4
stock	Q2	25	335	7
stock	Q3	10	340	2
stock	Q4	27	342	3
stock	Q5	12	338	3
university	Q1	2	388	2
university	Q2	48	382	2
university	Q3	20	385	2
university	Q4	63	387	2
university	Q5	52	387	1
university	Q6	10	388	2
university	Q7	16	405	3
university	Q8	16	390	3
university	Q9	6	393	3
vicodi	Q1	15	447	0
vicodi	Q2	10	436	0
vicodi	Q3	26	452	0
vicodi	Q4	41	442	1
vicodi	Q5	8	442	0

Table 2.6: Normalization time

Queries			
Q1	Q(?0)	←	worksFor(?0,?1), affiliatedOrganizationOf(?1,?2)
Q2	Q(?0,?1)	←	Person(?0), teacherOf(?0,?1), Course(?1)
Q3	Q(?0,?1,?2)	←	Student(?0), advisor(?0,?1), FacultyStaff(?1), takesCourse(?0,?2), teacherOf(?1,?2), Course(?2)
Q4	Q(?0,?1)	←	Person(?0), worksFor(?0,?1), Organization(?1)
Q5	Q(?0)	←	Person(?0), worksFor(?0,?1), University(?1), hasAlumnus(?1,?0)
Q6	Q(?0,?1)	←	Professor(?0), teacherOf(?0,?1), GraduateCourse(?1)
Q7	Q(?0,?2)	←	Department(?1), Professor(?2), Student(?0), memberOf(?0,?1), worksFor(?2,?1)
Q8	Q(?0,?2)	←	Student(?0), Course(?1), takesCourse(?0,?1), Professor(?2), teacherOf(?2,?1)
Q9	Q(?0)	←	Student(?0), advisor(?0,?1), headOf(?1,?2)
Q10	Q(?0)	←	Student(?0), advisor(?0,?1), memberOf(?0,?2), takesCourse(?0,?3), worksFor(?1,?2), teacherOf(?1,?3)
Q11	Q(?0)	←	Student(?0), memberOf(?0,?1), hasAlumnus(?1,?2), orgPublication(?1,?3)

Table 2.7: Queries used to compare with REQUIEM

Ontology	Query	Queries				Rewriting time				Datalog running time				Total time			
		RN	RF	RG	Kaos	RN	RF	RG	Kaos	RN	RF	RG	Kaos	RN	RF	RG	Kaos
lubm1	Q1	373	2	2	2	263	299	362	18	630	490	490	510	893	789	852	528
lubm1	Q2	556	324	1	94	323	592	593	14	760	580	490	730	1083	1172	1083	744
lubm1	Q3	391	327	1	94	252	568	2951	12	630	580	490	730	882	1148	3441	742
lubm1	Q4	2073	325	2	94	1417	3592	3287	12	2030	600	500	740	3447	4192	3787	752
lubm1	Q5	2591	333	10	94	1611	6075	5301	12	3320	590	500	740	4931	6665	5801	752
lubm1	Q6	380	332	11	94	303	520	463	12	650	590	520	750	953	1110	983	762
lubm1	Q7	695	615	-	95	560	1136	-	13	3500	3890	-	1390	4060	5026	-	1403
lubm1	Q8	551	359	-	94	450	677	-	11	4240	2490	-	1430	4690	3167	-	1441
lubm1	Q9	387	339	36	97	282	537	237997	13	990	930	760	830	1272	1467	238757	843
lubm1	Q10	479	359	39	99	306	919	202974	14	840	880	580	810	1146	1799	203554	824
lubm1	Q11	451	403	180	94	331	882	18523	13	680	620	510	760	1011	1502	19033	773
lubm2	Q1	373	2	2	2	263	299	362	18	1490	1200	1150	1200	1753	1499	1512	1218
lubm2	Q2	556	324	1	94	323	592	593	14	1650	1530	1160	1710	1973	2122	1753	1724
lubm2	Q3	391	327	1	94	252	568	2951	12	1440	1270	1150	1700	1692	1838	4101	1712
lubm2	Q4	2073	325	2	94	1417	3592	3287	12	3230	1310	1180	1730	4647	4902	4467	1742
lubm2	Q5	2591	333	10	94	1611	6075	5301	12	4750	1310	1150	1700	6361	7385	6451	1712
lubm2	Q6	380	332	11	94	303	520	463	12	1390	1320	1540	1950	1693	1840	2003	1962
lubm2	Q7	695	615	-	95	560	1136	-	13	8000	7800	-	2940	8560	8936	-	2953
lubm2	Q8	551	359	-	94	450	677	-	11	7510	3600	-	2120	7960	4277	-	2131
lubm2	Q9	387	339	36	97	282	537	237997	13	1440	1330	1260	1720	1722	1867	239257	1733
lubm2	Q10	479	359	39	99	306	919	202974	14	1900	1800	1410	1780	2206	2719	204384	1794
lubm2	Q11	451	403	180	94	331	882	18523	13	1690	2380	1220	1860	2021	3262	19743	1873
lubm3	Q1	373	2	2	2	263	299	362	18	2100	1690	1700	1800	2363	1989	2062	1818
lubm3	Q2	556	324	1	94	323	592	593	14	2230	2120	1730	2530	2553	2712	2323	2544
lubm3	Q3	391	327	1	94	252	568	2951	12	1940	1830	1670	2550	2192	2398	4621	2562
lubm3	Q4	2073	325	2	94	1417	3592	3287	12	4140	1880	1800	2680	5557	5472	5087	2692
lubm3	Q5	2591	333	10	94	1611	6075	5301	12	5770	1960	1940	2560	7381	8035	7241	2572
lubm3	Q6	380	332	11	94	303	520	463	12	2070	1900	1840	2530	2373	2420	2303	2542
lubm3	Q7	695	615	-	95	560	1136	-	13	11260	11350	-	4190	11820	12486	-	4203
lubm3	Q8	551	359	-	94	450	677	-	11	9300	4760	-	3040	9750	5437	-	3051
lubm3	Q9	387	339	36	97	282	537	237997	13	2030	1960	1850	2540	2312	2497	239847	2553
lubm3	Q10	479	359	39	99	306	919	202974	14	4520	4770	2540	3140	4826	5689	205514	3154
lubm3	Q11	451	403	180	94	331	882	18523	13	2030	1910	1680	2560	2361	2792	20203	2573
lubm4	Q1	373	2	2	2	263	299	362	18	2780	2410	2450	2470	3043	2709	2812	2488
lubm4	Q2	556	324	1	94	323	592	593	14	2980	2590	2390	3520	3303	3182	2983	3534
lubm4	Q3	391	327	1	94	252	568	2951	12	2770	2610	2410	3540	3022	3178	5361	3552
lubm4	Q4	2073	325	2	94	1417	3592	3287	12	5460	2650	2540	3500	6877	6242	5827	3512
lubm4	Q5	2591	333	10	94	1611	6075	5301	12	7280	2600	2410	3580	8891	8675	7711	3592
lubm4	Q6	380	332	11	94	303	520	463	12	2780	2630	2440	3520	3083	3150	2903	3532
lubm4	Q7	695	615	-	95	560	1136	-	13	15820	16030	-	5760	16380	17166	-	5773
lubm4	Q8	551	359	-	94	450	677	-	11	11130	5860	-	4040	11580	6537	-	4051
lubm4	Q9	387	339	36	97	282	537	237997	13	2760	2600	2560	3510	3042	3137	240557	3523
lubm4	Q10	479	359	39	99	306	919	202974	14	7530	7670	3900	4370	7836	8589	206874	4384
lubm4	Q11	451	403	180	94	331	882	18523	13	2860	2750	2370	3560	3191	3632	20893	3573

Table 2.8: Comparing with REQUIEM using DLV

Ontology	Query	Queries				Rewriting time				Datalog running time in Clingo				Total time			
		RN	RF	RG	Kaos	RN	RF	RG	Kaos	RN	RF	RG	Kaos	RN	RF	RG	Kaos
lubm1	Q1	373	2	2	2	263	299	362	18	1130	630	630	660	1393	929	992	678
lubm1	Q2	556	324	1	94	323	592	593	14	1590	860	670	870	1913	1452	1263	884
lubm1	Q3	391	327	1	94	252	568	2951	12	1170	910	680	860	1422	1478	3631	872
lubm1	Q4	2073	325	2	94	1417	3592	3287	12	10110	830	670	840	11527	4422	3957	852
lubm1	Q5	2591	333	10	94	1611	6075	5301	12	4620	840	680	850	6231	6915	5981	862
lubm1	Q6	380	332	11	94	303	520	463	12	930	850	660	900	1233	1370	1123	912
lubm1	Q7	695	615	-	95	560	1136	-	13	23840	23620	-	1150	24400	24756	-	1163
lubm1	Q8	551	359	-	94	450	677	-	11	110250	8800	-	1100	110700	9477	-	1111
lubm1	Q9	387	339	36	97	282	537	237997	13	990	860	1280	850	1272	1397	239277	863
lubm1	Q10	479	359	39	99	306	919	202974	14	3570	2560	1480	1230	3876	3479	204454	1244
lubm1	Q11	451	403	180	94	331	882	18523	13	1030	1000	2070	850	1361	1882	20593	863
lubm2	Q1	373	2	2	2	263	299	362	18	1970	1480	1390	1660	2233	1779	1752	1678
lubm2	Q2	556	324	1	94	323	592	593	14	3190	1680	1610	1790	3513	2272	2203	1804
lubm2	Q3	391	327	1	94	252	568	2951	12	2310	1760	1440	1830	2562	2328	4391	1842
lubm2	Q4	2073	325	2	94	1417	3592	3287	12	38960	2580	1440	1970	40377	6172	4727	1982
lubm2	Q5	2591	333	10	94	1611	6075	5301	12	11460	1740	1490	1760	13071	7815	6791	1772
lubm2	Q6	380	332	11	94	303	520	463	12	1860	1700	1420	1800	2163	2220	1883	1812
lubm2	Q7	695	615	-	95	560	1136	-	13	108640	109840	-	2290	109200	110976	-	2303
lubm2	Q8	551	359	-	94	450	677	-	11	-	32370	-	2080	-	33047	-	2091
lubm2	Q9	387	339	36	97	282	537	237997	13	1880	1730	4730	1780	2162	2267	242727	1793
lubm2	Q10	479	359	39	99	306	919	202974	14	11900	10820	3890	3670	12206	11739	206864	3684
lubm2	Q11	451	403	180	94	331	882	18523	13	1980	1810	3950	1760	2311	2692	22473	1773
lubm3	Q1	373	2	2	2	263	299	362	18	2510	1970	1950	1940	2773	2269	2312	1958
lubm3	Q2	556	324	1	94	323	592	593	14	4210	2360	1940	2500	4533	2952	2533	2514
lubm3	Q3	391	327	1	94	252	568	2951	12	3130	2470	2060	2510	3382	3038	5011	2522
lubm3	Q4	2073	325	2	94	1417	3592	3287	12	73660	2340	1930	2490	75077	5932	5217	2502
lubm3	Q5	2591	333	10	94	1611	6075	5301	12	17140	2410	1990	2470	18751	8485	7291	2482
lubm3	Q6	380	332	11	94	303	520	463	12	2560	2360	1980	2510	2863	2880	2443	2522
lubm3	Q7	695	615	-	95	560	1136	-	13	201180	197570	-	3700	201740	198706	-	3713
lubm3	Q8	551	359	-	94	450	677	-	11	-	50200	-	2800	-	50877	-	2811
lubm3	Q9	387	339	36	97	282	537	237997	13	2670	2400	9940	2500	2952	2937	247937	2513
lubm3	Q10	479	359	39	99	306	919	202974	14	20190	18420	6290	6390	20496	19339	209264	6404
lubm3	Q11	451	403	180	94	331	882	18523	13	2820	2620	5000	2470	3151	3502	23523	2483
lubm4	Q1	373	2	2	2	263	299	362	18	3390	2660	2630	2720	3653	2959	2992	2738
lubm4	Q2	556	324	1	94	323	592	593	14	5950	3300	6320	3730	6273	3892	6913	3744
lubm4	Q3	391	327	1	94	252	568	2951	12	4160	3460	2790	3490	4412	4028	5741	3502
lubm4	Q4	2073	325	2	94	1417	3592	3287	12	137120	3160	2700	3420	138537	6752	5987	3432
lubm4	Q5	2591	333	10	94	1611	6075	5301	12	27440	3260	2810	3390	29051	9335	8111	3402
lubm4	Q6	380	332	11	94	303	520	463	12	3450	6230	2670	3400	3753	6750	3133	3412
lubm4	Q7	695	615	-	95	560	1136	-	13	-	-	-	4030	-	-	-	4043
lubm4	Q8	551	359	-	94	450	677	-	11	-	69400	-	3720	-	70077	-	3731
lubm4	Q9	387	339	36	97	282	537	237997	13	3520	3230	20740	3460	3802	3767	258737	3473
lubm4	Q10	479	359	39	99	306	919	202974	14	32420	28870	9990	10520	32726	29789	212964	10534
lubm4	Q11	451	403	180	94	331	882	18523	13	4000	3660	6930	3380	4331	4542	25453	3393

Table 2.9: Comparing with REQUIEM using Clingo

2.7.2 Tables for scalability test

Queries			
Q1	q(X_1)	$\leftarrow$	worksFor(X_1, X_2), affiliatedOrganizationOf(X_2, X_3)
Q2	q(X_1, X_3)	$\leftarrow$	LeisureStudent(X_1), takesCourse(X_1,X_2), isTaughBy(X_2,X_3), SportsLover(X_3)
Q3	q(X_0, X_1)	$\leftarrow$	enrollIn(X_0,X_1), hasDegreeFrom(X_0, X_1)
Q4	q(X_1, X_3)	$\leftarrow$	Student(X_1), hasDegreeFrom(X_1, X_2), Professor(X_3), worksFor(X_3 ,X_2)
Q5	q(X_1)	$\leftarrow$	Postdoc(X_1), worksFor(X_1, X_2), University(X_2), hasAlumnus(X_2 ,X_1)
Q6	q(X_1)	$\leftarrow$	Person(X_1), like(X_1, X_2), Chair(X_3), isHeadOf(X_3,X_4), like(X_3,X_2)
Q7	q(X_1,X_2)	$\leftarrow$	Postdoc(X_1), hasAlumnus(X_2, X_1)
Q8	q(X_1,X_2)	$\leftarrow$	GraduateCourse(X_1), isTaughtBy(X_1, X_2), isHeadOf(X_2,X_3)
Q9	q(X_1)	$\leftarrow$	PeopleWithManyHobbies(X_1), isMemberOf(X_1, X_2)
Q10	q(X_1)	$\leftarrow$	SportsLover(X_1), isHeadOf(X_1, X_2), ResearchGroup(X_2)

Table 2.10: Queries to run with UOBM-Horn-$\mathcal{SHIQ}$

Ontology	Query	Generated queries/rules	Normalization time	Rewriting time	Datalog running time in DLV
uobm-hornshiq	Q1	2	627	59	260
uobm-hornshiq	Q2	3	609	53	220
uobm-hornshiq	Q3	9	616	67	230
uobm-hornshiq	Q4	175	597	53	620
uobm-hornshiq	Q5	17	596	53	250
uobm-hornshiq	Q6	258	597	73	820
uobm-hornshiq	Q7	8	606	55	250
uobm-hornshiq	Q8	178	613	56	690
uobm-hornshiq	Q9	181	596	54	740
uobm-hornshiq	Q10	3	612	54	240

Table 2.11: Experiment with Horn-$\mathcal{SHIQ}$ ontology

Chapter 3

Optimal Reasoning with Forest Logic Programs

During the previous two years of the project we studied a decidable fragment of Open Answer Set Programming (OASP) called Forest Logic Programs (FoLPs). The integrating formalism is called *f-hybrid* knowledge bases. One salient feature of OASP and thus also of FoLPs, is that while its syntax is typical ASP syntax (albeit allowing for a particular type of unsafe rules, called *free rules*), its semantics is a hybrid between the classical Answer Set Programming semantics and the classical First Order Logics semantics. From the ASP world it retains a stable model based semantics, while from the FOL world it retains the possibility of having an infinite domain of interpretation: the universe is a non-empty superset of the set of constants in the program. The open domain semantics together with the presence of unsafe rules, make it possible to simulate within FoLPs reasoning with the expressive DL $\mathcal{SHOQ}$: as such FoLPs serve as an integrative device for *f-hybrid knowledge bases*, a tightly-coupled combination of FoLPs themselves and $\mathcal{SHOQ}$ KBs [42].

Example 3.0.1. *Consider the following program:*

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

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 $fail$.

The fact that the universe of interpretation is not restricted to the Herbrand universe makes it possible to simulate within the formalism, General Inclusion Axioms with an *exists restriction* on their right-hand side: this is a feature which was identified as desirable during the analysis of the ONTORULE Use Cases. For a discussion regarding this feature please consult the analysis of the Steel Industry Use Case in the Appendix B of deliverable D3.3 [40].

FoLPs allow for the presence of only unary and binary predicates in rules which have a tree-like structure. This makes the fragment decidable by ensuring that it has the forest model property: if a unary predicate is satisfiable, then it is satisfied by a forest-shaped model. A forest shaped model is a model in which the universe of interpretation can be seen as a forest, two nodes in the forest being connected iff there is a binary atom in the model having as arguments the respective nodes.

A sound and complete algorithm for satisfiability checking of unary predicates w.r.t. FoLPs has been described in deliverable D3.2 [51]. The algorithm exploits the forest model property of the fragment: it is essentially a tableau-based procedure which builds such a forest model in a top-down fashion. It starts with a skeleton for a forest model which contains only one constraint: p, the unary predicate checked to be satisfiable, has to appear in the label of a node in the forest. Then, in order to satisfy existing constraints, it progressively introduces new ones by inserting (negated) predicates in the contents of nodes/arcs of the forest based on the rules of the program. The forest model is constructed by evolving a data structure called "completion structure" which contains a labeled forest, the model in construction, together with additional information needed for the construction. When certain conditions are met, like either there are no unsatisfied constraints, or the ones left can be satisfied similarly to previously met constraints, the algorithm terminates successfully. We will refer to this algorithm as $\mathcal{A}_1$. The algorithm is also described in [39] and in [42]

During the second year of the project we devised an optimization of the first algorithm by means of a knowledge compilation technique: the new algorithm which is described in deliverable D3.3 [40] computes in an initial step all possible building blocks of the model using $\mathcal{A}_1$ and then matches and appends these blocks using similar conditions for termination as the original algorithm. Such building blocks are restricted to completion structures, in which there is only one node fully expanded (i.e. all constraints associated to that node are satisfied), and are called *unit completion structures*. We will refer to this algorithm as $\mathcal{A}_2$. The algorithm has also been described in [41].

Both $\mathcal{A}_1$ and $\mathcal{A}_2$ run in the worst case in double exponential time. During this last year of the project we developed an algorithm with improved running time by dropping the worst case complexity one exponential level: it runs in the worst case in exponential time in the size of the input program. This also settles a gap concerning the complexity of FoLPs: it was known that FoLPs are EXPTIME-hard, but not known whether they are EXPTIME-complete. The new algorithm shows that they are indeed EXPTIME-complete. We refer to the new algorithm as $\mathcal{A}_3$.

$\mathcal{A}_3$ takes over the idea of using unit complete structures from $\mathcal{A}_2$. Constraints regarding contents of nodes are satisfied by finding appropriate unit completion structures and appending them. However, unlike $\mathcal{A}_2$, it employs different termination techniques. In particular it employs a new technique for identifying redundancy across a path and a caching technique.

The section is organized as follows: Section 3.1 introduces some technical preliminaries. Section 3.2 introduces formally Forest Logic Programs and the notions of forest model and forest satisfiability. Section 3.3 describes a simplified version of $\mathcal{A}_2$, while Section 3.4 describes the new algorithm. Finally, Section 3.5 draws some conclusions and discusses future work and Section 3.6 provides a list with the main notions used by the algorithms described in this chapter.

3.1 Preliminaries

We recall the open answer set semantics [50]. *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 regular literal L, $pred(L)$, and $args(L)$ denote the predicate, and the (tuple of) arguments of L[1], respectively. By $args_i(L)$, for a regular literal L, we understand the i-th argument of L.

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. Rules can also be named, as in $r : \alpha \leftarrow \beta$, where r is the name of the rule. 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 $t_1, \ldots, t_n$; this kind of rules enables a choice for the inclusion of atoms in the open answer sets. We call a predicate q *free* if there is a free rule: $q(X_1, \ldots, X_n) \vee not\ q(X_1, \ldots, X_n) \leftarrow$, 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 $\mathcal{B}_P$ ($\mathcal{L}_P$) be the set of regular atoms (literals) that can be formed from a ground program P.

[1]If the literal L has just one argument, $args(L)$ will return the argument itself.

For a term t, the *exact replacement* of ground term x with ground term y in t, denoted $t_{x|y}$, is defined as follows: $t_{x|y}=\begin{cases} y, & \text{if } t = x; \\ t, & \text{otherwise} \end{cases}$. The notation extends to tuples of terms, literals, rules, and programs. For a tuple of terms $T = (t_1, \ldots, t_n)$, $T_{x|y} = ((t_1)_{x|y}, \ldots, (t_n)_{x|y})$. For a regular literal $L = (not\ )p(t_1, \ldots, t_n)$, $L_{x|y} = (not\ )$ $p((t_1)_{x|y}, \ldots, (t_n)_{x|y})$. For a set of literals S, $S_{x|y} = \{L_{x|y} \mid L \in S\}$. For a named rule $r : \alpha \leftarrow \beta$, its image under the exact replacement of x with y is $r_{x|y} : \alpha_{x|y} \leftarrow \beta_{x|y}$ (where $r_{x|y}$ is the new name of the rule, and does not involve any term replacement). For a ground program P, its image under the exact replacement of x with y is $P_{x|y} = \{r_{x|y} \mid r \in P\}$.

An *interpretation* I of a ground program P is a subset of $\mathcal{B}_P$. 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* [45] 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 [97]. Let $\cdot$ be a concatenation operator between sequences of constants or natural numbers. A *tree* T with root c (also denoted as T_c), where c is a specially designated constant, is a set of nodes, where each node is a sequence of the form $c \cdot s$, where s is a (possibly empty) sequence of positive integers formed with the help of the concatenation operator (we denote the set of all such sequences with $\langle \mathbb{N}^* \rangle$, where $\mathbb{N}^*$ is the set of positive integers); for $x \cdot d \in T$, $d \in \mathbb{N}^*$, we must have that $x \in T$. For example a tree with root c and 2 successors will be denoted as $\{c, c \cdot 1, c \cdot 2\}$ or $\{c, c1, c2\}$ [2]. 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 $x \neq y$. The predecessor of a node x in a tree T is denoted with $prev_T(x)$ and it is the node y such that there exists $\imath \in \mathbb{N}^*$ such that $x = y \cdot i$. The deepest common ancestor of two nodes x and y in a tree T, denoted $common_T(x, y)$ is the node z such that $z <_T x$, $z <_T y$, and there is no node $z' \in T$ such that $z' >_T z$, $z' <_T x$, and $z' <_T y$. A node $x \in T$ is said to be to the right of a node $y \in T$ and denoted with $right_T(x, y)$ iff there exists a node

[2]By abuse of notation, we consider that there are at most 9 successors for every node, so we can abbreviate $a \cdot b$ with ab

$z \in T$, $i, j \in \mathbb{N}^*$, and $s_1, s_2 \in \langle\mathbb{N}^*\rangle$, such that $x = z \cdot i \cdot s_1$, $y = z \cdot j \cdot s_2$, and $i > j$. The subtree of T_c at y, denoted $T_c[y]$, is the set $\{x \mid x \in T_c, x = y \cdot s, s \in \langle\mathbb{N}^*\rangle\}$. A path in a tree T from x to y is denoted with $path_T(x, y) = \{z \mid x \leqslant z \leqslant y\}$.

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.

In the following, all terms in ground programs which we operate with are nodes in some extended forest, and as such they are sequences formed with the help of the $\cdot$ operator. Taking into account the structure of such terms, we introduce a finer grain (ground) term replacement operator which replaces the prefix of a term with another term. This is simply called *replacement* of x with y in t, it is denoted with $t_{x||y}$, and it is defined as $t_{x||y} = \begin{cases} y \cdot z, & \text{if } t = x \cdot z; \\ t, & \text{otherwise} \end{cases}$. Similarly as for the exact replacement, the notion of replacement is extended to (sets of) literals, tuples, rules, and programs.

When an extended forest $EF = (F, ES)$, is such that F is a set of trees $\{T_c \mid c \in C\}$, for C a set of distinguished constants, and there exists $d \in C$ such that $T_c = \{c\}$, for every $c \in C \backslash \{d\}$, and $ES \subseteq T_d \times C$, we call the forest an *extended tree with root* d *w.r.t.* C: all trees but one are single-node trees and the nodes of the *distinguished tree* T_d can be interlinked with constants from C; no other links from elements of C are allowed. The depth of an extended tree is the depth of its distinguished tree.

Finally, a directed graph G is defined as usual by its sets of nodes V and arcs A. We introduce some graph-related notations: $paths_G$ denotes the set of paths in G, where each path is a tuple of nodes from V: $paths_G = \{(x_1, \ldots, x_n) \mid ((x_i, x_{i+1}) \in A)_{1 \leqslant i < n}\}$, $paths_G(x, y)$ denotes the set of paths in G from x to y: $paths_G(x, y) = \{(x_1 = x, \ldots, x_n = y) \mid ((x_i, x_{i+1}) \in A)_{1 \leqslant i < n}\}$, while $conn_G$ denotes the set of pairs of connected nodes from V: $conn_G = \{(x, y) \mid \exists Pt = (x_1, \ldots, x_n) \in paths_G : x_1 = x \wedge x_n = y\}$. As an extended forest is a particular type of graph, these notations apply also to extended forests. Cycles and elementary cycles in directed graphs are defined as usually. In order to operate with paths in directed graphs we also introduce some tuple operators: the concatenation of two tuples $T_1 = (x_1, \ldots, x_n)$, and $T_2 = (y_1, \ldots, y_m)$, denoted $T_1 \hat{} T_2$ is the tuple $(x_1, \ldots, x_n, y_1, \ldots, y_m)$. A tuple T_1 is part of another tuple T_2: $T_1 \subseteq T_2$, if there exists two (possibly empty) tuples T_3 and T_4 such that $T_2 = T_3 \hat{} T_1 \hat{} T_4$.

3.2 FoLPs

Forest Logic Programs are a subset of Open Answer Set Programming (OASP) which allows one to simulate the DL $\mathcal{SHOQ}$, underpinning the tightly-coupled combination of rules and ontologies: *f-hybrid* knowledge bases.

Definition 3.2.1. *A* forest logic program (FoLP) *is a program with only unary and binary predicates, and such that a rule is either:*

- *a* free rule*:*
$$a(s) \vee not\ a(s) \leftarrow \text{ or } f(s,t) \vee not\ f(s,t) \leftarrow \tag{3.1}$$
where s and t are terms;

- *a unary rule:*
$$a(s) \leftarrow \beta(s), (\gamma_m(s,t_m), \delta_m(t_m))_{1 \leqslant m \leqslant k}, \psi \tag{3.2}$$
with $\psi \subseteq \bigcup_{1 \leqslant i \neq j \leqslant k} \{t_i \neq t_j\}$ and $k \in \mathbb{N}$, or a binary rule*:*
$$f(s,t) \leftarrow \beta(s), \gamma(s,t), \delta(t) \tag{3.3}$$
where $a \in upreds(P)$ and $f \in bpreds(P)$, s, t, and $(t_m)_{1 \leqslant m \leqslant k}$ are terms, β, δ, $(\delta_m)_{1 \leqslant m \leqslant k} \subseteq upreds(P) \cup not\ (upreds(P))$ (sets of (possibly negated) unary predicates), γ, $(\gamma_m)_{1 \leqslant m \leqslant k} \subseteq bpreds(P) \cup not\ (bpreds(P))$ (sets of (possibly negated) binary predicates), and

  1. *inequality does not appear in any γ: $\{\neq\} \cap \gamma_m = \emptyset$, for $1 \leqslant m \leqslant k$, and $\{\neq\} \cap \gamma = \emptyset$;*
  2. *there is a positive atom that connects the head term s with any successor term which is a variable: $\gamma_m^+ \neq \emptyset$, if t_m is a variable, for $1 \leqslant m \leqslant k$, and $\gamma^+ \neq \emptyset$, if t is a variable;*

- *a constraint: $\leftarrow a(s)$ or $\leftarrow f(s,t)$, where s and t are terms.*

In every rule, all terms which are variables are distinct[3].

Example 3.2.2. *The following program*[4] *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* supportedBy *which describes the 'has support' relationship is free:* rule r_3*. No individual can be at the same time both a special member and a regular member:* constraint r_4*. Somebody is a regular member if it is involved in*

[3]This restriction precludes the presence in rules of literals of the form $f(X,X)$ or $not\ f(X,X)$ which would break the forest model property.

[4]The example is a variation of an example introduced in deliverable D3.3 [40].

some project: rule r_5. *The binary predicate* involvedIn *which describes the 'involved in a project' relationship is free:* rule r_6. *There is a project* j*:* fact r_7.

$$\begin{array}{lrcl}
r_1: & smember(X) & \leftarrow & supportedBy(X,Y), smember(Y) \\
r_2: & smember(X) & \leftarrow & supportedBy(X,Y), rmember(Y), \\
 & & & supportedBy(X,Z), rmember(Z), \\
 & & & Y \neq Z \\
r_3: & supportedBy(X,Y) \lor not\ supportedBy(X,Y) & \leftarrow & \\
r_4: & & \leftarrow & smember(X), rmember(X) \\
r_5: & rmember(X) & \leftarrow & involvedIn(X,Y), project(Y) \\
r_6: & involvedIn(X,Y) \lor not\ involvedIn(X,Y) & \leftarrow & \\
r_7: & project(j) & \leftarrow &
\end{array}$$

As already mentioned, FoLPs have the *forest model property*.

Definition 3.2.3. *Let* P *be a program. 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[5]*, and a labeling function* $\mathcal{L} : \{T_\varepsilon\} \cup \{T_a \mid a \in cts(P)\} \cup A_{EF} \to 2^{preds(P)}$ *such that*

- $p \in \mathcal{L}(\varepsilon)$*,*
- $U = N_{EF}$*, and*
- $M = \{\mathcal{L}(x)(x) \mid x \in N_{EF}\} \cup \{\mathcal{L}(x,y)(x,y) \mid (x,y) \in A_{EF}\}$*, and*
- *for every* $(z, z \cdot i) \in A_{EF}$*:* $\mathcal{L}(z, z \cdot i)^+ \neq \emptyset$*.*

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*.*

Proposition 3.2.4 ([49])**.** *FoLPs have the forest model property.*

Example 3.2.5. *Consider the FoLP* P *introduced in Example 3.2.2.*

The unary predicate $smember$ *is forest satisfiable w.r.t.* P*: there is a forest model* $(\{j, x, y, z, t\}, \{smember(x), supportedBy(x,y), smember(y), rmember(z), rmember(t), supportedBy(y,z), supportedBy(y,t), involvedIn(z,j), involvedIn(t,j), project(j)\})$ *in which* $smember$ *appears in the label of the (anonymous) root of one of the trees in the forest (see Figure 3.1).*

[5]Note that in this case $T_\varepsilon \in \{T_a \mid a \in cts(P)\}$. Thus, the extended forest contains for every constant from P a tree which has as root that specific constant and possibly, but not necessarily, an extra tree with unidentified root node.

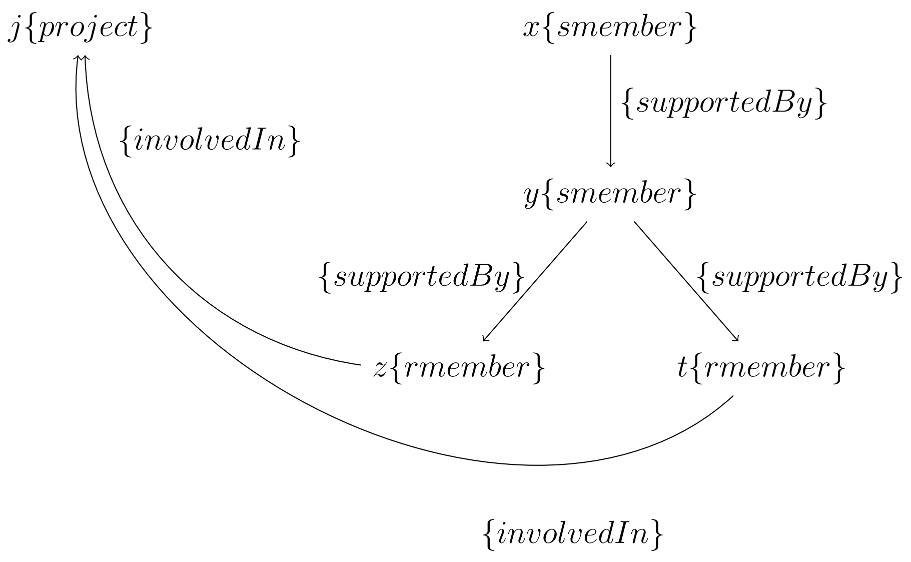


Figure 3.1: A forest model for P.

3.3 Previous Algorithm for Reasoning with FoLPs using Unit Completion Structures

As mentioned in the introduction, all algorithms we developed for reasoning with FoLPs have the same underlying principle: they try to construct a forest model in a tableau-like fashion. All algorithms share the same data structure, called *completion structure*, which is a representation of a forest model in construction. In section 3.3.1 we describe this data structure and recall how it can be evolved using so-called expansion rules introduced in Deliverable D3.2 [51]. These expansion rules are used by $\mathcal{A}_2$, the algorithm developed during the second year of the project and described in Deliverable D3.3 [40], to construct the set of unit completion structures. The new version of the algorithm, $\mathcal{A}_3$ reuses the knowledge compilation technique introduced by $\mathcal{A}_2$. As such, section 3.3.2 recalls $\mathcal{A}_2$.

3.3.1 Completion Structures

The main data structure used by all three algorithms is a so-called *completion structure*. A completion structure describes a forest model in construction. It contains an extended forest EF, whose set of nodes constitutes the universe of the model in construction, 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. Note that unlike the labeling function $\mathcal{L}$ in Definition 3.2.3, that describes which atoms are in the forest model, the labeling function $\textsc{ct}$ keeps track also of which atoms are not in the forest model. This is needed as the completion structure is updated by justifying both the presence or the absence of a certain atom in the model.

There is a difference in how a completion structure is updated by $\mathcal{A}_1$ as opposed to how it is updated by $\mathcal{A}_2$ and $\mathcal{A}_3$. The original algorithm $\mathcal{A}_1$ updates a completion structure by means of so-called expansion rules which justify or assert the presence/absence in the model of one atom at a time. The 'local status' function $\textsc{lst}$ assigns the value $unexp$ to pairs of nodes/arcs and possibly negated unary/binary predicates which have not yet been 'expanded', i.e. justified, and the value exp to such pairs which have already been considered. However, $\mathcal{A}_2$ and $\mathcal{A}_3$, update the structure by considering one node at a time and trying to satisfy all constraints imposed by that node in a single step. So, in this case, the local status function has been replaced by a 'status' function $\textsc{st}$ which assigns one of the values exp or $unexp$ to nodes of the forest, depending whether their content has been justified or not. Based on this difference concerning the status function, we distinguish between $\mathcal{A}_1$- and $\mathcal{A}_2$- completion structures.

Furthermore, all algorithms have to ensure that the constructed forest model is a well-supported one [37], or in other words, no atom in the model is circularly justified (does not depend on itself) or infinitely justified (does not depend on an infinite chain of other atoms). As such, a graph G which keeps track of dependencies between atoms in the model is maintained both by a $\mathcal{A}_1$- and a $\mathcal{A}_2$- completion structure. The formal definition is given below.

Definition 3.3.1. *An $\mathcal{A}_1$-/$\mathcal{A}_2$- completion structure for a FoLP P is a tuple $\langle EF, \textsc{ct}, \textsc{lst}/\textsc{st}, G\rangle$ where:*

- *$EF = \langle F, ES\rangle$ is an extended forest,*
- *$\textsc{ct} : N_{EF} \cup A_{EF} \to 2^{preds(P) \cup not\ (preds(P))}$ is the 'content' function,*
- *$\textsc{lst} : N_{EF} \times 2^{upreds(P) \cup not\ upreds(P)} \cup A_{EF} \times 2^{bpreds(P) \cup not\ bpreds(P)} \to \{exp, unexp\}$/$\textsc{st} : N_{EF} \to \{exp, unexp\}$ is the 'local status'/'status' function,*
- *$G = \langle V, A\rangle$ is a directed graph which has as vertices atoms in the answer set in construction: $V \subseteq \mathcal{B}_{P_{N_{EF}}}$.*

$\mathcal{A}_1$-/$\mathcal{A}_2$- completion structures are constructed by starting with a skeleton for a model of a FoLP P which satisfies a unary predicate p, which is called $\mathcal{A}_1$-/$\mathcal{A}_2$- *initial completion structure for checking satisfiability of a unary predicate p w.r.t. P* and then progressively updating such a structure.

An $\mathcal{A}_1$-/$\mathcal{A}_2$- initial completion structure for checking satisfiability of a unary predicate p w.r.t. a FoLP P imposes a single constraint on the model in construction: that some atom $p(\varepsilon)$ has to be part of the model, where ε is an anonymous individual or one of the constants in the program. As every model of P has as part of its universe the set $cts(P)$, the extended forest EF is initialized with a set of single-node trees, one tree for each constant appearing in P (having the respective constant as a root) and possibly a new single-node tree with an anonymous root (in case ε, the node where p is asserted to be satisfied, is anonymous)[6]. The content of ε is initialized with $\{p\}$, while the contents of the other nodes (roots) are initialized with $\emptyset$. G is initialized to the graph with a single vertex $p(\varepsilon)$. Formally:

Definition 3.3.2. *An* $\mathcal{A}_1$-/$\mathcal{A}_2$- initial completion structure for checking satisfiability of a unary predicate p w.r.t. a FoLP P *is a completion structure* $\langle EF, \text{CT}, \text{ST}, G\rangle$*, where:*

- $EF = (F, \emptyset)$, $F = \{T_\varepsilon\} \cup \{T_a \mid a \in cts(P)\}$*, where* ε *is a constant, possibly in* $cts(P)$*,*
- $T_x = \{x\}$*, for* $x \in \{\varepsilon\} \cup cts(P)$*,*
- $\text{LST}(\varepsilon, p) = unexp$*/*$\text{ST}(x) = unexp$*, for* $x \in \{\varepsilon\} \cup cts(P)$*,*
- $G = \langle V, \emptyset\rangle$, $V = \{p(\varepsilon)\}$*, and*
- $\text{CT}(\varepsilon) = \{p\}$.

Note that the extended forest EF in an $\mathcal{A}_1$-/$\mathcal{A}_2$- *initial completion structure for checking satisfiability of a unary predicate* p *w.r.t. a FoLP* P is an extended tree.

The original algorithm, $\mathcal{A}_1$, expands an $\mathcal{A}_1$-completion structure by means of so-called expansion rules which justify or assert the presence/absence in the model of one atom at a time. Expansion rules satisfy current constraints in the structure, or in other words, they justify the presence/absence of certain atoms in the constructed model, by making true the body of a ground rule which has the atom in the head (in case the atom is in the model) or making false all bodies of ground rules which have the atom in the head (in case the atom is not in the model). Concretely, expansion rules may introduce new successors for the node under consideration in order to obtain successful groundings for unary rules, and may assert predicate symbols or their negation to the contents of nodes/arcs in the completion structure in order to make the body of the corresponding ground rule satisfiable/unsatisfiable. The expansion rules which take care of this are called *expand unary/binary positive/negative* rules and they are formally described in deliverable D3.2 [51] as the expansion rules (i)-(ii) and (iv)-(v).

Newly introduced domain elements give rise to new ground atoms and rules and some of these rules might render the program inconsistent. In order to be sure that the partially constructed model is a complete one every ground atom which can be formed with unary/binary predicates from the program and nodes/arcs in the forest model in construction has to be proved to be either part or not part of the forest model. As such, if for a unary/binary atom, neither the atom nor its negation appear in the content of some

[6]Note that this complies with the generic shape of a forest model described in section 3.2.

node/arc in the forest, either the unary/binary atom or its negation is inserted in the content of such a node/arc. The expansion rules which take care of this are called *choose unary/binary* rules and they are formally described in deliverable D3.2 [51] as the expansion rules (iii) and (vi).

For examples concerning the application of the expansion rule please consult deliverable D3.2 [51].

Before describing the knowledge compilation method introduced in Deliverable D3.3 [40] we recall one more notation introduced in D3.2 (as part of the applicability rule (vii) Saturation): a node $x \in N_{EF}$ is said to be *saturated* if every unary predicate or its negation appear in its content with status exp and every binary predicate or its negation appear in the content of each of its outgoing arcs with status exp.

3.3.2 Unit Completion Structures

A unit completion structure (UCS) is a completion structure in which EF is an extended tree of depth 1 having as roots of the trees the constants in the program and possibly an additional node standing for an anonymous individual: if there is such an anonymous individual, it is the root of the distinguished tree in the extended tree. The root of the distinguished tree, ε, is saturated: every unary predicate appears (either in a positive or a negated form) in $\textsc{ct}(\varepsilon)$ and every binary predicate appears (either positive or negated) in the content of every outgoing arc of ε. As the distinguished tree has depth 1, we call the nodes which are direct successors of ε in EF simply *successor nodes* (in the UCS). Note that successor nodes can be either anonymous individuals or constants from the program.

Unit completion structures can be used as building blocks of a forest model. A UCS describes how the literals formed with the (possibly negated) unary/binary predicates in the content of ε and its outgoing arcs are justified by the presence of some other (possibly negated) predicates in the contents of the nodes/arcs of the structure. No predicate in the contents of successor nodes is expanded. At an abstract level a UCS captures the process of justifying the constraints imposed by a node in a completion structure by introducing new constraints in the form of successors of that node.

3.3.2.1 Constructing the Set of Unit Completion Structures

In order to construct a unit completion structure one starts with a skeleton, an *initial unit completion structure* which is similar to an $\mathcal{A}_1$-initial completion structure for checking the satisfiability of a unary predicate p w.r.t. a FoLP P. An initial unit completion structure has the same extended tree skeleton like an $\mathcal{A}_1$-initial completion structure, but it does not impose any constraints regarding membership of predicates to nodes/arcs. This is because UCSs have to be more generic if they are to be reused as building blocks of the model. It also employs a 'local status function' as a UCS is constructed from an initial UCS by using the expansion rules (i)-(vi) introduced in Deliverable D3.2 [51]. So, an initial unit completion structure is actually an $\mathcal{A}_1$-completion structure.

Definition 3.3.3. *An* initial unit completion structure with root ε *for a FoLP P is an $\mathcal{A}_1$-completion structure $\langle EF, \text{CT}, \text{ST}, G\rangle$ where:*

- *$EF = (F, ES)$, $F = \{T_\varepsilon\} \cup \{T_a \mid a \in cts(P)\}$, where ε is a constant, possibly in $cts(P)$, and $ES = \emptyset$,*
- *$T_x = \{x\}$ for every $x \in \{\varepsilon\} \cup cts(P)$,*
- *$\text{CT}(x) = \emptyset$, for every $x \in \{\varepsilon\} \cup cts(P)$,*
- *$G = \langle V, A\rangle$, $V = \emptyset$, $A = \emptyset$.*

Next we specify when a unit completion structure is 'fully' expanded.

Definition 3.3.4. *A* unit completion structure $\langle EF, \text{CT}, \text{LST}, G\rangle$ *with root ε for a FoLP P, with $EF = (\{T_\varepsilon\}, ES)$, is an $\mathcal{A}_1$-completion structure derived from an initial unit completion structure with root ε for P by application of the expansion rules (i)-(vi) described in Deliverable D3.2 Section [51], which has the following properties:*

- *for all $p \in upreds(P)$, either $p \in \text{CT}(\varepsilon)$ and $\text{LST}(p, \varepsilon) = exp$, or $not\ p \in \text{CT}(\varepsilon)$ and $\text{LST}(not\ p, \varepsilon) = exp$;*
- *for all $c \in \mathbb{N}^*$ s.t. $\varepsilon \cdot c \in T$, and for all $f \in bpreds(P)$, either $f \in \text{CT}(\varepsilon, \varepsilon \cdot c)$ and $\text{LST}(p, (\varepsilon, \varepsilon \cdot c)) = exp$, or $not\ f \in \text{CT}(\varepsilon, \varepsilon \cdot c)$ and $\text{LST}(not\ p, (\varepsilon, \varepsilon \cdot c)) = exp$;*
- *for all $c \in cts(P)$ s.t. $(\varepsilon, c) \in ES$ and for all $f \in bpreds(P)$ either $f \in \text{CT}(\varepsilon, c)$ and $\text{LST}(p, (\varepsilon, c)) = exp$, or $not\ f \in \text{CT}(\varepsilon, c)$ and $\text{LST}(not\ p, (\varepsilon, c)) = exp$;*
- *for all c s.t. $\varepsilon \cdot c \in T$ and for all $\pm p \in \text{CT}(\varepsilon \cdot c)$, $\text{LST}(\pm p, \varepsilon \cdot c) = unexp$;*
- *for all c s.t. $(\varepsilon, c) \in ES$ and for all $\pm p \in \text{CT}(c)$, $\text{LST}(\pm p, c) = unexp$.*

For examples of unit completion structures for a FoLP P we point the reader to deliverable D3.3 [40].

Proposition 3.3.5. *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. 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. □

Once a unit completion structure is constructed, the local status function is no longer relevant. As such, from now on we will refer to unit completion structures as triples $\langle EF, \text{CT}, G\rangle$, leaving the local status function apart.

3.3.2.2 Using Unit Completion Structures

As mentioned previously, in a unit completion structure, the contents of the root and of the arcs are fully justified while no constraint associated with one of the successor nodes is satisfied. An $\mathcal{A}_2$-completion structure is evolved by starting with an $\mathcal{A}_2$-initial completion structure and repeatedly appending new unit completion structures to the structure such that every new added UCS justifies the constraints imposed by some unexpanded node in the structure. Leaf nodes of the completion structure in construction (successor nodes of previously added UCS) are matched with new UCS-s and are eventually replaced by these. The notion of matching will be made clear later.

We introduce next the notion of *local satisfiability* for a unit completion structure.

Definition 3.3.6. *A unit completion structure UC for a FoLP P with root ε* locally satisfies *a (possibly negated) unary predicate p iff $p \in \text{CT}(\varepsilon)$. Similarly, UC locally satisfies a set S of (possibly) negated unary predicates iff $S \subseteq \text{CT}(\varepsilon)$.*

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.

A node of a completion structure can be matched with a unit completion structure if the unit completion structure locally satisfies the content of the node and the constraints imposed by the UCS on nodes which are constants from P are not in contradiction with the current contents of those nodes.

Definition 3.3.7. *Let $CS = \langle EF, \text{CT}, \text{ST}, G\rangle$ be an $\mathcal{A}_2$-completion structure. A node $x \in N_{EF}$ is* matchable *with a unit completion structure $UC = \langle EF', \text{CT}', G'\rangle$ with root ε, with $EF' = (F', ES')$, iff:*

- $\text{ST}(x) = unexp$,
- $x = \varepsilon$, *if*[7] $\varepsilon \in cts(P)$,
- *UC locally satisfies $\text{CT}(x)$, and*
- *for every arc $(x, c) \in ES'$, and for every $\pm p \in \text{CT}'(c)$: $\mp p \notin \text{CT}(c)$.*

We say that UC matches *x.*

Next we define the operation which expands an $\mathcal{A}_2$-completion structure by adding a new UCS which matches an unexpanded node in the structure. Suppose that x is such an unexpanded node in an $\mathcal{A}_2$-completion structure $CS = \langle EF,\ \text{CT},\ \text{ST},\ G\rangle$, with $G = (V, A)$,

[7]Unit completion structures with roots constants can only be matched with the corresponding constant nodes.

and that x is matchable with a unit completion structure $UC = \langle EF^{'}, \textsc{ct}^{'}, G' \rangle$, with root ε, $EF = (F', ES^{'})$, and $G^{'} = (V^{'}, A^{'})$. Suppose also that $x \in T_c \in EF$. Node x can then be expanded by replacing it with UC using an operation called $expand_{CS}(x, UC)$ which updates CS as follows:

- $\textsc{st}(x) = exp$,
- $T_c = T_c \cup (T_\varepsilon)_{\varepsilon||x}$;
- $ES = ES \cup \{(x, v) \mid (\varepsilon, v) \in ES'\}$;
- if $u \in T_\varepsilon$ and $v \in succ_{EF'}(u)$: $\textsc{ct}(u_{\varepsilon||x}) = \textsc{ct}'(u)$ and $\textsc{ct}(u_{\varepsilon||x}, v_{\varepsilon||x}) = \textsc{ct}'(u, v)$;
- if $u \in T_c[x]$ (the new $T_c[x]$) and $v \in N_{EF}$: $\textsc{st}(u_{\varepsilon||x}) = \textsc{st}'(u)$ and $\textsc{st}(u_{\varepsilon||x}, v_{\varepsilon||x}) = \textsc{st}'(u, v)$;
- for all $c \in cts(P)$: $\textsc{ct}(c) = \textsc{ct}(c) \cup \textsc{ct}'(c)$;
- $V = V \cup \{a_{\varepsilon||x} \mid a \in V'\}$;
- $A = A \cup \{(a_{\varepsilon||x}, b_{\varepsilon||x}) \mid (a, b) \in A'\}$.

Rule. Match. *For a node $x \in N_{EF}$: if $\textsc{st}(x) = unexp$, non-deterministically choose a unit completion structure UC which matches x and perform $expand_{CS}(x, UC)$.*

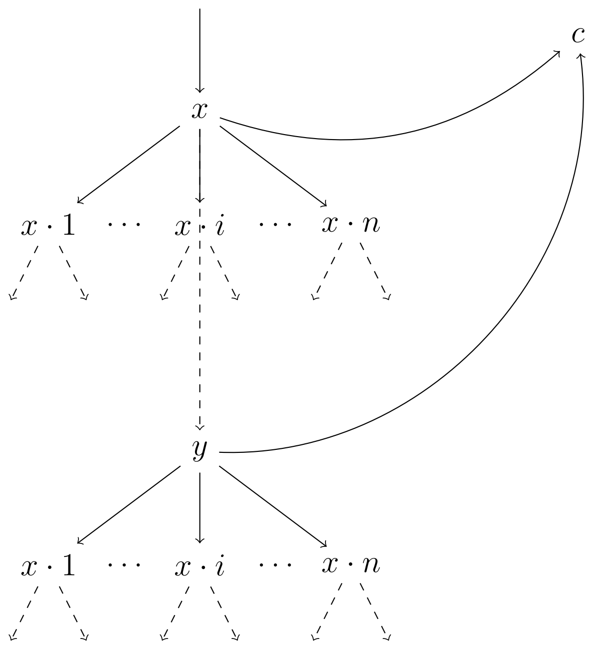
Now that we have a way to evolve a completion structure some conditions regarding termination are in order. The algorithm uses two rules for this: the first one, blocking, describes a condition for successful termination of expansion of a branch of a completion structure, while the other, redundancy, describes a condition for unsuccessful termination - if this condition is met, the algorithm backtracks.

Rule. (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.:*

- $\textsc{ct}(x) \subseteq \textsc{ct}(y)$*, and*
- *the set $connpr_G(y, x) = \{(p, q) \mid (p(y), q(x)) \in paths_G \wedge q$ is not free$\}$ is empty.*

We call (y, x) a blocking pair. *No expansions can be performed on a blocked node ($\textsc{st}(x)$=exp).*

Unlike the typical case for tableau algorithms for DLs [4], subset blocking is not enough for pruning the expansion of a path in the extended forest. To understand why, we recall the intuition behind using blocking techniques in tableau algorithms: the idea is that a completion structure which contains a blocking pair (y, x) is unfolded to a model by



c
x
$x \cdot 1$ $\cdots$ $x \cdot i$ $\cdots$ $x \cdot n$
y
$x \cdot 1$ $\cdots$ $x \cdot i$ $\cdots$ $x \cdot n$

Figure 3.2: Justifying a blocked node y by replicating the justification of its corresponding blocking node x

justifying the content of the blocked node x similarly to the way the content of its corresponding blocking node y has been already justified. This can be done either by copying the subtree T_y at x or by reusing the successors of y as successors of x. The first case is described by Figure 3.2: in this case one obtains an infinite forest shaped model, as the new copy of T_y contains a new copy of x, which again will be justified by copying there T_y, and so on. In the second case, the resulted model is no longer forest-shaped: this is the way we construct models from completion structures in our Soundness proof and it is depicted in Figure 3.7.

The particularity in dealing with FoLPs consists in the fact that unraveling the completion by applying one of the two operations described above can potentially introduce infinite paths in G (in the first case) or cycles in G (in the second case). This would contradict the fact that every atom in the open answer set has to be finitely motivated [48, Theorem 2]. In order to avoid this, the blocking rule verifies also that there is no path from a $p(y)$ to a $q(x)$. The extra condition makes the blocking rule insufficient to ensure the termination of the algorithm. The following applicability rule ensures termination.

Rule. (ix) Redundancy. *A node $x \in N_{EF}$ is* redundant *iff:*

- *x is saturated and 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. $\textsc{ct}(x) = \textsc{ct}(y_i)$.*

The algorithm aborts when a redundant node is detected.

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 $(k + 1)th$ node, in case it exists, is blocked [39]. One can thus search for forest models only of the latter type. As such the detection of a redundant node indicates a failure in the expansion process and stops the expansion.

Next we define when the expansion of a completion structure is *complete*, and when the completion structure is a 'good one', i.e. it is *clash-free*.

Definition 3.3.8. *An* $\mathcal{A}_2$-complete completion structure *for a FoLP P and a unary predicate $p \in upreds(P)$, is a completion structure that results from repeated applications of the rule* Match *to an initial completion structure for p and P, taking into account the applicability rules (viii) and (ix), s. t. no further rules can be further applied.*

The local clash conditions regarding contradictory structures or structures which have cycles in the dependency graph G are no longer relevant: if among the first $k + 1$ nodes on a path with equal content, there is no blocking node, the last node on the path is redundant.

Definition 3.3.9. *An $\mathcal{A}_2$-completion structure $CS = \langle EF, \textsc{ct}, \textsc{st}, G\rangle$ is* clash-free *if EF does not contain redundant nodes and there is no node $x \in N_{EF}$ s.t. $st(x) = unexp$.*

3.3.2.3 Termination, Soundness, Completeness

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

Proposition 3.3.10. *An initial completion structure for a unary predicate p and a FoLP P can always be expanded to an $\mathcal{A}_2$-complete completion structure in a finite number of steps.*

The algorithm is sound and complete:

Proposition 3.3.11. *A unary predicate p is satisfiable w.r.t. a FoLP P iff there is an $\mathcal{A}_2$-complete clash-free 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.

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

Proposition 3.3.12. $\mathcal{A}_2$ *runs in the worst-case in double exponential time.*

3.4 Optimized/Optimal Reasoning with FoLPs

This section describes $\mathcal{A}_3$, the worst-case optimal algorithm we developed during the last year of the project. Like before, a structure is constructed by appending UCSs using the *Match* rule, but a different strategy is employed for termination.

Firstly, the algorithm employs a technique that identifies when some redundant computation has been performed during the expansion of a path and stops the expansion of that path, much earlier than the redundancy rule in $\mathcal{A}_1$ did. This led to the replacement of the redundancy rule with a new rule with the same name. This rule is described in Section 3.4.1.

Secondly, $\mathcal{A}_3$ is able to identify when some computation on a path can be reused during the expansion of another path: if a node which is currently selected for expansion is similar to a non-ancestor node which has already been expanded, the justification of the latter is reused when dealing with the current node. The new rule which deals with this is called *caching*. This rule is described in Section 3.4.2.

Section 3.4.3 introduces the usual notions of complete and clash-free $\mathcal{A}_3$-completion structure, while Section 3.4.4 shows that the algorithm terminates by computing a bound on the size of an $\mathcal{A}_3$-completion structure: a structure has a maximum number of nodes which is exponential in the size of the input program.

Further on, Sections 3.4.5 and 3.4.6 show that the algorithm is sound and complete. While the two new applicability rules are at a first glance not that much different to previous applicability rules they rely on different proof strategies, especially on a different strategy to reduce an infinite model to a finite one (which is part of the completeness proof). As such we consider these proofs to be a main contribution of this work and reproduce them inline.

The usage of the caching rule has improved the worst case running time of the algorithm by one exponential level. The formal complexity analysis can be found in section 3.4.7.

3.4.1 Failure: Redundancy

As discussed in section 3.3.2 the blocking condition is complex enough to not always be fulfilled when exploring a finite number of nodes. The previous algorithm used an extra condition to ensure termination: if a certain number of nodes with equal content had already been explored on a path, there was a failure and the algorithm aborted. Now we introduce a more refined strategy for aborting expansion of a path which is based on the idea that the set of oldest paths running between two nodes with similar content should decrease. While before failure was detected only when reaching a node with exponential depth, the new strategy identifies failure much earlier.

The idea is to keep track of the oldest path in G ('oldest' refers to its starting level w.r.t. the forest) from which every atom makes part and to try to minimize the set of oldest paths running along a path of the forest. Nodes with identical content are allowed on the same path only if every subsequent occurrence of such a node shrinks the set of oldest paths.

A new notation is introduced: by *rank* of an atom a one understands the shallowest depth of a node x such that there exists a unary/binary p/f where $(p(x)/f(x,y),a) \in paths_G$.

Formally:

$$rank(p(x)) = min(\{|x|\} \cup \{rank(a)|(a,p(x)) \in A_G\})$$
$$rank(f(x,y)) = min(\{|y|\} \cup \{rank(a)|(a,f(x,y)) \in A_G\})$$
$$rank(x) = \min_{p \in \mathrm{CT}(x)} rank(p(x))$$

Example 3.4.1. *Consider again the forest model depicted in Figure 3.1. Every atom in the model can be reached by a path starting with $smember(x)$. As $smember(x)$ is not reached by any other atom, its rank is equal to its depth, 1. Thus, all atoms in the model have rank 1.*

Example 3.4.2. *Figure 3.3 shows an extract from a completion structure in which every predicate p in the content of a node x is augmented with the rank of $p(x)$. The arcs between predicates in the content of some node are arcs in the dependency graph: thus, G contains arcs from $b(x)$ to $a(y)$, $b(y)$, and $c(y)$, respectively. As $rank(b(x)) = 1$ we have that also: $rank(a(y)) = rank(b(y)) = rank(c(y)) = 1$.*

We will denote with $in(k,x)$ the set of incoming paths from level k to node x (presuming $|x| \geqslant k$):

$$in(k,x) = \{p|rank(p,x) = k\}$$

Example 3.4.3. *For the completion structure depicted in Figure 3.3 we have that: $in(x,1) = \{a,b\}$, $in(x,2) = \{c\}$, and $in(1,y) = \{a,b,c\}$.*

Formally:

$$\begin{tikzcd}
 & {} \arrow[d] & {} \arrow[d] & \\
x \arrow[ddd, dashed] & \{(a,1), & (b,1)\} \arrow[ddl] \arrow[dd] \arrow[ddr] & (c,2)\} \\
 & & & \\
 & & & \\
y & \{(a,1), & (b,1), & (c,1)\}
\end{tikzcd}$$

Figure 3.3: Redundancy: y is redundant as the set of paths coming from the ancestor at depth 1 increases

Rule. (ix') Redundancy. *A node $x \in N_{EF}$ is* redundant *if there is an ancestor y of x in F, $y <_F x$, $y \notin cts(P)$, s. t.:*

- $\text{CT}(x) \subseteq \text{CT}(y)$,
- $rank(x) = rank(y) = r$, *and* $in(r, x) \supseteq in(r, y)$.

The expansion stops when a redundant node is identified.

Intuition: Both strategies for identifying redundant nodes are related to techniques for reducing an infinite forest model to a finite one (used in the completeness proof of the algorithm). In the general case this can be done by considering nodes in the infinite model which are on the same path and have equal content and collapsing the two nodes onto each other by deleting the path between the two nodes (together with all the paths which start with nodes on this path). However, nodes with equal content cannot be indiscriminately collapsed: some extra conditions have to be met in order for the remaining structure to still remain a model.

In the original algorithm, the technique used for reducing a model was to first identify blocking pairs (nodes with equal content with no path running between them) and then collapse nodes with equal content if the set of paths between a 'reference' node and the first node is included or equal within the set of paths between the reference node and the second node. Some extra conditions had to be met for collapsing the two nodes, like there is no blocking node between them. Such conditions can only be checked at 'proof time', but not at 'construction time'. That's why at construction time one could only use the bound established by this technique, but not the technique itself.

The new technique for reducing models using the set of oldest paths traversing a node does not use any reference point when comparing nodes with equal content. Also, except for checking subset inclusion of the set of oldest paths, no extra condition has to be met before collapsing a node into another. This is due to the fact that when reducing a model and scanning a path, first such redundant nodes are identified and collapsed, and then eventually a blocking pair is found. This is guaranteed by the fact that, for infinite paths, by always chasing the set of oldest paths (and exhausting them in a finite number of steps) we reach a point where there are no running paths between two nodes (within finite

$$\begin{tikzcd}
x\{(smember, 1), not\ rmember, not\ project\} \arrow[d, "\{(supportedBy, 1), not\ \ldots)\}"] \\
y\{(smember, 1), not\ rmember, not\ project\}
\end{tikzcd}$$

Figure 3.4: x and y form a redundancy pair

distance of each other), and due to the infinity of the path we reach two nodes with equal content with this property (within finite distance of each other).

Example 3.4.4. *Nodes x and y in Figure 3.3 are such that $\text{CT}(y) \subset \text{CT}(x)$, and the set of oldest paths is expanding when traversing y: $in(1, y) \supset in(1, x)$. Thus, y is redundant.*

Example 3.4.5. *Consider the FOLP P in example 3.2.2 and the open answer set depicted in Figure 3.6. That particular open answer would never be constructed by our algorithm: if one constructs a completion structure for checking satisfiability of $smember$ w.r.t. P, in the style of that particular forest model, one encounters a redundancy node, y. The situation is depicted in Figure 3.4 (negative predicates do not have ranks): $\text{CT}(x) = \text{CT}(y) = \{smember, not\ rmember, not\ project\}$, $rank\ (\ smember\ (x)) = rank(smember(y)) = 1$ and $in(1, x) = in(1, y) = \{smember\}$, and thus, node y is redundant.*

3.4.2 Caching

Blocking can be generalized to the so-called anywhere blocking or caching where a node reuses the justification/expansion of another node which is not on the same path as itself. Again, the typical condition regarding subset inclusion of the contents of the nodes has to be fulfilled. Additionally, a condition regarding sets of paths running between the common ancestor of the nodes and the nodes themselves has to be fulfilled. Formally:

Rule. (x) Caching. *A node $y \in T \in N_{EF}$ is said to be* cached *if there is a node $x \in T \in N_{EF}$, $y \not<_T x$, $x \not<_T y$, $x \notin cts(P)$, s. t.:*

- *$right_T(y, x)$,*
- *$\text{CT}(y) \subseteq \text{CT}(x)$, and*
- *$connpr_G(z, y) \subseteq connpr_G(z, x)$, where z is the common ancestor of x and y: $z = common_T(x, y)$.*

We call (y, x) a caching pair *and y a* caching node. *A cached node is no longer expanded ($\text{ST}(x)$=exp).*

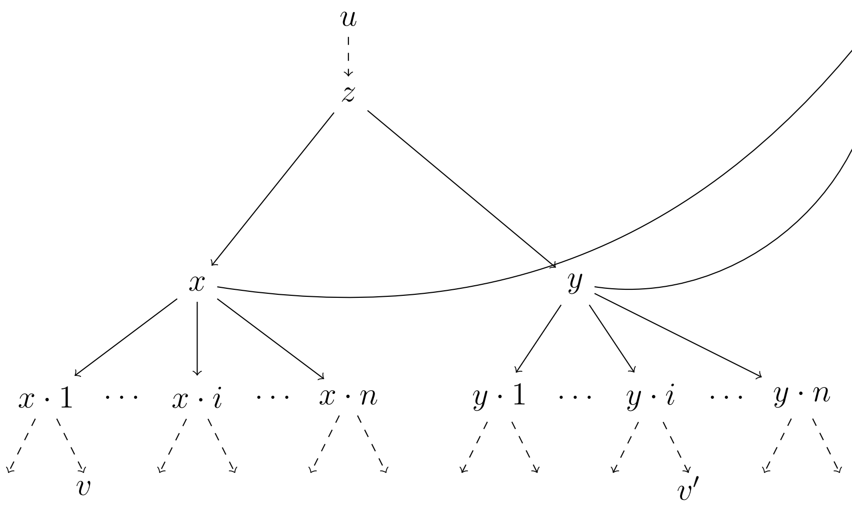


Figure 3.5: Justifying a cached node y by replicating the justification of its corresponding caching node x

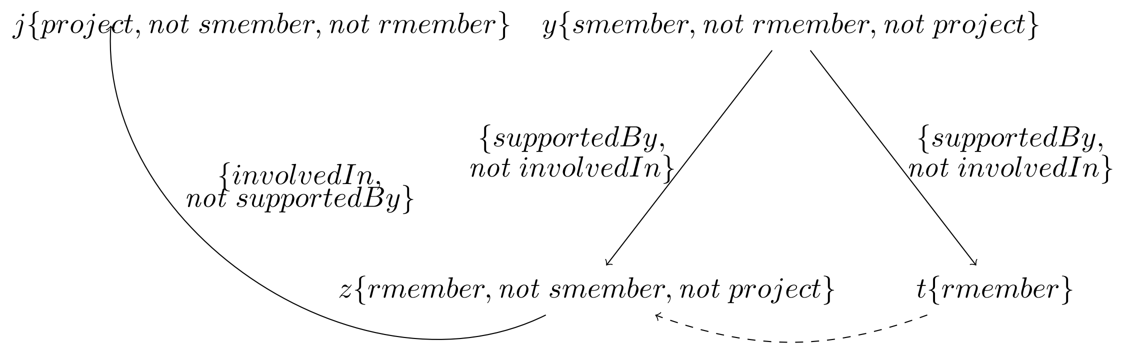
Intuition. Similarly to dealing with blocking pairs, the cached node will be expanded similarly to the caching node. One prerequisite for this is that the content of the cached node is a subset of the content of the caching node.

Like in the case of blocking, the content of the cached node can be justified in two different ways: either by copying the subtree T_x at y or by reusing the successors of x as successors of y. In the first case (depicted in Figure 3.5), it has to hold that if (u, v) is a blocking pair, with u being a leaf node in T_x, and $v \geqslant_T z$, where $z = common_T(x, y)$, then (u, v') is still a blocking pair, where v' is the copy of v in the new subtree T_y (1). In the second case, the obtained model is no longer forest shaped and one has to check that no cycles are introduced in G (2): this is the approach we take in the Soundness proof and it is described in Figure 3.8 in Section 3.4.5. The extra condition $connpr_G(z, y) \subseteq connpr_G(z, x)$ ensures that (1) and (2) hold.

For a cached node to not reuse its own justification in case a successor of its caching node is at its turn a cached node, we impose that a cache node is always 'to the right' of the corresponding caching node in their common tree. Together with this requirement, we enforce the following expansion strategy for the completion structure: *a node $x \in T \in F$ can be expanded iff every node y s.t.: $right_T(y, x)$ is either expanded, blocked, or cached*[8]. This equates to expanding trees in the extended forest in a depth-first manner. The *Match* rule becomes:

Rule. Match'. *For a node $x \in N_{EF}$: if $\text{ST}(x) = unexp$ and for every node y s.t. $right_T(y, x)$: $\text{ST}(y) = exp$, non-deterministically choose a unit completion structure UC*

[8]We want to avoid the situation in a which a node could potentially be cached, but instead it is expanded using the *Match* rule, as its corresponding caching node, which is at its right, has not been yet expanded.



$j\{project, not\ smember, not\ rmember\}$
$y\{smember, not\ rmember, not\ project\}$
$\{supportedBy, not\ involvedIn\}$
$\{supportedBy, not\ involvedIn\}$
$\{involvedIn, not\ supportedBy\}$
$z\{rmember, not\ smember, not\ project\}$
$t\{rmember\}$

Figure 3.6: A completion structure in which (z, t) is a caching pair

which matches x and perform $expand_{CS}(x, UC)$.

Example 3.4.6. *Figure 3.6 shows a completion structure for P from example 3.2.2 in which every node except t is expanded: note that the completion structure contains no redundancy pair. We have that $y = common_T(z, t)$, $\text{CT}(t) \subset \text{CT}(z)$, and $connpr_G(y, z) = connpr_G(y, t) = (smember, rmember)$, and thus z and t form a caching pair: t will be expanded similar to z either by reusing the successors of z or replicating the expansion of z: note that in this case the two types of justification give the same result as the only successor of z is a constant j (thus, also when replicating the expansion of z, a new successor is not introduced, but j is reused).*

3.4.3 Complete/Clash-free Completion structures

In this section we redefine the notions of complete completion structure and clash-free completion structure to reflect on the changes introduced by the new applicability rules.

Definition 3.4.7. *An $\mathcal{A}_3$-*complete completion structure *for a FoLP P and a $p \in upreds(P)$, is an $\mathcal{A}_3$-completion structure that results from the repeated application of the rule* Match' *to an initial $\mathcal{A}_3$-completion structure for p and P, taking into account the applicability rules (viii)* Blocking*, (ix')* Redundancy*, and (x)* Caching *s. t. no rules can be further applied.*

As regards clash conditions, the presence of redundant nodes as defined by rule (ix') *Redundancy* constitutes also in this case a clash. Another clash condition is the impossibility to expand an unexpanded node (by finding an appropriate matchable unit completion structure, blocking or caching node):

Definition 3.4.8. *A complete $\mathcal{A}_3$-completion structure $CS = \langle EF, \text{CT}, \text{ST}, G\rangle$ is* clash-free *if there is no redundant node in EF and for every $x \in N_{EF}$: $\text{ST}(x) = exp$.*

An overview of the algoritm $\mathcal{A}_3$ for checking satisfiability of p w.r.t. a FoLP P is provided by Algorithm 3.1.

3.4.4 Termination

In this section we show that the algorithm $\mathcal{A}_3$ terminates: first we compute a bound on the path length in any $\mathcal{A}_3$-completion structure, and then, using this result, we compute a bound in the total number of nodes in any $\mathcal{A}_3$-completion structure. Both bounds are exponential in the size of the input FoLP P. The latter result is a direct consequence of employing the caching rule.

Proposition 3.4.9. *Every path in an $\mathcal{A}_3$-completion structure for a unary predicate p and a FoLP P has at most an exponential number of nodes in the size of P.*

Proof. We show that any path has at most $n2^{2n}$ nodes, where $n = |upreds(P)|$.

There is a finite amount of nodes with different contents: 2^n, on any path in the completion structure and in the completion structure itself. As such, there are at least $n2^n$ nodes with equal content on any path which contains $n2^{2n}$ nodes. Let $x_1 < \ldots < x_{n2^n}$ be a sequence of such nodes and let $(r_l)_{1 \leqslant l \leqslant n}$ be the ordered sequence of ranks of unary predicates in $\text{CT}(x_1)$: $r_l \in \{k \mid p \in \text{CT}(x_1) \wedge rank(p, x_1) = k\}|$, for $1 \leqslant l \leqslant n$, and $r_l \geqslant r_{l+1}$, for $1 \leqslant l < n$. As some predicates might have equal ranks, and thus $r = |\{k \mid p \in \text{CT}(x_1) \wedge rank(p, x_1) = k\}| < n$, we take $r_i = |x_1|$, for every $i > r$. We show that $rank(x_{j2^n}) > r_j$, for every $1 \leqslant j \leqslant n$ by induction.

Base case: $j = 1$. We have that $rank(x_i) \geqslant rank(x_1) = r_1$, for $1 \leqslant i \leqslant 2^n$, and (x_i, x_k) is neither a blocking nor a caching pair, for any $1 \leqslant i < k \leqslant 2^n$. Assume that $rank(x_{2^n}) = r_1$. Then $rank(x_i) = r_1$, for $1 \leqslant i \leqslant 2^n$, and $in(x_1, r) \not\subseteq in(x_k, r)$, for any $1 \leqslant i < k \leqslant 2^n$. But $|\{S \mid S = in(x_i, r),$ for some $x_i \in N_{EF}$ and $r \in \mathbb{N}\}| = 2^n$, which contradicts with the previous statement. Thus, the original assumption was false and $rank(x_{2^n}) > r_1$.

Induction case: if $rank(x_{2^j}) > r_j$, for a certain $1 \leqslant j < n$, one can bring a similar argument to the one from the base case to show that $rank(x_{2^{j+1}}) > r_{j+1}$.

As such, $rank(x_{j2^n}) > r_j$, for every $1 \leqslant j \leqslant n$, and in particular, $rank(x_{n2^n}) > r_n$, for every $1 \leqslant j \leqslant n$. As $rank(p, x_1) \leqslant r_n$, for every $p \in \text{CT}(x_1)$, it results that $rank(p, x_1) < rank(x_{n2^n})$, for every $p \in \text{CT}(x_1)$. This translates in the fact that the set of oldest paths in G traversing x_{n2^n} started at a node below x_1, and thus there are no paths in G running between x_1 and x_{n2^n}. As $\text{CT}(x_1) = \text{CT}(x_{n2^n})$, this implies that (x_1, x_{n2^n}) is a blocking pair and thus x_{n2^n}, being a blocked node is the last node on the path. This reasoning applies to every possible content for a node, thus in case $n > 1$, we achieve that there have to be less than $n2^{2n}$ nodes on every path: otherwise, there is a blocking node for every possible type of content for a node, which contradicts the fact that a path has at most one blocking node.

Algorithm 3.1: Overview of $\mathcal{A}_3$.

```
input  : FoLP $P$, unary predicate $p$;
output: checks satisfiability of $p$ w.r.t.$P$;
1) Construct the set of Unit Completion Structures (UCSs) for $P$ (if not constructed
already)::;
    To construct a UCS: a) Construct an initial unit completion structure for $p$ w.r.t. $P$
    as in Definition 3.3.3;
    b) Apply expansion rules (i)-(vi) introduced in Deliverable D3.2 [51] until the
    conditions in Definition 3.3.4 are met.;
2) Construct an $\mathcal{A}_2$-initial completion structure for $p$ w.r.t. $P$ as in Definition 3.3.2;
3) For every $x \in N_{EF}$ apply one of the followings rules (in decreasing order of priority)
(we assume $EF$ is explored in a depth-first fashion): ;
    a) if there is an ancestor $y$ of $x$: $y <_F x$, $y \notin cts(P)$, s. t. $\text{CT}(x) \subseteq \text{CT}(y)$, and
    $connpr_G(y, x) = \{(p, q) \mid (p(y), q(x)) \in paths_G \wedge q$ is not free$\}$ is empty then
        $x$ is blocked;
    end
    b) if there is an ancestor $y$ of $x$ in $F$, $y <_F x$, $y \notin cts(P)$, s. t. $\text{CT}(x) \subseteq \text{CT}(y)$,
    $rank(x) = rank(y) = r$, and $in(r, x) \supseteq in(r, y)$ then
        $x$ is redundant: return false;
    end
    c) if there is a node $y \in T \in N_{EF}$, $y \not<_T x$, $x \not<_T y$, $y \notin cts(P)$, s. t. $right_T(x, y)$,
    and $\text{CT}(x) \subseteq \text{CT}(y)$, and $connpr_G(z, x) \subseteq connpr_G(z, y)$, where
    $z = common_T(x, y)$ then
        $x$ is cached;
    end
    d) if $\text{ST}(x) = unexp$ and for every node $y$ s.t. $right_T(y, x)$: $\text{ST}(y) = exp$ then
        non-deterministically choose a unit completion structure $UC$ which matches $x$
        and perform $expand_{CS}(x, UC)$ (Match');
    end
4) if for every node $x \in N_{EF}$: $st(x) = exp$ then
    return true
end
return false.
```

Furthermore, one can show that after an exponential number of steps, one always reaches a complete completion structure. Note that in the previous version of the algorithm a complete completion structure had in the worst case a double exponential number of node in the size of the program. Now, due to caching, the complexity drops one exponential level.

Proposition 3.4.10. *A complete $\mathcal{A}_3$-completion structure for a unary predicate p and a FoLP P has at most an exponential number of nodes in the size of P.*

Proof. Interestingly, the additional condition concerning paths running between the common ancestor and the two nodes in a caching pair can be reformulated in a condition regarding inclusion of the intersections of sets of paths running through the tree with the two nodes ordered by their ranking.

$$connpr_G(z, x) \subseteq connpr_G(z, y) \text{ iff } in(r, x) \subseteq in(r, y), \text{for every } r \leqslant rank(z)$$

This property enables us to obtain an exponential bound on the number of nodes in any complete completion structure using the three applicability rules. To do this we overestimate the number of nodes, by making caching even harder by imposing an even stricter condition: $in(r, x) \subseteq in(r, y),$ for every $r \geqslant 0$.

We count how many structures of the type $((x_1, r_1), (x_2, r_2), \ldots)$ are, where $x_1, x_2, \ldots \in upreds(P) \cup not\ upreds(P)$, $r_1, r_2, \ldots \in \overline{0, n}$, x_i-s are distinct, and n is a natural number exponential in the size of P (the maximum length of a path in a completion structure - see Proposition 3.4.9), or, in other words, the number of possible node contents annotated with the rank of every predicate in the content (predicates which appear negated in the content of some node are annotated with 0). This is equal to the number of functions $f : 2^{upreds}(P) \rightarrow \overline{0, n} \cup \overline{0, n}^2 \cup \ldots \overline{0, n}^{|upredsP|}$ such that $f(x) \in \overline{0, n}^{|x|}$, which at its turn is exponential in the size of P.

Assume there are two distinct nodes with identical annotation structures as described above. If they are on the same path, they form a redundant pair, otherwise they form a caching pair.

3.4.5 Soundness

Proposition 3.4.11 (soundness)**.** *Let P be a FoLP and $p \in upreds(P)$. If there exists a complete clash-free completion structure for p w.r.t. P, then p is satisfiable w.r.t. P.*

Proof. From a clash-free complete completion structure for p w.r.t. P, we construct an open interpretation, and show that this interpretation is an open answer set of P that satisfies p. Let $\langle EF, \text{CT}, \text{ST}, G\rangle$ be such a clash-free complete completion structure with $EF = \langle F, ES\rangle$ the extended forest and $G = (V, A)$ the corresponding dependency

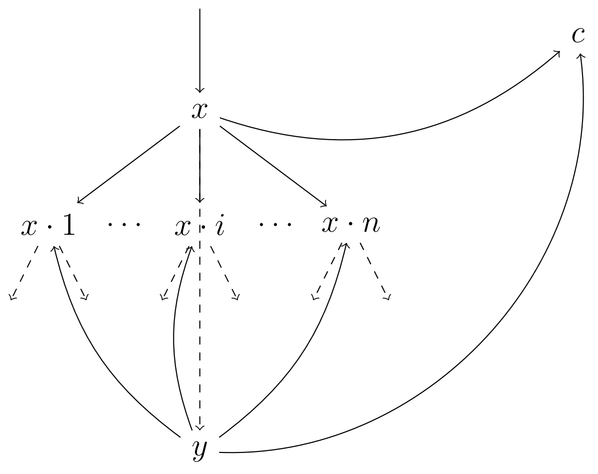


Figure 3.7: Justifying a blocked node y by reusing the successors of its corresponding blocking node x

graph and let bl and ch be the sets of blocking pairs and caching pairs corresponding to the completion. Let $blocked$ and $cached$ be the sets of blocked and cached nodes respectively: $blocked = \{y \mid (x, y) \in bl\}$ and $cached = \{y \mid (x, y) \in ch\}$.

1. *Construction of open interpretation.*

   We construct a new graph $G_{ext} = (V_{ext}, A_{ext})$ by extending G in the following way: for every pair of blocking/caching nodes, the content of the blocking/caching node is copied into the content of the blocked/cached node, and all connections from the blocking/caching node to its successors or within itself are replicated by connections from the blocked/cached node to the successors of the blocking/caching node or within itself (or, in other words, the content of the blocked/cached node is identical with the content of the blocking/caching node and it is motivated in a similar way). The underlying forest is also extended with arcs from the blocked/cached node to all successors of the blocking/caching node. Formally:

   - $V_{ext} = V \cup \{a_{x|y} \mid a \in V \wedge args_1(a) = x \wedge (x, y) \in bl \cup ch\}$;
   - $A_{ext} = A \cup \{(a_{x|y}, b_{x|y}) \mid (a, b) \in A \wedge args_1(a) = x \wedge (x, y) \in bl \cup ch\}$;
   - $A_{EF}^{ext} = A_{EF} \cup \{(y, z) \mid (x, y) \in bl \cup ch \wedge (x, z) \in A_{EF}\}$.

   Lemma 3.4.12. *Let $(x, y) \in bl \cup ch$ and $G_{ext} = (V_{ext}, A_{ext})$ constructed as described above. Then, for any ground rule $r \in P_{N_{EF}}$: $V_{ext} \models r$ iff $V_{ext} \models r_{x|y}$ iff $V_{ext} \models r_{y|x}$.*

   Proof. By construction of V_{ext}. □

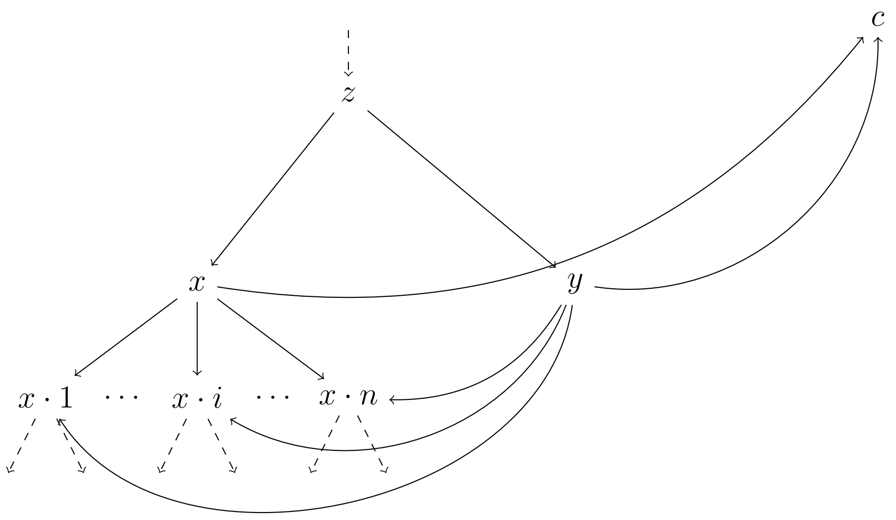


Figure 3.8: Justifying a cached node y by reusing the successors of its corresponding caching node x

Lemma 3.4.13. *Let $UC = \langle EF, \text{CT}, G\rangle$ be a unit completion structure for a FoLP P with $EF = (\{T_\varepsilon\}, ES)$, and $G = (V, A)$. Then, the open interpretation induced by UC: (N_{EF}, V), is an open answer set of the program: $\cup_{r \in P} r_{args_1(head(r))|\varepsilon}$. This is equivalent to $V \models \cup_{r \in P_{N_{EF}}} r_{args_1(head(r))||\varepsilon}$, or, in other words, the set of atoms induced by UC satisfies the grounding of P with elements from N_{EF} s.t. the first term in the head of each ground rule is ε.*

Proof. By construction of a unit completion structure. □

Let there be an open interpretation (U, M), with $U = N_{EF}$, i.e., the universe is the set of nodes in the extended forest, and $M = V_{ext}$, i.e., the interpretation corresponds to the set of nodes in the extended graph.

2. *M is a model of P_U^M.* First of all let's note that $M \models P_U^M$ iff $M \models P_U$. We will show that $M \models P_U$.

   Let's note that $P_U = \cup_{x \in U} \cup_{r \in P_U} r_{args_1(head(r))||x}$.

   For every node $x \in U$ we will show that $M \models \cup_{r \in P_U} r_{args_1(head(r))|x}$:

   - (i) suppose $x \notin blocked \cup cached$. Then, at some point in the construction of CS, x has been expanded by replacing it with a unit completion structure $UC = \langle EF', \text{CT}', G'\rangle$, where $G' = (V', A')$. According to Lemma 3.4.13, $V' \models \cup_{r \in P_{N_{EF'}}} r_{args_1(head(r))||\varepsilon}$. Let $V'' = \{a_{\varepsilon||x} \mid a \in V'\}$. Then $V'' \models \cup_{r \in P_{N_{EF'}}} r_{args_1(head(r))||x}$. As $V'' \subseteq V$, $V \subseteq M$, and $\cup_{r \in P_{N_{EF'}}} r_{args_1(head(r))||x} = \cup_{r \in P_U} r_{args_1(head(r))||x}$, so $M \models \cup_{r \in P_U} r_{args_1(head(r))||x}$.

- (ii) suppose $x \in \mathit{blocked} \cup \mathit{cached}$. Then, according to Lemma 3.4.12, for every $r \in P_U$: $M \models r$ iff $M \models r_{x|y}$, where y is the corresponding blocking or caching node. That $M \models r_{x|y}$ follows from case (i).

3. *M is a minimal model of P_U^M.* Before proceeding with the actual proof we introduce a notation and a lemma which will prove useful in the following. Let $EF^{'}$ be the directed graph $(N_{EF}, A^{'})$ which has as nodes all the nodes from EF and as arcs all the arcs of EF plus some 'extra' arcs which point from blocked/cached nodes to successors of corresponding blocking/caching nodes: $A^{'} = A_{EF} \cup \{(y,z) \mid \exists x \text{ s. t. } (x,y) \in \mathit{bl} \cup \mathit{ch} \wedge z \in \mathit{succ}_{EF}(x)\}$. The new graph captures in a more accurate way the structure of M: blocked/cached nodes are connected to successors of the corresponding blocking nodes, as their contents is justified similarly as the content of the blocking/caching nodes.

   The following lemma associates paths in the dependency graphs G/G_{ext} to paths in the underlying extended forest: $EF/EF^{'}$. It basically says that by projecting a path in the dependency graph on the arguments of every atom in the path and eliminating all binary arguments and redundant unary arguments one obtains a path in the extended forest.

   Lemma 3.4.14. *Let $Pt = (a_1, \ldots, a_n) \in \mathit{paths}_G/\mathit{paths}_{G_{ext}}$, with $\mathit{pred}(a_1) \in \mathit{upreds}(P)$, and $T_1 = (\mathit{args}(a_{i_1}), \ldots, \mathit{args}(a_{i_m}))$ be a tuple obtained by selecting all and only the unary atoms in Pt_1 in the order they appear in Pt and retaining only their argument: $1 \leqslant i_j \leqslant n$, $i_j < i_{j+1}$, for every $1 \leqslant j \leqslant m$, and $\mathit{pred}(a_k) \in \mathit{upreds}(P)$ iff there exists $1 \leqslant j \leqslant m$ such that $k = i_j$, for every $1 \leqslant k \leqslant n$. Then, the tuple obtained by eliminating consecutive duplicates in T_1, $T_2 = (b_1, \ldots, b_p)$, where for every $1 \leqslant j \leqslant p$, there exists $1 \leqslant k \leqslant m$ such that $b_j = \mathit{args}(a_{i_k})$ and $\mathit{args}(a_{i_k}) \neq \mathit{args}(a_{i_{k-1}})$ is a path in $EF/EF^{'}$: $T_2 \in \mathit{paths}_{EF}/\mathit{paths}_{EF^{'}}$. We will also call T_2, the* argument path *of Pt and denote it with $\mathit{argpath}(Pt)$.*

   Furthermore, if Pt_1 is a cycle in G/G_{ext}, than T_2 is a cycle in $EF/EF^{'}$.

   Proof. We construct a sequence of pairs of indexes $((k_1, q_1), \ldots, (k_{p-1}, q_{p-1}))$ such that k_i is the greatest index for which $\mathit{args}(a_{k_i}) = b_i$ and q_i is the smallest index for which $\mathit{args}(a_{k_i}) = b_{i+1}$, for every $1 \leqslant i < p$.

   Then, we consider subpaths of Pt of the form $(a_{k_i}, \ldots, a_{q_i})$, for $1 \leqslant i < p$. Every such subpath has the form: $(p(b_i), f_1(b_i, b_{i+1}), \ldots, f_s(b_i, b_{i+1}), q(b_{i+1}))$, with $p, q \in \mathit{upreds}(P)$, $f_1, \ldots, f_s \in \mathit{bpreds}(P)$, and $s \geqslant 1$. Thus: $(b_i, b_{i+1}) \in A/A^{'}$ for every $1 \leqslant i < p$: T_2 is a path in $EF/EF^{'}$.

   If Pt is a cycle then $a_1 = a_n$. By construction of T_2, $b_1 = b_n = \mathit{args}(a_1)$. $\square$

   Now we can proceed to the actual proof of statement. Assume there is a model $M' \subset M$ of $Q = P_U^M$. Then $\exists l_1 \in M : l_1 \notin M'$. Take a rule $r_1 \in Q$ of the form $l_1 \leftarrow \beta_1$ with $M \models \beta_1$; note that such a rule always exists by construction of M and expansion rule (i) . If $M' \models \beta_1$, then $M' \models l_1$ (as M' is a model), a contradiction.

Thus, $M' \not\models \beta_1$ such that $\exists l_2 \in \beta_1 : l_2 \notin M'$. Continuing with the same line of reasoning, one obtains an infinite sequence $\{l_1, l_2, \ldots\}$ with $(l_i \in M)_{1 \leqslant i}$ and $(l_i \notin M')_{1 \leqslant i}$. M is finite (the complete clash-free completion structure has been constructed in a finite number of steps, and when constructing $M(V_{ext})$ we added only a finite number of atoms to the ones already existing in V), thus there must be $1 \leqslant i, j, i \neq j$, such that $l_i = l_j$. We observe that $(l_i, l_{i+1})_{1 \leqslant i} \in E_{ext}$ by construction of E_{ext} and expansion rule (i), so our assumption leads to the existence of a cycle in G_{ext}.

Assume G_{ext} contains a cycle $C = (a_1, \ldots, a_n = a_1)$. Then, potentially, the cycle falls into one of the following categories:

- 'local' cycles: cycles in which all unary atoms have identical arguments or there are no unary atoms.
- 'blocking' cycles: non-local cycles which do not contain unary atoms having as arguments cached nodes.
- 'caching' cycles: non-local cycles which have as arguments cached nodes.

Note that G does not contain any cycle (by construction), so every cycle in G_{ext} has to be a result of the introduction of new nodes/arcs in G: as such, each cycle should contain at least an atom having as one of its arguments a blocked or cached node. We will show by reductio ad absurdum that each of these types of cycles cannot appear in G_{ext}. In the following we consider only elementary cycles as in the absence of elementary cycles there cannot be any cycles whatsoever.

Lemma 3.4.15. *There are no (elementary) local cycles in G_{ext}.*

Proof. Assume $C = (a_1, \ldots, a_n = a_1)$ is an (elementary) local cycle in G_{ext}. Then C contains only atoms of the form $p(x)$, and/or $f(x, y)$, for $p \in \mathit{upreds}(P)$, $f \in \mathit{bpreds}(P)$, and $x, y \in N_{EF}$. Assume $x \in \mathit{blocked/cached}$. Then let $z \in N_{EF}$ be such that $(x, z) \in \mathit{bl/ch}$. Then $C_{x|z} = ((a_1)_{x|z}, \ldots, (a_n)_{x|z} = (a_1)_{x|z})$ is a cycle in G. Contradiction with the fact that there are no cycles in G. □

To show that there are no elementary blocking cycle in G_{ext} we employ a three step process: the first two steps restrict the set of possible cycles by constraining the structure of the argument path of such a cycle (lemmas 3 and 3).

Lemma 3.4.16. *There is no elementary cycle C in G_{ext} such that its argument path contains a blocking path from EF: for every $x, y \in bl$ such that $x, y \in T$, $path_T(x, y) \not\subseteq argpath(C)$.*

Proof. Assume $path_T(x, y) \subseteq argpath(C)$. Then, there are two nodes $a_1, a_2 \in G$, with $args(a_1) = x$, and $args(a_2) =$ and a path $Pt \in paths_G(a_1, a_2)$ such that $Pt \in C$. But this contradicts with the fact that $connpr_G(x_1, x_2) = \emptyset$. Thus, the initial assumption was false. □

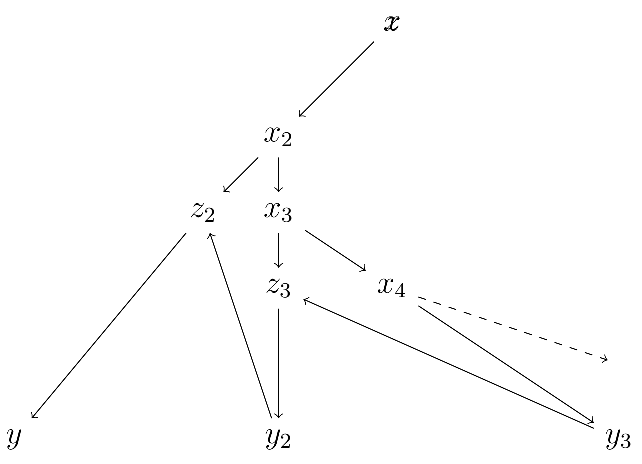


Figure 3.9: Splitting blocking paths: infinite division

Lemma 3.4.17. *Let C be an elementary cycle in G_{ext} which contains a node y such that $(x, y) \in bl$ and $x, y \in T$. Then, $path_T(z, y) \in argpath(C)$, where $z = x \cdot i$ and $z < y$.*

Proof. Assume the opposite. As $y \in argpath(C)$ and $argpath(C)$ is a cycle, there has to be an arc of the type $(t, y) \in argpath(C)$. Let $z_2 \in path_T(succ_T(z), y)$ be such that $path_T(z_2, y) \in argpath(C)$ and $(prev_T(z_2), z_2) \notin argpath(C)$. Every node in $path_T(z, y)$, including z_2, has as incoming arcs the arc from its predecessor in EF and possibly blocking arcs. As $(prev_T(z_2), z_2) \notin argpath(C)$, z_2 has an incoming blocking arc: let (y_2, z_2) be such an incoming blocking arc, where $y_2 \in T$ and let $x_2 \in T$ be the corresponding blocking node: $(x_2, y_2) \in bl$. As (y_2, z_2) is a blocking arc, it means that $z_2 \in succ_T(x_2)$, or in other words $x_2 = prev_T(z_2)$. As $z_2 \in path_T(succ_T(z), y)$, it implies that $x_2 \in path_T(z, prev_T(y))$.

As $y_2 \in argpath(C)$ and $argpath(C)$ is a cycle, there has to be an arc of the type $(t, y_2) \in argpath(C)$. From lemma 3 we know that $path_T(x_2, y_2) \nsubseteq argpath(C)$. Thus, there is a node $z_3 \in path_T(succ_T(x_2), y_2)$ such that $path_T(z_3, y_2) \in argpath(C)$ and $(prev_T(z_3), z_3) \notin argpath(C)$. Like before in the case of z_2, z_3 has an incoming blocking arc (y_3, z_3) with $(x_3, y_3) \in bl$. In this case: $x_3 \in path_T(x_2, prev_T(y_2))$.

The process repeats itself ad infinitum: figure 3.9 describes it in the general case. One obtains a sequence of tuples (x_i, y_i, z_i) such that $(x_i, y_i) \in bl$, $x_{i+1} \in path_T(x_i, prev_T(y_i))$, $z_i = succ_T(x_i)$, $path_T(z_{i+1}, y_i) \in argpath(C)$, and $(y_i, z_i) \in argpath(C)$. If $y_i \neq y_j$, for every $i \neq j$, one has an infinite number of nodes in EF which is a contradiction with the fact that there is a finite amount of nodes in EF.

If however there exist $l < k$ such that $y_k = y_l$, then $x_l = x_{l+1} = \ldots x_k$, and $(y_k, z_k)\hat{}\ path_T(z_k, y_k - 1)\hat{}\ \ldots\hat{}\ (y_{l+1}, z_l)\hat{}\ path_T(z_{l+1}, y_l = y_k)$ is a cycle in G_{ext} strictly included in C. This contradicts with the fact that C is an elementary cycle (see Figure 3.10).

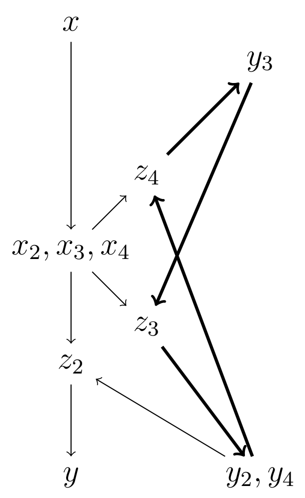


Figure 3.10: Splitting blocking paths: the case where $x_2 = x_3 = x_4$ and $y_2 = y_4$: z_4, y_3, z_3, y_2, z_4 form a cycle

Thus, in both cases we obtained a contradiction, and the initial assumption was false.

Lemma 3.4.18. *There are no elementary blocking cycles in G_{ext}.*

Proof.

Assume $C = (a_1, \ldots, a_n = a_1)$ is a blocking cycle in G_{ext} which contains a blocked node $y \in T$: $(x, y) \in bl$. Then, from lemma 3.4.14 $argpath(C)$ is a cycle in EF'. Also, according to lemma 3 $argpath(C)$ does not contain $path_T(x, y)$, but according to lemma 3 it does contain $path_T(z, y)$, where $z = succ_T(x)$. As $z \in argpath(C)$, $argpath(C)$, and $(x, z) \not\subseteq argpath(C)$ there has to be a blocking arc of the type $(t, z) \in argpath(C)$. The situation is described in Figure 3.11.

Due to the construction of $argpath(C)$, there have to be unary predicates $p, q, r \in upreds(P)$ such that $(p(t), q(z)) \in A_{ext}$ and $(q(z), r(y)) \in conn_G$. But, as $prev_T(z) = x$, (x, t) is a blocking pair, and $(p(t), q(z)) \in A_{ext}$ implies $(p(x), q(z)) \in A$. Together with $(q(z), r(y)) \in conn_G$ it implies that $(p(x), q(z)) \in conn_G$, and thus $(p, q) \in connpr_G(x, y)$, which is in contradiction with $(x, y) \in bl$.

Thus, the initial assumption was false and G_{ext} does not contain any elementary blocking cycle.

Lemma 3.4.19. *Let C be an elementary caching cycle in G_{ext} for which $argpath(C)$ contains a cached node y such that $(x, y) \in ch$ and for every pair $(s, t) \in ch$, with $s, t \in T$: $right_T(s, x)$ or $t \notin C$. Then, $path_T(z \cdot i, y) \subseteq argpath(C)$, where $z = common_T(x, y)$ and $z < z \cdot i \leqslant y$.*

$$\begin{tikzcd}
 & x \arrow[dl, dashed] \arrow[d] \\
t \arrow[r] & z \arrow[d] \\
 & y
\end{tikzcd}$$

Figure 3.11: If (t, z) is a blocking arc, (x, t) is a blocking pair, and the content of (t, z) is justified similarly to the content of (x, z), then $paths_G(t, y) = paths_G(x, y)$

Proof.

Assume that $argpath(C)$ does not contain $path_T(z \cdot i, y)$. As $y \in argpath(C)$ and $argpath(C)$ is a cycle, there has to be an arc of the type $(t, y) \in argpath(C)$. Let $z_2 \in path_T(succ_T(z \cdot i), y)$ be such that $path_T(z_2, y) \in argpath(C)$ and $(prev_T(z_2), z_2) \notin argpath(C)$. Every node in $path_T(z, y)$, including z_2, has as incoming arcs the arc from its predecessor in EF and possibly blocking and/or caching arcs. $(prev_T(z_2), z_2) \notin argpath(C)$, so z_2 must have either an incoming caching arc or an incoming blocking arc which is part of $argpath(C)$.

- (i) Assume z_2 has an incoming caching arc (*). Then, there is a caching pair $(prev_T(z_2), t)$ and $prev_T(z_2)$ is a caching node. We have that $prev_T(z_2) \in path_T(z \cdot i, prev_T(y))$ and thus $z_2 \geqslant_T z \cdot i$. At the same time $x > z$ and due to the expansion and caching strategy $right_T(y, x)$. Thus: $x = z \cdot j \cdot s$, for some $s \in \langle \mathbb{N}^* \rangle$, and $j \in \mathbb{N}^*$, so it holds that $right_T(z_2, x)$. This in contradiction to the fact that x is the right-est caching node in $argpath(C)$. Thus (*) was false, and there are no incoming caching arcs to z_2 which are part of $argpath(C)$.
- (ii) Assume z_2 has an incoming blocking arc. Then, there is a blocking pair $(x_2 = prev_T(z_2), y_2)$ and x_2 is a blocking node with: $x_2 \in path_T(z \cdot i, prev_T(y))$. As there is no cycle which contains a blocking path, $path_T(x_2, y_2)$ is not part of the cycle. The argument follows similarly to the argument in the proof of lemma 3: a sequence of tuples (x_i, y_i, z_i) is constructed with similar properties as in the other lemma. This situation is described in Figure 3.12. The only difference to lemma 3, is that we always have to show that (x_i, y_i) cannot be a caching path, which is done similarly to item (i). As in the proof of lemma 3 we eventually reach a contradiction.

Thus, in both cases we reach a contradiction, and the original assumption was false:

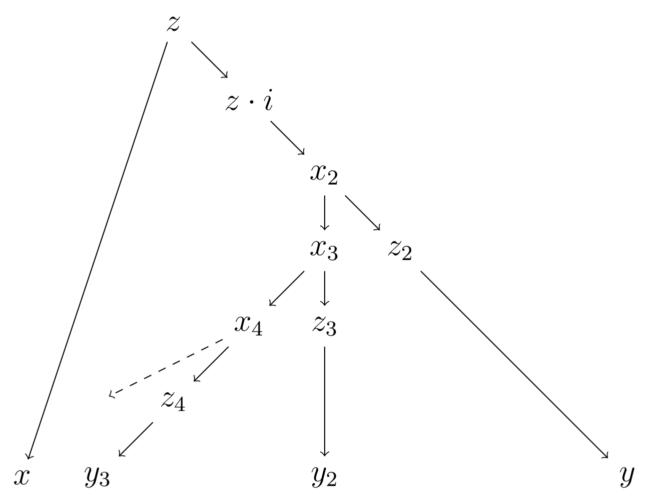


Figure 3.12: Splitting blocking paths in a potential caching cycle

$path_T(z \cdot i, y) \subseteq argpath(C)$. □

Lemma 3.4.20. *Let C be an elementary caching cycle in G_{ext} such that its argument path contains $n+1$ distinct caching nodes, for $n \in \mathbb{N}$. Then, there is an elementary cycle in G_{ext} such that its argument path contains n distinct caching nodes.*

Proof. Let $Pt = argpath(C)$. Let $(x, y) \in ch$ such that: $y \in Pt$, and for every pair $(s, t) \in ch$, with $s, t \in T$: $right_T(s, x)$ or $t \notin C$. Then, according to lemma 3.4.19: $path_T(z \cdot i, y) \subseteq Pt$, where $z = common_T(x, y)$. In the following we show how to transform C in a cycle which does not contain y. (x, y) is a caching pair, so there must be a successor of x in T, $x \cdot j$, such that $(y, x_j) \subseteq Pt$.

There are two distinct cases:

- (i) $path_T(z, y) \subseteq Pt$: let $Pt_1 = path_T(z, y)\hat{}(y, x \cdot j)$ and $Pt_2 = path_T(z, x \cdot j)$. Then, for every path $Pg_1 \in paths_{G_{ext}}(p(z), q(x \cdot j))$, for some $p, q \in upreds(P)$, such that $argpath(Pg_1) = Pt_1$, there is a path $Pg_2 \in paths_{G_{ext}}(p(z), q(x \cdot j))$, such that $argpath(Pg_2) = Pt_2$. This follows from the fact that $connpr_G(z, y) \subseteq connpr_G(z, x)$ and $connpr_G(y, x \cdot j) = connpr_G(x, x \cdot j)$ (from the caching condition and construction of G_{ext}).
  Thus, a path $Pg \in paths_{G_{ext}}$ with $argpath(Pg) = path_T(z, y)\hat{}(y, x \cdot j)\hat{}R$, for some $R \in paths_{G_{ext}}$, is a cycle iff there is another path $Pg' \in paths_{G_{ext}}$ with $argpath\ (\ Pg') = path_T(z, y)\hat{}(y, x{\cdot}j)\hat{}R$, which is is a cycle. $argpath(Pg')$ does not contain cached node y and does not introduce any new cached node, so, it decreases the number of cached nodes in the cycle. Figure 3.13 depicts this case: the thick lines are the part from $argpath(Pg)$ which is replaced with $path_T(z, x \cdot j)$.

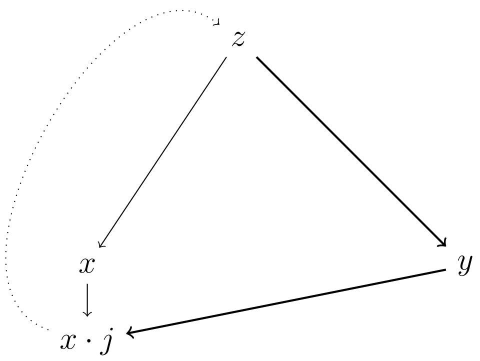


Figure 3.13: Reducing the number of cached nodes which appear in atoms in a cycle of G: $(x, y) \in ch$ and y is eliminated.

- (ii) $path_T(z \cdot i, y) \subseteq Pt$, but $path_T(z, y) \not\subseteq Pt$: in this case $z \cdot i$ has an incoming blocking or caching arc $(t, z \cdot i)$, which translates in its predecessor z being a blocking or caching node and $(z, t) \in bl \cup ch$. In either of the cases, $(t, z \cdot i)$ is justified similarly to $(z, z \cdot i)$ and thus one can obtain an equivalent cycle by substituting t with z in C. The new cycle $C' = C_{t|y}$ fulfills the condition that $path_T(z, y) \subseteq argpath(C')$ and $path_T(z, y) \subseteq Pt$ and thus falls into case (i). Figure 3.14 depicts this case: the thick lines are the part from $argpath(C)$ which is replaced with $path_T(z, z \cdot i)$.

Lemma 3.4.21. *There are no caching cycles in G_{ext}.*

Proof. Assume C is a caching cycle in G_{ext} which contains n caching nodes. Then, by repeated application of lemma 3.4.20, one obtains a cycle with 0 caching nodes, thus a cycle which is either a blocking or local cycle. According to lemmas 3.4.18 and 3.4.15 there are no such cycles in G_{ext}, thus the initial assumption is false, and there are no caching cycles in G_{ext}. □

3.4.6 Completeness

Proposition 3.4.22 (completeness)**.** *Let P be a FoLP and $p \in upreds(P)$. If p is satisfiable w.r.t. P, then there exists a complete clash-free completion structure for p w.r.t. P.*

Proof.

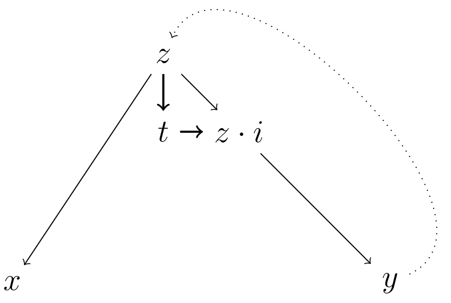

$$\begin{tikzcd}
 & z \arrow[dl] \arrow[d] \arrow[dr] & & \\
 & t \arrow[r] & z \cdot i \arrow[dr] & \\
x & & & y \arrow[uull, dotted, bend right]
\end{tikzcd}$$

Figure 3.14: Reducing the number of cached nodes which appear in atoms in a cycle of G: reducing a cycle C in which $(z, z \cdot i) \not\subseteq argpath(C)$ to a cycle in which $(z, z \cdot i) \subseteq argpath(C)$.

First, we introduce an operation which replaces the node y of a completion structure $CS = \langle EF, \text{CT}, \text{ST}, G\rangle$, where $EF = (F, ES)$ and $G = (V, A)$, with a matchable unit completion structure $UC = \langle EF' = (F', ES'), \text{CT}', \text{ST}', G'\rangle$ with root ε. The result of the operation is a new completion structure obtained by (i) deleting $T_c[y]$ from CS, where $y \in T_c$, and (ii) adding UC instead using the $expand$ operation introduced in section 3.3.2, and is denoted with $replace_{CS}(y, UC)$.

i) The removal of $T_c[y]$ transforms CS as follows:

- $ES = ES - \{(u, v) \mid u \in T_c[y]\}$;
- CT and ST are undefined for $\{u \mid u \in T_c[y]\} \cup \{(u, v) \mid u \in T_c[y]\}$;
- $V = V - \{a \mid args_1(a) \in T_c[y]\}$, $A = A - \{(a, b) \mid args_1(a) \in T_c[y]\}$;
- $T_c = T_c - T_c[y]$.

ii) The addition of UC: $expand_{CS}(x, UC)$.

If p is satisfiable w.r.t. P then p is forest-satisfiable w.r.t. P (Proposition 3.2.4). We construct a clash-free complete completion structure for p w.r.t. P, by guiding the application of the match, blocking, caching, and redundancy rules with the help of a forest model of P which satisfies p. The proof is inspired by completeness proofs in Description Logics for tableau, for example in [57], but requires additional mechanisms to eliminate redundant parts from Open Answer Sets.

Lemma 3.4.23. *Let (U, M) be a forest model for a FoLP P, with the extended forest $EF = \langle\{T_\varepsilon\} \cup \{T_a \mid a \in cts(P)\}, ES\rangle$, and the labeling function $\mathcal{L} : \{T_\varepsilon\} \cup \{T_a \mid a \in cts(P)\} \cup A_{EF} \to 2^{preds(P)}$ as in definition 3.2.3. Then, for every node $x \in U$, there is* *a unit completion structure for P: $UC = \langle EF', \textsc{ct}, G\rangle$, with $EF' = (\{T_{\varepsilon'}\}, ES')$, and $G = (V, A)$, which satisfies the following:*

- $y \in N_{EF'}$ *iff* $y_{\varepsilon'||x} \in N_{EF}$*;*
- $(\varepsilon', y) \in A_{EF'}$ *iff* $(x, y_{\varepsilon'||x}) \in A_{EF'}$*;*
- $\textsc{ct}(\varepsilon') = \mathcal{L}(x) \cup \mathit{not}\ (\mathit{upreds}(P) - \mathcal{L}(x))$*;*
- $\textsc{ct}(y) \subseteq \mathcal{L}(y_{\varepsilon'||x}) \cup \mathit{not}\ (\mathit{upreds}(P) - \mathcal{L}(y_{\varepsilon'||x}))$*, for every* $y \in N_{EF'}$*;*
- $\textsc{ct}(\varepsilon', y) = \mathcal{L}(x, y_{\varepsilon'||x}) \cup \mathit{not}\ (\mathit{upreds}(P) - \mathcal{L}(x, y_{\varepsilon'||x}))$*, for every* $y \in N_{EF'}$*.*

Proof. Follows from the completeness of algorithm $\mathcal{A}_2$.

Now we proceed to the actual construction. Let U, M be the forest model which guides the expansion with $EF = \langle\{T_\varepsilon\} \cup \{T_a \mid a \in \mathit{cts}(P)\}, ES\rangle$, where $p \in \mathcal{L}(\varepsilon)$ and let $CS = \langle EF', \textsc{ct}, \textsc{st}, G\rangle$ be an initial completion structure for checking satisfiability of p w.r.t. P with $EF' = \langle\{T'_{\varepsilon'}\} \cup \{T'_a \mid a \in \mathit{cts}(P)\}, ES'\rangle$, where $p \in \textsc{ct}(\varepsilon')$. We will expand CS in a depth-first fashion (the order of processing trees is not important, just that their contents are expanded depth-first; the expansions of different trees can also be interleaved). Always a node with status unexp is selected for expansion.

Let π be a function which relates nodes from the extended forest in the completion structure in construction to nodes in the forest model: $\pi : N_{EF'} \to U$. We show that at any point during the construction the following property holds: (Γ) for every node $x \in N_{EF'}$ there is a node $\pi(x) \in N_{EF}$, such that $\textsc{ct}(x) \subseteq \mathcal{L}(\pi(x)) \cup \mathit{not}\ (\mathit{upreds}(P) - \mathcal{L}(\pi(x)))$. Intuitively, the positive content of a node/edge in the completion structure is contained in the label of the corresponding forest model node, and the negative content of a node/edge in the completion structure cannot occur in the label of the corresponding forest model node.

The property will be proved by induction and it is used at every step of the construction (for nodes for which it was already proved to hold): as such the induction step coincides with the construction step.

Base case: We set $\pi(\varepsilon') := \varepsilon$ and $\pi(a) := a$, for every $a \in \mathit{cts}(P)$. That the induction hypothesis is fulfilled follows from the way the initial completion structure for p w.r.t. P was defined.

Induction/Construction step: Let x be the node currently selected for expansion in EF': $\textsc{st}(x) := \mathit{unexp}$. Perform the following operations:

(i) Check whether the blocking or caching conditions are met:

- assume there is a node $y \in N_{EF'}$ such that (y, x) form a blocking pair. Then mark x as a blocked node and stop its expansion.

- assume there is a node $y \in N_{EF'}$ such that (y, x) form a caching pair. Then mark x as a cached node and stop its expansion.

Naturally, in both cases (Γ) still holds, as we have not modified the content of nodes and we also did not add any new nodes. Note that when applying the blocking or caching rule we no longer use the guidance of (U, M): (U, M) might justify in a different way the atoms which have x and its successors as one of their arguments; we are interested in finding a finite representation of a model which satisfies p, not necessarily of the original model which we used for guidance (actually the soundness proof constructs a non-forest model from a clash-free complete completion structure).

(ii) If x is neither blocked nor cached, according to the induction hypothesis, there is a node $\pi(x) \in N_{EF}$ such that $\textsc{ct}(x) \subseteq \mathcal{L}(\pi(x)) \cup \mathit{not}\ (\mathit{upreds}(P) - \mathcal{L}(\pi(x)))$. Let UC be a unit completion structure with root ε' corresponding to node $\pi(x)$ as in Lemma 3.4.23. UC has the property that $\textsc{ct}(\varepsilon'') = \mathcal{L}(\pi(x)) \cup \mathit{not}\ (\mathit{upreds}(P) - \mathcal{L}(\pi(x)))$ and $\textsc{ct}(a) \subseteq \mathcal{L}(a) \cup \mathit{not}\ (\mathit{upreds}(P) - \mathcal{L}(a))$, for every $a \in \mathit{cts}(P)$; this, together with the induction hypothesis, implies that $x \in N_{EF'}$ is matchable with UC. Apply the *Match* rule for x and UC.

For every node y added to/updated from EF' by addition of UC: $y \in N_{EF'}$ and $(x, y) \in A_{EF'}$, we have that: $\textsc{ct}(y) \subseteq \mathcal{L}(y_{x||\pi(x)}) \cup \mathit{not}\ (\mathit{upreds}(P) - \mathcal{L}(y_{x||\pi(x)}))$. We set $\pi(y) = y_{x||\pi(x)}$, for every such node, and the induction hypothesis holds.

(ii) Check whether the redundancy rule condition is met: assume there is a node $y \in N_{EF'}$ such that (y, x) form a redundancy pair.

Note that unlike the models constructed by our algorithm, arbitrary forest models might contain 'redundant' nodes (or better said they translate to completion structures which contain such nodes). A redundancy pair (y, x) signals a redundant computation in the form of the tree in the extended forest from y to x The way to overcome this is to simply ignore the redundancy when constructing a completion structure. As the redundant part of the model is first incorporated in the completion structure, when encountering such a redundancy pair we modify the structure by cutting out the redundant part: y is replaced with x (technically with the completion structure at x): $\mathit{replace}_CS(y, CS_x)$.

As concerns the image of y under π in EF, it is changed to the previous image of x: $\pi(y) := \pi(x)$. The induction hypothesis still holds.

Given that the construction process described above terminates after a finite amount of time, its result is obviously a clash-free complete completion structure: the extended forest has been constructed by appending UCS-s to matchable nodes of the forest, no rule can be further applied, all redundant nodes are eliminated, and every node is expanded, blocked, or cached. Next we show by reductio ad absurdum that the construction can always be performed in a finite amount of steps.

We show that the number of operations related to constructing a path of the completion structure is finite. Assume the opposite, that the construction of a path in the structure does not terminate. First, one can only apply blocking or caching once on every path. Second, every completion structure has a finite number of nodes (from the Termination theorem). The only possibility for the construction to not terminate is by application of the redundancy rule an infinite number of times: note that also in this case, the path in construction should always have a finite number of nodes. Thus, in this case, there will be a repeated processing of chunks of the forest model which are found to be redundant.

In order to formalize this scenario, we first introduce the notion of *relaxed completion structure* which is a completion structure constructed in the usual way, except for the fact that it can contain redundancy pairs: the redundancy rule is not taken into account. Note that any completion structure is a relaxed completion structure, while the reciprocal statement is not true.

Lemma 3.4.24. *Let (x, y) and (y, z) be two redundancy pairs in a relaxed completion structure. Then (x, z) is still a redundancy pair.*

Proof. Follows directly from the definition of redundancy pair and transitivity of the subset-equal relationship. □

Formally, let $(x_i, y_i)_{i \geqslant 0}$ be the infinite sequence of redundancy pairs which are identified during the construction process on the same path of the completion structure. Note that these redundancy pairs do not coexist at any time during the construction process: each time a new pair is identified, previous redundancies have already been removed. Let also $CS^0 = \langle EF^0, ct^0, st^0, G^0 \rangle$ be a relaxed completion structure which is constructed similarly to the completion structure in discussion, CS, all steps being the same except that in the case of CS^0 the redundancy rule does not apply. Starting from CS^0, we define inductively a sequence of relaxed completion structures $(CS^i)_{i \geqslant 0}$, each one (except for CS^0) being obtained from the previous completion in the sequence by elimination of the redundant part indicated by the redundancy pair (x_i, y_i). Formally, $CS^i = replace_{CS^{i-1}}(x_{i-1}, CS^{i-1}[y_{i-1}])$. We also introduce the notation u^i to denote the new node in the relaxed completion structure CS^i corresponding to u in CS^0, also denoted as u^0 (if it was not deleted along the way). We have that for every $u^i \in N_{EF^i}$:

$$u^{i+1} = \begin{cases} u^i, \text{ if } u^i \leqslant x_i \\ u^i_{y_i \| x_i} \text{ if } u^i \geqslant y_i \\ undefined, \text{ otherwise} \end{cases}$$

It is clear from the previous identity that for every node $u \in N_{EF^i}$, there exists a node $v \in N_{EF^0}$, such that $u = v^i$ (as nodes are always removed from the original relaxed completion structure). Let $u_i, v_i \in N_{EF^0}$ be such that $x_i = u_i^i$ and $y_i = v_i^i$, for every $i \geqslant 0$. As each redundancy pair 'appears' later in the construction process we have that $v_{i+1} > v_i$, for every $i > 0$. As the path in CS^0 to which u_i and v_i belong is infinite and the result of removing every redundancy pair is a finite pair, an infinite part of the path is eventually removed. This infinite part is formed from chained pairs of nodes which correspond to redundancy pairs in some 'future' relaxed completion structure. The following lemma formalizes this observation.

Lemma 3.4.25. *There is a sequence $(k_i)_{i \geqslant 0}$ (possibly infinite) such that $v_{k_i} = u_{k_{i+1}}$ and an index $n \geqslant 0$ such that $path(v_{k_0}, u_{k_n})$ is a path of infinite length in CS^0.*

Proof. We start by constructing a sequence of pairs $(v_{l_i}, u_{l_i})_{l \geqslant 0}$ such that $v_{l_{i+1}} \geqslant u_{l_i}$, for every $i \geqslant 0$. For this we simply eliminate all pairs (v_i, u_i) from the original sequence for which there is an index j such that $v_j \leqslant v_i < u_i < u_j$. Note that this sequence might be finite. The number of nodes after applying all transformations corresponding to the redundancy pairs is: $\sum_{u_{l_i} < v_{l_{i+1}}} |path(u_{l_i}, v_{l_{i+1}})|$. This is a finite number, thus also $|\{i \mid u_{l_i} < v_{l_{i+1}}\}|$ is also finite. We have that $\sum_i |path(v_{l_i}, u_{l_i})|$ is infinite. Depending on the length of the sequence $(v_{l_i}, u_{l_i})_{l \geqslant 0}$ there are two possibilities:

- the sequence is finite: then there must be an index l_m such that $path(v_{l_m}, u_{l_m})$ is infinite. Let $k_0 = k_n = m$.
- the sequence is infinite: as, $|\{i \mid u_{l_i} < v_{l_{i+1}}\}|$, there must be an index l_m such that $u_{l_i} = v_{l_{i+1}}$, for every $i \geqslant m$. Let $k_0 = m$ and k_n some arbitrary number at infinite distance from k_0.

The following lemma will prove useful in concluding our argument regarding the impossibility of applying the redundancy rule an infinite number of times when constructing a path in a completion structure guided by a forest model.

Lemma 3.4.26. *For every $i \geqslant 0$, $rank(u_i) = rank(v_i)$.*

Proof. As (x_i, y_i) is a redundancy pair, it is clear that $rank(u_i^i) = rank(v_i^i)$. We show that $rank(u_i^j) = rank(v_i^j)$ implies $rank(u_i^{j-1}) = rank(v_i^{j-1})$, for every $0 < j \leqslant i$. Figure 3.15 depicts the possible positions of u_i^{j-1}, v_i^{j-1} relative to the positions of x_{j-1} and y_{j-1}. There are two different situations:

- a) $y_{j-1} \leqslant u_i^{j-1}$ (Figure 3.15 a)): again this case splits in two subcases:
  - $rank_j(u_i^j) = rank_j(x_{j-1}) = k$: then, there exist $a, b \in upredsP$ such that $rank_j(a(x_{j-1})) = k$ and $(a, b) \in connpr_{G_j}(x_{j-1}, v_i^j)$; this, implies that $rank_{j-1}(a(x_{j-1})) = k$ and $(a, b) \in connpr_{G_{j-1}}(y_{j-1}, v_i^{j-1})$(see figure); from the fact that (x_{j-1}, y_{j-1}) is a redundancy pair, it results that: $rank_{j-1}(a(y_{j-1})) = k$ and together with $(a, b) \in connpr_{G_{j-1}}(y_{j-1}, v_i^{j-1})$, it results $rank_{j-1}(b(v_i^{j-1})) = k$, thus $rank_{j-1}(v_i^{j-1}) = k = rank_{j-1}(u_i^{j-1})$;
  - $rank_j(u_i^j) > rank_j(x_{j-1})$: similar to the previous case;

$$\begin{tikzcd}
CS_{j-1}: \arrow[d] & CS_j: \arrow[d] \\
x_{j-1} \arrow[r, dashed] \arrow[d] & x_{j-1} \arrow[d] \\
y_{j-1} \arrow[ur, dashed] \arrow[d] & u_i^j = (u_i^{j-1})_{y_{j-1}||x_{j-1}} \arrow[d] \\
u_i^{j-1} \arrow[d] & v_i^j = (v_i^{j-1})_{y_{j-1}||x_{j-1}} \\
v_i^{j-1} &
\end{tikzcd}$$

a) $y_{j-1} \leqslant u_i^{j-1}$

$$\begin{tikzcd}
CS_{j-1}: \arrow[d] & CS_j: \arrow[d] \\
u_i^{j-1} \arrow[d] & u_i^j = u_i^{j-1} \arrow[d] \\
x_{j-1} \arrow[r, dashed] \arrow[d] & x_{j-1} \arrow[d] \\
y_{j-1} \arrow[ur, dashed] \arrow[d] & v_i^j = (v_i^{j-1})_{y_{j-1}||x_{j-1}} \\
v_i^{j-1} &
\end{tikzcd}$$

$b) u_i^{j-1} \leqslant x_{j-1}$

Figure 3.15: Backward preservation of equal rankings for (u_i^j, v_i^j) pairs.

- b) $u_i^{j-1} \leqslant x_{j-1} < y_{j-1} < v_i^{j-1}$ (Figure 3.15 b)): in this case, $rank_j(u_i^j) = rank_j(x_{j-1}) = k$. Then there exists $a, b, c \in upreds(P)$ such that $rank_j(a(u_i^j)) = k$, $(a, b) \in connpr_{G_j}(u_i^j, x_{j-1})$, and $(b, c) \in connpr_{G_j}(x_{j-1}, v_i^j)$.

  From the figure, one can see that $rank_{j-1}(x_{j-1}) = k$. As (x_{j-1}, y_{j-1}) is a redundancy pair, $rank_{j-1}(y_{j-1}) = k$. If $(a, b) \in connpr_{G_{j-1}}(u_i^{j-1}, x_{j-1})$ and $rank$ $(a(u_i^{j-1}))$, then $(a, b) \in connpr_{G_{j-1}}(u_i^{j-1}, y_{j-1})$ (again from the redundancy of (x_{j-1}, y_{j-1})).

From lemma 3.4.25 and 3.4.26 it results that there is a sequence of nodes $(u_{k_i}, v_{k_i})_{0 \leqslant i \leqslant n}$ such that $u_{k_i} = v_{k_{i+1}}$, and $rank(u_{k_i}) = rank(v_{k_i}) = r$, for every $0 \leqslant i \leqslant n$ and the path $path(u_{k_0}, v_{k_n})$ is infinite. As $rank(u_{k_0}) = rank(v_{k_0}) = r$, this implies that there is a path of infinite length of rank r in G_0. As G_0 reflects the dependencies between the atoms in the forest model, this is equivalent to the existence of an atom in the forest model which is motivated by an infinite chain of atoms in the model. This contradicts with the fact that any atom in an open answer set is justified in a finite number of steps[48, Theorem 2]. Thus, the construction of a complete clash-free completion structure from a forest model does terminate.

3.4.7 Complexity

Proposition 3.4.10 implies the following complexity result for our algorithm $\mathcal{A}_3$:

Proposition 3.4.27. *$\mathcal{A}_3$ runs in the worst case in non-deterministic exponential time.*

However, one can transform the algorithm to a deterministic procedure which can be executed in exponential time. The deterministic procedure which we will call $DET - \mathcal{A}_3$ consists in constructing an AND/OR extended forest with depth double in the size of the largest depth encountered when running the nondeterministic algorithm. At odd levels, there are OR nodes with unexpanded content (they contain just the constraints imposed by their predecessor or the predicate checked to be satisfiable in case of one root node and an empty set for the other root nodes), while at even levels, there are AND saturated nodes which are 'realizations' of their predecessor, i.e., they (together with their outgoing arcs and direct successors) describe a possible way to expand the predecessor node. For every OR node, each of its 'realizations' spawns a new copy of the graph G. We call such a structure an AND/OR completion structure.

Blocking and caching are applied by considering only pairs of AND nodes in the extended forest. For simplicity, we consider the stricter caching condition used in the proof of lemma 3.4.10.

A leaf of the AND/OR extended forest is labeled with *false* if it is unexpanded and it is not a blocked or cached node, with *true* if it is a blocked node, and it is labeled with the label of its corresponding caching node otherwise (if the leaf is cached node). A predicate p is satisfiable in such a structure if the root node of every tree in the structure evaluates to *true*. In this case the structure is called a *successful* AND/OR completion structure. The evaluation can be done straightforwardly as the evaluation of a caching node does not depend on the evaluation of its cached done due to the fact that, like before, the extended forest is constructed in a depth-first manner.

Proposition 3.4.28. *$DET - \mathcal{A}_3$ is sound, complete, and runs in the worst case in deterministic exponential time.*

Proof Sketch.

Soundness

From a successful AND/OR completion structure we construct a clash-free complete completion structure.

For every pair (S, r) for which there is at least a node x in the extended AND/OR forest with $\text{CT}(x) = S$ and $rank(x) = r$, let $x_{(S,r)}$ be the 'witness' AND node for (S, r), i.e. the node which is expanded and which will be a caching node in every caching pair of nodes with profile (S, r).

Assume that the root node of every tree in the successful deterministic structure evaluates to *true*. For every OR node, pick a successor which is true and add it to the completion structure in construction. For every AND node y, if it is blocked or expanded, simply add it to structure in construction. If y is cached and $x_{(\text{CT}(y),rank(y))}$ has not been added to the completion structure in construction, copy $x_{(\text{CT}(y),rank(y))}$ instead of y to the structure.

Completeness

From a clash-free complete completion structure we construct a successful AND/OR completion structure. At every OR node we always add as the first successor of the node the unit completion structure chosen when constructing the clash-free complete completion structure. It is easy to see that a deterministic structure constructed in such a way is successful.

Complexity

Using a similar argument as in lemma 3.4.10 one can show that the size of a successful deterministic structure is still exponential in the size of P: clearly the depth of the AND/OR extended forest is still exponential in the size of P (it is double the depth of the deepest complete completion structure constructed using the nondeterministic algorithm) and the caching argument still holds.

Thus, satisfiability checking of a unary predicate p w.r.t. a FoLP P can be evaluated in exponential time in the size of P. This, together with the fact that the same task is EXPTIME-hard (see D3.2 [51]), implies that the problem is EXPTIME-complete. With this we close an existing gap regarding the complexity of reasoning with FoLPs and f-hybrid knowledge bases.

Proposition 3.4.29. *Satisfiability checking of a unary predicate p w.r.t. a FoLP P is* EXPTIME-*complete.*

Finally, this result translates in a similar result concerning f-hybrid knowledge bases. f-hybrid knowledge bases have been introduced in deliverable D3.2 [51] as a tight combination between FoLPs and the DL $\mathcal{SHOQ}$. As $\mathcal{SHOQ}$ is known to be EXPTIME-complete, it follows that f-hybrid knowledge bases are EXPTIME-hard. We also know that reasoning with f-hybrid knowledge bases can be reduced to reasoning with FoLPs. Thus, it can be deduced that they are EXPTIME-complete.

Proposition 3.4.30. *Satisfiability checking of a unary predicate p/concept C w.r.t. an f-hybrid knowledge base KB is* EXPTIME-*complete.*

3.5 Conclusions: reasoning with FoLPs and beyond

3.5.1 Discussion

We presented an optimal worst case algorithm for reasoning with Forest Logic Programs. The algorithm exploits the forest model property of the fragment and builds on techniques introduced in the previous years of the project, like pre-computing all possible building blocks of a model in the form of trees of depth 1. However due to the introduction of new termination techniques, the worst case complexity drops one exponential level compared to its previous variants: from double exponential time to exponential time.

Thus, while FoLPs can simulate reasoning with the DL $\mathcal{SHOQ}$, and allow for additional features like a minimal model based semantics and a cleaner syntax, the worst case reasoning complexity stays the same. As reasoning with the tight combination of FoLPs and $\mathcal{SHOQ}$ ontologies, f-hybrid knowledge bases, can be reduced to reasoning with FoLPs, the algorithm can be employed also for reasoning with f-hybrid knowledge bases. It also establishes that satisfiability checking w.r.t. f-hybrid knowledge bases is EXPTIME-complete.

While not mentioned explicitly here, the algorithm $\mathcal{A}_2$ described in D3.3 [40] also identifies and eliminates so-called *redundant unit completion structures*: these are structures which are strictly less general than others, so they can always be replaced in a model with other more general structures. Assuming that the new algorithm $\mathcal{A}_3$ also employs this technique of discarding redundant unit completion structures, it addresses computational redundancy issues across three orthogonal axis: (i) *local redundancy*: eliminating redundant unit completion structures eliminates local redundancy, i.e. redundancy among the successors of a single node, (ii) *redundancy along a path*: achieved by means of the redundancy rule, and (iii) *redundancy across paths*: achieved by means of the caching rule.

3.5.2 Related Work

Datalog$^{\pm}$ [15, 16] is an extension of Datalog which can simulate some DLs from the DL-Lite family [19]. The extension consists in allowing a special type of rules with existentially quantified variables in the head, called tuple generating dependencies (TGDs). Note that our free rules are different from TGDs, as they allow for universally quantified variables which do not appear in the body of the rule to appear in the head.

The formalism is undecidable in the general case. Like in the case of OASP, several syntactical restrictions have been imposed on the shape of TGDs in order to regain decidability. Two such restrictions are: (1) every rule should have a guard, an atom which contains all variables in the rule body, giving rise to *guarded Datalog*$^{\pm}$, and (2) every rule should have a singleton body atom, giving rise to *linear Datalog*$^{\pm}$. The guardedness condition has been relaxed to *weakly-guardedness*, where the weak guard has to contain only the variables in the body that appear in so-called affected positions, positions where newly invented values can appear during reasoning [14]. Reasoning relies on a proof technique from database theory, the chase algorithm, which repairs databases according to the set of dependencies.

Some further generalizations to the guarded fragment of Datalog$^{\pm}$ are so-called *sticky sets* of TGDs [17], *weakly-sticky* sets of TGDS, and *sticky-join* sets of TGDs [18] which generalize both sticky sets and linear TGDs. All these fragments are defined by imposing restrictions on multiple occurrences of variables in rule bodies. The syntactical restrictions on rules bodies are orthogonal to the ones we imposed for achieving decidability on FoLPs: neither Datalog$^{\pm}$ rules are enforced to have a tree-shape like FoLPs, nor variables in FoLP rules have to fulfill the conditions required for the different sets of TGDs to belong to one of the previously mentioned decidable fragments of Datalog$^{\pm}$. TGDs do not contain negation. However, so-called stratified normal TGDs have been introduced, which are TGDs whose body atoms can appear in a negated form together with a semantics in terms of canonical models. FoLPs support full negation as failure (under the stable models semantics).

3.5.3 Future Work

Another interesting fragment of Open Answer Set Programming which has been proved to be decidable is *Conceptual Logic Programs under the Inverted World Assumption*. The fragment has the *tree model property* and can simulate the description logic $\mathcal{SHIQ}$. Conceptual Logic Programs (CoLPs) are FoLPs in which constants are disallowed. The Inverted World Assumption refers to the fact that the signature of the programs is such that for every binary f, there exists an inverse binary predicate f^i; semantically, the assumption refers to the following condition: for every Open Answer Set (U, M), and for every binary predicate f it holds that: $f^i(x, y) \in M$ iff $f(y, x) \in M$. IWA is equivalent to adding $f(X, Y) \leftarrow f^i(Y, X)$ and $f^i(Y, X) \leftarrow f(X, Y)$ to the original program and evaluating it under the usual semantics.

In [48] satisfiability checking w.r.t. IWA has been reduced to checking emptiness of a two-way alternating tree automata. However, there is no practical algorithm for dealing with such programs. We plan to investigate how the algorithm $\mathcal{A}_3$ can be adapted for reasoning with CoLPs under IWA: the non-trivial part of such an adaptation consists in dealing with the IWA. A natural step in this direction is to generalize the notion of UCS to UCS under IWA: a UCS would be still an extended tree with depth 1, but the root can have an outgoing arc to its predecessor. Matching UCSs consists in this case in a double match (between 2 pairs of successor nodes). We conjecture that any arbitrary tree model of a CoLP under IWA can be reduced to an *exponential* size structure which can then be unraveled to a finite bounded size model by applying similar transformations to the ones used for reducing a FoLP model in the completeness proof. However, it is questionable whether the classical tableau approach still works. In the presence of backwards arcs in the forest (from nodes to their predecessors), the rank of a node/atom is no longer a function of predecessor nodes: it might be needed to traverse the whole structure in order to compute it. Figure 3.16 gives an example where the rank of a node depends on the content of one of its neighbors: both $x \cdot 1$ and $x \cdot 2$ are the successors of x in a tree, while the arcs depicted in the figure are arcs in the atom dependency graph of the constructed model. Before fully expanding $x \cdot 1$ there is no path in G from a predicate with argument x to a predicate with argument $x \cdot 2$, and thus the rank of $x \cdot 2$ is $d + 1$. However after fully expanding $x \cdot 1$, the rank of $x \cdot 2$ is d.

$$\begin{tikzcd}
depth = d \arrow[dr, dashed] & & & & \\
 & p(x) \arrow[dl] & r(x) \arrow[drr] & & \\
q(x_1) \arrow[ur] & & & s(x_2) & rank = d \arrow[l, dashed]
\end{tikzcd}$$

Figure 3.16: Computing the rank of a node in the presence of backward arcs in the tree

One possibility would be to generate a completion structure with size within the computed bound and check whether the completion structure is clash-free and complete. Formal proofs for reasoning with this fragment are subject of future work.

Further on, we plan to formally define the fragment of FoLPs under the IWA and to investigate reasoning with this fragment. Such a fragment would allow the simulation of the expressive DL $\mathcal{SHOIQ}$.

3.6 Overview of Notions

In this section we provide a list with the main notions related to the algorithms described in this chapter (which were not previously introduced in Section 3.1) and pointers to the pages where these notions were formally introduced.

Completion Structure: a representation of a forest model in construction. It contains an extended forest (the universe of the model), two labeling functions CT (content: points out which atoms are/are not in the model) and ST (status: which atoms/nodes have been expanded), and an atom dependency graph (p. 43).

Initial Completion Structure for Checking Satisfiability of p w.r.t. P: a completion structure which imposes a single constraint on the model in construction: that some atom $p(\varepsilon)$ has to be part of the model, where ε is an anonymous individual or one of the constants in the program (p. 44).

Node Saturation: A node is saturated when all unary predicates in the program appear either in a positive or a negative form in its content and they have all been expanded using the expansion rules introduced by algorithm $\mathcal{A}_1$ and all binary predicates appear either in a positive or a negative form in the content of each of its successors and they have all been expanded using the expansion rules introduced by algorithm $\mathcal{A}_1$ (p. 45).

Unit Completion Structure (UCS) with root ε: a completion structure in which the extended forest is an extended tree of depth 1; the root ε of the distinguished tree in the extended tree is saturated, all other predicates in the contents of nodes are unexpanded (p. 46).

local satisfiability: a unit completion structure with root ε locally satisfies a set of (possibly negated) unary predicates S iff $S \subseteq \textsc{ct}(\varepsilon)$ (p. 47).

matchability: an unexpanded node is matchable with a unit completion structure UC iff UC locally satisfies the content of the node and the constraints imposed by the UCS on nodes which are constants are not in contradiction with the current contents of those nodes (p. 47).

$expand_C S(x, UC)$: the operation which expands an $\mathcal{A}_2$-completion structure by adding a new UCS which matches an unexpanded node in the structure (p. 48).

Match Rule: the rule used by algorithms $\mathcal{A}_2$/$\mathcal{A}_3$ which applies the $expand_C S(x, UC)$ operation in order to expand an unexpanded node (p. 48).

Blocking Rule: the rule used by all three algorithms to stop the successful expansion of a branch by reusing some computations performed to expand a node on the same branch (p. 48).

Redundancy Rule: two rules used by the algorithms $\mathcal{A}_1$ and $\mathcal{A}_3$ to stop the unsuccessful expansion of a branch (p. 49, p. 53).

Caching Rule: a new rule introduced by the algorithm $\mathcal{A}_3$ to stop successful expansion of a branch by reusing some computations performed to expand a node on another branch (p. 54).

Complete Completion Structure: a completion structure which can no longer be expanded (p. 50).

Clash-free Completion Structure: a completion structure which contains no redundancy nodes (p. 50).

Chapter 4

Integration of Production Rules and Ontologies via Transaction Logic with Partially Defined Actions

4.1 Motivation

Production systems (PS) are one of the oldest knowledge representation paradigms that are still popular today. Production systems are widely used in biomedical information systems, to enforce constraints on databases, to model business processes, accounting, etc.

Such systems consist of a set of *production rules* that rely on forward chaining reasoning to update the underlying database, called *working memory*. Traditionally, PS have had only operational semantics, where satisfaction of rule conditions is checked using pattern matching, and rule actions produce assertion and deletions of facts from the working memory. PS syntax and semantics have been standardized as W3C's Production Rule Dialect of the Rule Interchange Format (RIF-PRD) [99] . The RIF-PRD specification has a number of limitations, however. First, it omits certain important primitives that are found in many commercial production systems such as IBM's *JRules*[58]. The *FOR*-loop and the *while*-loop constructs are examples of such an omission. Second, RIF-PRD still does not integrate with ontologies [6, 54]. Here, by ontology we mean a formal representation of a domain of interest, expressed in terms of concepts and roles, which denote classes of objects and binary relations between classes of objects, respectively.

To illustrate the need for ontology integration, consider a set PS that keeps a number of clinical databases that are compliant with the health insurance regulations. The clinical record of each patient together with other data must be accessible by all the clinics in the network. This needs a shared vocabulary that, in this case, is defined in a shared DL ontology. However, each PS can have extra concepts outside the ontology, which are meant for local use only.

Example 4.1.1. *The following production rules state that* (i) *if a doctor D requests a DNA test T to be performed for patient P, then the system records that P is taking the test T; and* (ii) *patients getting a DNA test must not be considered unhealthy.*

r_1: `Forall` D, P, T: `if` $\mathit{requested}(D, P, T) \wedge \mathit{dnaT}(T)$ `then` $\mathit{Assert}(\mathit{takesT}(P, T))$
r_2: `For` P, T: $\mathit{takesT}(P, T) \wedge \mathit{dnaT}(T)$ `do` $\mathit{Retract}(\mathbf{neg}\, \mathit{healthy}(P))$

The DL ontology that defines the shared concepts and implements different constraints is as follows

$$\mathit{flu} \sqsubseteq \mathbf{neg}\, \mathit{healthy} \qquad \mathit{dnaT} \sqsubseteq \mathbf{neg}\, \mathit{virusT} \qquad \exists \mathit{takesT}.\mathbf{neg}\, \mathit{virusT} \sqsubseteq \mathit{healthy}$$

The DL axioms say that a patient with a flu is not a healthy patient, that DNA tests do not search for viruses, and that if a person is taking a test not related with any virus disease, then we can conclude that she is healthy. Note that here we are using strong negation $\mathbf{neg}$ *[82], a weaker form of classical negation that does not add expressivity to the logic, but makes knowledge representation more natural. The* `Forall` *construct in* r_1 *should not be confused with the* `For` *construct in* r_2*. The former is just a way RIF-PRD declares variables used in the body of a rule. The latter is a FOR-loop extension found in commercial systems, but not in RIF-PRD.* □

The complexity of the regulations in our example makes it difficult to determine whether executing a production rule leaves the database in a compliant state. Suppose we have the following initial database $\mathsf{WM}_0 = \{\mathsf{requested}(\text{Smith, Laura, pcr}), \mathsf{flu}(\text{Laura}), \mathsf{dnaT}(\text{pcr})\}$. This example raises several question: *(i)* how do we interpret the retraction executed by r_2, where P is instantiated with $Laura$, given that $\mathbf{neg}\, \mathsf{healthy}(\text{Laura})$ is inferred by the ontology, *(ii)* how to interpret the rule conditions of r_1 and r_2 given the open world semantics of DL, and *(iii)* how do we treat the inconsistency that results after execution of rule r_1 in WM_0? (Observe that in the state resulting from execution of r_1 in WM_0 we can infer $\mathsf{healthy}(\text{Laura})$ and $\mathbf{neg}\, \mathsf{healthy}(\text{Laura})$.)

To answer these questions we need to define a precise semantics (both model-theoretic and computational) to the combination of rules, ontologies, and production systems,

Our contribution in this chapter is three-fold: *(i)* a new semantics for production systems augmented with *DL* ontologies that includes looping-rules, and can handle inconsistency; *(ii)* a sound embedding of the combination of PS and rule-based ontologies into *Transaction Logic with partially defined actions* (abbr., $\mathcal{TR}^{PAD}$) [89], which provides a model-theoretic semantics to the combination; *(iii)* an extension of $\mathcal{TR}^{PAD}$ with default negation under a variant of the well-founded semantics [96] for $\mathcal{TR}^{PAD}$.

Our formalization is significantly more general than RIF-PRD or other existing formalizations of production rules in that it supports wider ontology integration and covers important extensions that exist in commercial systems such as the aforesaid *FOR*-loop.

We opted for $\mathcal{TR}$ because it is a purely logical formalism that combines declarative and procedural knowledge. $\mathcal{TR}^{PAD}$ allows actions to be defined as *sequences* of simpler constituent actions, and its—model-theoretic—semantics is defined over *sequences of states*. These two features makes the path from the operational semantics to the model-theoretic semantics shorter and clearer.

The rest of this chapter is organized as follows. Section 4.2 briefly surveys previous results on the combination of PS and ontologies, and on the reduction of PS to formalisms with denotational semantics. Section 4.3 presents the necessary background on first order logic and description logic. Section 4.4 introduces an operational semantics for production systems augmented with *DL* ontologies. Section 4.5 augment $\mathcal{TR}^{PAD}$ with default negations, and provides a well-founded semantics for such extension. Section 4.6 provides a reduction from the semantics proposed here to $\mathcal{TR}^{PAD}$ and presents soundness results for this reduction. Section 4.7 summarizes the approach. Proofs of the main results and further details are found in [88].

4.2 Related Work

In this section we compare our approach with other literature on the declarative semantics for production systems and on the operational and declarative semantics for the combination of PS and ontologies. In Deliverable D3.3 [40] and in [90, 28] an operational and model-theoretic semantics to the combination of PS and ontologies is provided. The model-theoretic semantics is given by an embedding of PS into fix-point logic. However, they cannot handle looping rules, their semantics cannot handle inconsistencies, their interpretation of retraction of DL facts is not intuitive since a fact can remain true after being deleted, and their reduction to a declarative formalism is considerably more complex than the one presented here. In [67, 102], the goal is to devise languages for unifying some aspects of active rules, logic rules, and production systems. They do not deal with considerably more complex standard languages such as production systems augmented with ontologies and looping rules. In particular, [67, 102] do not show how to embed production systems into those languages, although they provide some examples showing how typical production rules can be expressed in their language. In [85] the authors only allow a very restricted type of production systems: stratified PS. Such PS are much weaker that the ones formalized here, and again, they do not consider ontologies. In addition, they do not tackle the problem of the integration with ontologies. In [26, 7], the authors reduce the semantics of PS to logic programming (LP). Their reduction is considerably more complex and less compact than ours—it results in an infinite number of rules. In addition, they use stable models semantics which has much higher computational complexity than the well founded semantics used here. Given the complexity of such a reduction, the proposed integration with LP ontologies is not ideal, since the ontology needs to be transformed with state arguments and auxiliary predicates. In addition, neither of them allow looping rules. Finally, [62] presents a new formalism that combines some aspects of logic rules and production rules. However, negation in rule conditions[1] and looping rules are not allowed. Furthermore, their embedding into Horn Logic is less clear and compact than our embedding in $\mathcal{TR}^{PAD}$.

4.3 Preliminaries

4.3.1 First Order Logic

The alphabet of a first order language $\mathcal{L}$ includes a countably infinite disjoint sets of variables $\mathcal{V}$, constant symbols $\mathcal{C}$, and predicate symbols $\mathcal{P}$. A *term* is a constant or a variable. Each predicate symbols has an *arity* n, which is a non-negative integer. We will not include function symbols since nether description logics nor production systems use them.[2]

Our language, $\mathcal{L}$, includes all the usual first-order operators $\vee, \wedge, \neg, \forall, \exists, \rightarrow, =$ plus the symbol $\mathbf{neg}$, which represents the *explicit negation* (also sometimes called *strong* negation) [82]. The $\mathbf{neg}$ symbol applies only to atoms. In the actual use in this paper, $\neg$ will appear only in ontologies, while $\mathbf{neg}$ will be used both in ontologies and in production rules. In a certain sense, which will be made clear later, $\mathbf{neg}\, f$ will imply $\neg f$, but not vice versa.

Formulas are defined recursively as usual in first-order logic. A literal is either an atomic formula f, or a formula of the form $\mathbf{neg}\, f$ where f is an atomic formula. Atoms are also called **positive literals**. A **negative literal** is an atom preceded with the symbol $\mathbf{neg}$ (e.g., $\mathbf{neg}\, p(X)$).

Since later on we will be integrating ontologies with production systems, we will need to use Herbrand domains and the unique name assumption. Therefore, we will use the Herbrand semantics from the outset. As is well-known, this semantics is equivalent to the general one for universal clausal form. The semantics defines *semantic structures*. The domain of a Herbrand semantic structure is called the Herbrand universe $\mathcal{U}$; in our restricted case it is just the set of all constants $\mathcal{C}$ in the language $\mathcal{L}$. The Herbrand base $\mathcal{B}$ is a set of all ground literals in the language, which includes the $\mathbf{neg}$-negated literals of the form $\mathbf{neg}\ f$. Note that the Herbrand universe and Herbrand base are infinite, fixed, and depend only on the language $\mathcal{L}$.

Definition 4.3.1 (Semantic Structure). *A semantic structure $\mathcal{I} = \langle \mathcal{U}, \mathbf{B}, \sigma \rangle$ consists of the following items:*

- *A subset $\mathbf{B}$ of $\mathcal{B}$.*

[1] The authors informally claim that negation could be added, but they do not provide formal details

[2] Many production systems do use built-in and external functions and so do some DLs. However, this does not bring any new or interesting issues in our context, so we disregard functions in order to simplify the exposition.

- *a* variable assignment, σ, *that maps each variable in* $\mathcal{V}$ *to a domain element in* $\mathcal{U}$.
- σ *is a* ***variable assignment****, i.e., a mapping* $\mathcal{V} \longrightarrow \mathcal{U}$. $\square$

We now define satisfaction of formulas by semantic structures. If ϕ is a formula of $\mathcal{L}$, $\mathcal{I}$ is a structure for $\mathcal{L}$, then satisfaction of ϕ in $\mathcal{I}$ is denoted $\mathcal{I} \models \phi$. Given a structure $\mathcal{I}$, and a term (i.e., constant or variable) t, we define $t^{\mathcal{I}}$ as: $t^{\mathcal{I}} = \sigma(t)$ if t is a variable and $t^{\mathcal{I}} = t$ if t is a constant. Note that this implies the *unique name assumption* (UNA). That is, if $c_1, c_2 \in \mathcal{C}$ are two distinct constants then that $c_1^{\mathcal{I}} \neq c_2^{\mathcal{I}}$.

The relation $\models$ is defined recursively as follows:

- If t_1 and t_2 are terms, then $\mathcal{I} \models t_1 = t_2$ if and only if $t_1^{\mathcal{I}}$ is the same element as $t_2^{\mathcal{I}}$.
- If P is an n-place predicate letter in $\mathcal{L}$ and $t_1, \ldots, t_n$ are terms, then $\mathcal{I} \models P(t_1 \ldots t_n)$ (respectively, $\mathcal{I} \models \mathbf{neg}\, P(t_1 \ldots t_n)$) if and only if $P(t_1^{\mathcal{I}} \ldots t_n^{\mathcal{I}}) \in \mathbf{B}$ (respectively, $\mathbf{neg}\, P(t_1^{\mathcal{I}} \ldots t_n^{\mathcal{I}}) \in \mathbf{B}$).
- $\mathcal{I} \models \neg\phi$ if and only if it is not the case that $\mathcal{I} \models \phi$.
- $\mathcal{I} \models (\phi \wedge \psi)$ if and only if $\mathcal{I} \models \phi$ and $\mathcal{I} \models \psi$.
- $\mathcal{I} \models (\phi \vee \psi)$ if and only if $\mathcal{I} \models \phi$ or $\mathcal{I} \models \psi$.
- $\mathcal{I} \models \forall v : \phi$ if and only if $\mathcal{I}' \models \phi$ for structure $\mathcal{I}'$ that agrees with $\mathcal{I}$ except possibly on the variable v.
- $\mathcal{I} \models \exists v : \phi$ if and only if $\mathcal{I}' \models \phi$ for some structure $\mathcal{I}'$ that agrees with $\mathcal{I}$ except possibly on the variable v.

A formula ϕ is *valid*, if $\mathcal{I} \models \phi$, for every structure $\mathcal{I}$.

A formula ϕ is *satisfiable* if there is a structure $\mathcal{I}$ such that $\mathcal{I} \models \phi$.

If Γ is a set of sentences and if $\mathcal{I} \models \phi$ for each sentence ϕ in Γ, then we say that $\mathcal{I}$ is a *model* of Γ. So a set of sentences is satisfiable if it has a model.

We say that Γ *entails* ϕ, written $\Gamma \models \phi$, if and only if $\Gamma \cup \{\neg\phi\}$ is not satisfiable.

4.3.2 Description Logics

In this Section we briefly review the basic notions from Description Logic (DL) that we will use in this chapter. Details can be found in [6].

Description Logic is a family of knowledge representation formalisms that provide a syntax and a model-theoretic semantics for a compact representation of information. A DL knowledge base has two parts: the *T-box*, with terminological knowledge, which consists of a number of class definitions, and the *A-box*, which consists of assertions about actual individuals.

Concept axioms in the *T-box* are of the form $C \sqsubseteq D$ (meaning the extension of C is a subset of the extension of D; D is more general than C) or $C \equiv D$ (where $C \equiv D$ is interpreted as $C \sqsubseteq D$ and $D \sqsubseteq C$) with C and D (possibly complex) descriptions. Given a T-box axiom of the form $C \sqsubseteq D$, the concept C is called the *body*, and D is called the *head* of the axiom.

Descriptions and T-box axioms can be understood as formulas of first-order logic with one free variable and closed universal formulas respectively. For example, the description $A \sqcap \neg B \sqcap \exists R.C$ corresponds to the formula $A(x) \wedge \neg B(x) \wedge \exists y.(R(x,y) \wedge C(y))$. Therefore, the semantics of DL can be given by its translation to FOL. Details can be found in [6].

In Section 4.6, we will use DLs that can be embedded into Logic Programming (LP). In recent years, the relationship between DLs and Logic Programming (LP) has attracted much interest and several LP-expressible DLs have been proposed [3, 56, 78, 79, 33, 66, 52]. In particular, [52] defines a class of DLs called *Datalog-rewritable DLs*. This class is interesting in our setting because reasoning with DLs in such a class can be reduced reasoning with Datalog programs.

Definition 4.3.2 (Datalog-rewritable). *A DL $\mathcal{D}$ is* **Datalog-rewritable** *if there is a transformation* dtg *from $\mathcal{D}$ to Datalog programs such that for any knowledge base $(\mathcal{T}, \mathcal{A})$ in $\mathcal{D}$, where $\mathcal{T}$ is a T-box and $\mathcal{A}$ is an A-box, the following holds: For any concept or role name Q, and an individual i in $(\mathcal{T}, \mathcal{A})$,*

$$(\mathcal{T}, \mathcal{A}) \models Q(i) \;\; \textit{iff} \;\; \mathsf{dtg}((\mathcal{T}, \mathcal{A})) \models Q(i)$$

We say that dtg *is* **modular** *if*

$$\mathsf{dtg}((\mathcal{T}, \mathcal{A})) = \mathsf{dtg}(\mathcal{T}) \cup \mathsf{dtg}(\mathcal{A}) \qquad \square$$

One Datalog-rewritable DL is $\mathcal{LDL}^+$ [52]. For concreteness, in Section 4.6 we will work with this DL, but our results do not depend on a particular choice of a Datalog-rewritable DL. $\mathcal{LDL}^+$ is defined as shown in Figure 4.1, by restricting the shape of the axioms in the T-box. Further details and a reduction to Datalog can be found in [52].

An $\mathcal{LDL}^+$ KB is a pair $(\mathcal{T}, \mathcal{A})$, where $\mathcal{T}$ is a finite T-box and $\mathcal{A}$ is a finite A-box such that

- $\mathcal{T}$ is a set of *terminological axioms* of the form $C \sqsubseteq D$, where C is a body concept and D is a head concept; and *role axioms* $E \sqsubseteq F$, where E is a body role and F is a head role.

- $\mathcal{A}$ is a set of assertions of the form $D(o)$ and $F(o_1, o_2)$ where D is a head concept and F is a head role.

Body Roles				**Head Roles**			
E_1, E_2	$\rightarrow$	P	(Role name)				
		E_1^-	(Inverse)				
		$E_1 \circ E_2$	(Sequence)	F_1, F_2	$\rightarrow$	P	(Role name)
		E_1^+	(Transitive Closure)			F_1^-	(Inverse)
		$E_1 \sqcap E_1$	(Conjunction)			$F_1 \sqcap F_2$	(Conjunction)
		$E_1 \sqcup E_2$	(Disjunction)			$\top^2$	(Top)
		$\top^2$	(Top)				
		$\{o, o\}$	(Nominals)				

Basic Concepts				**Head concepts**			
C_1, C_2	$\rightarrow$	A	(Concept name)				
		$\exists E_1.C_1$	($\exists$ Restriction)				
		$\leqslant nE_1.C_1$	($\leqslant$ Restriction)				
		$C_1 \sqcup C_2$	(Disjunction)	D	$\rightarrow$	C_1	(Basic Concept)
		$C_1 \sqcap C_2$	(Conjunction)			$\forall E_1.C_1$	($\forall$ Restriction)
		$\{o\}$	(Nominals)				
		$\top$	(Top)				

Figure 4.1: $\mathcal{LDL}^+$ Syntax

Example 4.3.3. *[52] Suppose we have the following knowledge base* $(\mathcal{T}, \mathcal{A})$*:*

$$\mathcal{T} = \begin{cases} \leqslant 2PapersToRev \sqsubseteq OverL \\ OverL \sqsubseteq \forall Superv^+.OverL \end{cases} \mathcal{A} = \begin{cases} Superv(a,b) \\ Superv(b,c) \end{cases}$$

The first two axioms say that someone with more than two papers to review is overloaded and that an overloaded person causes all the supervised persons to be overloaded as well. The statements in the A-box defines the supervision hierarchy. □

We conclude this brief introduction of $\mathcal{LDL}^+$ with a review of its relationship to OWL 2 fragments.

- **OWL 2 EL**. This fragment correspond the DL $\mathcal{EL}^{++}$. This fragment and $\mathcal{LDL}^+$ do not subsume each other. Not all $\mathcal{EL}^{++}$ axioms can be expressed in $\mathcal{LDL}^+$ but those that do not contain $\bot$, concrete domains, and existential quantification in axioms' right side. On the other hand, $\mathcal{LDL}^+$ has many constructs that $\mathcal{EL}^{++}$ does not allow: number restrictions, inverse, general sequence of roles, role conjunction, role disjunction, etc.

- **OWL 2 QL**. This fragment correspond to the DL-*Lite* family. Again, this fragment neither subsumes nor is subsumed by $\mathcal{LDL}^+$. DL-Lite axioms that have no negation and existential quantification on the right side are also axioms in $\mathcal{LDL}^+$, but $\mathcal{LDL}^+$ has constructs that are not expressible in DL-*Lite*, such as role sequence.

- **OWL 2 RL**. This fragment is a strict subset of $\mathcal{LDL}^+$.

4.4 Combining RIF -Production Systems and Ontologies

In this section we propose a new semantics for the combination of production systems and *arbitrary* DL ontologies. This approach follows the outline of [90], but includes looping rules, it can handle inconsistencies produced by the system, and it gives a more intuitive semantics to the retraction of DL facts.

The alphabet of a language $\mathcal{L}_{\mathsf{PS}}$ for a production system is defined the same way as in the case of DL except that now the set of all predicates $\mathcal{P}$ is partitioned into two countably infinite subsets, $\mathcal{P}_{PS}$ and $\mathcal{P}_{DL}$. The latter will be used to represent predicates occurring in the ontology. A term is either a variable or a constant symbol and, to avoid unnecessary distractions, we will leave out the various additional forms allowed in RIF, such as frames and the RIF membership and subclass relations ($o\#t$, $t\#\#s$). However, they can easily be added without increasing the complexity of the problem. A **atomic formula** is a statement of the form $p(t_1 \dots t_n)$, where $p \in \mathcal{P}$. A literal is either an atom, a formula of the form $\mathbf{neg}\, f$ where f is a $\mathcal{P}_{DL}$-atom, or a formula of the form $\neg f$ where f is a $\mathcal{P}_{PS}$-atom.

4.4.1 Syntax

The alphabet of a language $\mathcal{L}_{\mathsf{PS}}$ for a production system is defined the same way as in the case of DL except that now the set of all predicates $\mathcal{P}$ is partitioned into two countably infinite subsets, $\mathcal{P}_{PS}$ and $\mathcal{P}_{DL}$. The latter will be used to represent predicates occurring in the ontology. A term is either a variable or a constant symbol and, to avoid unnecessary distractions, we will leave out the various additional forms allowed in RIF, such as frames and the RIF membership and subclass relations ($o\#t$, $t\#\#s$). However, they can easily be added without increasing the complexity of the problem.

Definition 4.4.1 (Atomic Formulas)**.** *Let $p \in \mathcal{P}$ be a predicate and $t_1, \dots, t_n$ be terms. A RIF atomic formula is a statement of the form $p(t_1 \dots t_n)$.* $\square$

A **literal** is either an atom, a formula of the form $\mathbf{neg}\, f$ where f is an $\mathcal{P}_{DL}$-atom, or a formula of the form $\neg f$ where f is an $\mathcal{P}_{PS}$-atom.

Definition 4.4.2 (Condition Formula)**.** *A **condition formula** ϕ has one of the following forms:*

- *a literal l,*
- *$\phi_1 \wedge \phi_2$, where ϕ_1 and ϕ_2 are condition formulas.*
- *$\phi_1 \vee \phi_2$, where ϕ_1 and ϕ_2 are condition formulas.* $\square$

Observe that all the rule conditions in our example are condition formulas.

An essential feature of production systems is the ability to perform actions such as insertion and deletion of atoms. We now define the concrete actions for accomplishing that.

Definition 4.4.3 (Atomic Action)**.** *Let $p(\vec{c})$ be a literal. An* ***atomic action*** *is a statement that has one the following forms:*

- ***assert***(l)*: Adds the literal l to the working memory*
- ***retract***(l) : $\begin{cases} \textit{if } p \in \mathcal{P}_{PS} & \textit{Removes the } \text{atom}^{3} l \textit{ from the working memory} \\ \textit{if } p \in \mathcal{P}_{DL} & \textit{Enforces the literal } l \textit{ to be false in the working} \\ & \textit{memory} \qquad \square \end{cases}$

Beside these elementary actions, RIF also provides actions to change or delete objects and properties. Such actions can be treated similarly to `FOR`-rules below or as sequences of simpler actions, so we leave them out as well.

Definition 4.4.4 (Production System Augmented with Ontology)**.** *A* production system augmented with an ontology *(abbreviated as just* production system*, or* PS*) is a tuple*

$$\mathsf{PS} = (\mathcal{T}, \mathsf{L}, R)$$

such that

- *$\mathcal{T}$ is a DL ontology (T-box) whose predicates belong to $\mathcal{P}_{DL}$;*
- *L is a set of rule labels, and*
- *R is a set of rules, which are statements of one of the following forms*[4]

$$\texttt{IF-THEN Rule:} \quad r\!: \quad \texttt{Forall}\,\vec{x}\texttt{: if} \;\; \phi_r(\vec{x})\,\texttt{then}\,\psi_r(\vec{x}) \qquad (4.1)$$

$$\texttt{FOR Rule:} \quad r\!: \quad \texttt{For}\,\vec{x}\texttt{:}\,\phi_r(\vec{x}) \;\; \texttt{do}\,\psi_r(\vec{x}) \qquad (4.2)$$

where

- *$r \in \mathsf{L}$ is the above rule's label,*
- *ϕ_r is a condition formula in $\mathcal{L}$ with free variables $\vec{x}$,*
- *$\psi_r(\vec{x})$ is a sequence of atomic actions with free variables contained in $\vec{x}$.* $\square$

[3]Negative literals with predicate symbols in $\mathcal{P}_{PS}$ cannot occur in the working memories. See Definition 4.4.5.

[4] To avoid a misunderstanding, recall that the `Forall` construct is just a RIF-PRD syntax for declaring variables; it does not indicate a loop. In contrast, the `For-do` construct specifies a loop; it is found only in commercial PS systems, like JRules.

4.4.2 Operational Semantics

We now turn to the operational semantics of the combination of PS with ontologies. In a PS, two different constants represent two different domain elements, which is called the *unique name assumption*. In addition, production systems assume that each constant symbol is also a symbol in the domain of discourse, i.e., they are dealing with Herbrand domains.

It is also worth noting that the semantics presented in this section does not depend on the specifics of the DL associated with production systems. For concreteness, one can think of $\mathcal{L}DL^+$.

Definition 4.4.5 (Working Memory). *A* ***working memory****,* WM*, for a PS language* $\mathcal{L}$ *is a disjoint union*

$$\mathsf{WM} = \mathsf{WM}_{PS} \uplus \mathsf{WM}_{DL}$$

where WM_{PS} *is a set of ground atoms that use predicate symbols from* $\mathcal{P}_{PS}$ *and* WM_{DL} *is a set of ground literals that use predicate symbols from* $\mathcal{P}_{DL}$*.* $\Box$

Definition 4.4.6 ($\mathcal{T}$-structure). *Let* $\mathcal{T}$ *be a DL T-box. A* $\mathcal{T}$**-structure***,* $\mathcal{I}$*, for a PS language* $\mathcal{L}$ *has the form*

$$\mathcal{I} = (\mathsf{WM}_{PS} \uplus \mathsf{WM}_{DL} \uplus \mathbf{E}_{DL}, \sigma)$$

where $\mathsf{WM} = \mathsf{WM}_{PS} \uplus \mathsf{WM}_{DL}$ *is a working memory,* $\mathbf{E}_{DL}$ *is a set of* $\mathcal{P}_{DL}$*-literals,* σ *is a variable assignment, and* $(\mathsf{WM}_{DL} \uplus \mathbf{E}_{DL}, \sigma)$ *is a model of* $\mathcal{T}$*.* $\Box$

We say that (WM, σ), where WM is a working memory, is a ***prestructure***.

Example 4.4.7. *In Example 4.1.1, the two disjoint sets composing the initial working memory* WM_0 *are as follows:*

$\mathsf{WM}_{0PS} = \{\mathit{requested}(Smith, Laura, pcr)\}$ $\qquad$ $\mathsf{WM}_{0DL} = \{\mathit{flu}(Laura), \mathit{dnaT}(pcr)\}$

In addition, we can build up a $\mathcal{T}$*-structure,* $\mathcal{I}$*, by pairing any arbitrary assignment* σ *with* WM_0 *together with* $\{\mathbf{neg}\ \mathit{healthy}(Laura)\}$*. That is,* $\mathcal{I} = (\mathsf{WM}_0 \uplus \{\mathbf{neg}\ \mathit{healthy}(Laura)\}, \sigma)$*.*

$\Box$

Definition 4.4.8 (Satisfaction). *A* $\mathcal{T}$*-structure* $\mathcal{I} = (\mathsf{WM}_{PS} \uplus \mathsf{WM}_{DL} \uplus \mathbf{E}_{DL}, \sigma)$ satisfies *a literal l, denoted* $\mathcal{I} \models l$*, iff*

- *if l is a* $\mathcal{P}_{\mathsf{PS}}$*-atom then* $l^{\mathcal{I}} \in \mathsf{WM}_{PS}$
- *if l is a* $\mathcal{P}_{DL}$*-literal then* $\mathsf{WM}_{DL} \uplus \mathbf{E}_{DL} \models l^{\mathcal{I}}$

If ϕ *is a formula of the form* $\neg\phi_1$*,* $\phi_1 \wedge \phi_2$*,* $\phi_1 \vee \phi_2$ *then we define* $\mathcal{I} \models \phi$ *as usual in FOL. A formula* ϕ ***holds*** *in a prestructure* (WM, σ) *relative to an ontology* $\mathcal{T}$*, denoted* $\mathcal{T}, (\mathsf{WM}, \sigma) \models \phi$*, iff* $\mathcal{I} \models \phi$ *for every* $\mathcal{T}$*-structure of the form*

$$\mathcal{I} = (\mathsf{WM} \uplus \mathbf{E}_{DL}, \sigma)$$

(That is, WM *and* σ *are fixed but the* $\mathbf{E}_{DL}$ *varies.)* $\Box$

A prestructure is *$\mathcal{T}$-consistent* if there is a $\mathcal{T}$-structure with the same working memory and variable assignment, i.e., $(\mathsf{WM} \uplus \mathbf{E}_{DL}, \sigma)$ that does not entail f and $\mathbf{neg}\, f$ for any atom f. Note that in such a $\mathcal{T}$-structure, if $\mathbf{neg}\, f$ is true then $\neg f$ is also. A working memory is *$\mathcal{T}$-consistent* if it is part of a $\mathcal{T}$-consistent prestructure.

Definition 4.4.9 (Atomic Transition)**.** *Let* (WM, σ) *be a prestructure,* t_1, t_2 *be terms, and* α *be an action. We say that there is an* α-**transition** *from the prestructure* (WM, σ) *to the prestructure* (WM', σ)*, denoted* $(\mathsf{WM}, \sigma) \overset{\alpha}{\twoheadrightarrow} (\mathsf{WM}', \sigma)$*, iff*

- *if* $\alpha =$ ***assert***$(p(t_1, t_2))$ *then* $\mathsf{WM}' = (\mathsf{WM} \cup \{p(t_1^\sigma, t_2^\sigma)\}) - \{\mathbf{neg}\, p(t_1^\sigma, t_2^\sigma)\}$

- *if* $\alpha =$ ***retract***$(p(t_1, t_2)])$ *then* $\begin{cases} \text{if } p \in \mathcal{P}_{PS} & \mathsf{WM}' = \mathsf{WM} - \{p(t_1^\sigma, t_2^\sigma)\} \\ \text{if } p \in \mathcal{P}_{DL} & \mathsf{WM}' = (\mathsf{WM} \cup \\ & \{\mathbf{neg}\, p(t_1^\sigma, t_2^\sigma)\}) - \{p(t_1^\sigma, t_2^\sigma)\} \end{cases}$

where t^σ *is* $\sigma(t)$ *if* t *is a variable and it is* t *if* t *is a constant.* □

Definition 4.4.10 (Compound Transition)**.** *Let* (WM_0, σ) *be a prestructure and* $\alpha_1 \cdots \alpha_n$ *be atomic actions. We write*

$$(\mathsf{WM}_0, \sigma) \overset{\alpha_1 \ldots \alpha_n}{\twoheadrightarrow} (\mathsf{WM}_n, \sigma)$$

iff there are prestructures $(\mathsf{WM}_1, \sigma) \ldots (\mathsf{WM}_{n-1}, \sigma)$ *such that*

$$(\mathsf{WM}_0, \sigma) \overset{\alpha_1}{\twoheadrightarrow} (\mathsf{WM}_1, \sigma) \overset{\alpha_2}{\twoheadrightarrow} (\mathsf{WM}_2, \sigma) \overset{\alpha_3}{\twoheadrightarrow} \ldots \overset{\alpha_{n-1}}{\twoheadrightarrow} (\mathsf{WM}_{n-1}, \sigma) \overset{\alpha_n}{\twoheadrightarrow} (\mathsf{WM}_n, \sigma) \qquad \square$$

If, for some σ and $n \geqslant 1$, there is a transition $(\mathsf{WM}_0, \sigma) \overset{\alpha_1 \ldots \alpha_n}{\twoheadrightarrow} (\mathsf{WM}', \sigma)$ between prestructures then we will also write $\mathsf{WM}_0 \overset{\alpha_1 \ldots \alpha_n}{\twoheadrightarrow} \mathsf{WM}'$.

Since actions may introduce inconsistencies with respect to the ontology, we need to define the notion of a consistent result of applying an action. Let α be an action and suppose that

$$\mathsf{WM} \overset{\alpha}{\twoheadrightarrow} \mathsf{WM}'$$

is a transition among prestructures such that WM is $\mathcal{T}$-consistent and WM' is not. One way to define a consistent result of applying an action to WM is to take a maximal subset of WM' that belongs to some $\mathcal{T}$-consistent prestructure. However, a maximal subset might not be unique. A workaround here is to take the intersection of all the possible consistent results. This approach is called *When in Doubt Throw it Out* (WIDTIO) [100].

Definition 4.4.11 (Consistent Result)**.** *Let* WM *and* WM_n *be working memories, such that* WM *is* $\mathcal{T}$*-consistent, and* α *be an action. Suppose that there is a transition of the form*

$$\mathsf{WM} \overset{\alpha}{\twoheadrightarrow} \hat{\mathsf{WM}}$$

and $\hat{\mathsf{WM}}$ *is not* $\mathcal{T}$*-consistent. Let* M *be the set of all maximal subsets of* $\hat{\mathsf{WM}}$ *that contain* $\hat{\mathsf{WM}} - \mathsf{WM}$ *and are* $\mathcal{T}$*-consistent. We define the* $\mathcal{T}$***-consistent result*** *of applying* α *to* WM *as*

$$\hat{\mathsf{WM}}^{cons} = \bigcap_{\mathsf{WM}^{max} \in M} \mathsf{WM}^{max}$$

□

Example 4.4.12. *Suppose we execute* r_1 *in* WM_0*. We obtain the inconsistent working memory* $\mathsf{WM}_1 = \{\mathsf{takeT}(Laura, pcr), \mathsf{flu}(Laura), \mathsf{dnaT}(pcr), \mathsf{requested}(Smith, Laura, pcr)\}$*. We have two maximal consistent subsets of* WM_1

- $\mathsf{WM}_1' = \{\mathsf{takeT}(Laura, pcr), \mathsf{dnaT}(pcr), \mathsf{requested}(Smith, Laura, pcr)\}$
- $\mathsf{WM}_1' = \{\mathsf{takeT}(Laura, pcr), \mathsf{flu}(Laura), \mathsf{requested}(Smith, Laura, pcr)\}$

Thus, the consistent result is:

$$\mathsf{WM}_1^{cons} = \{\mathsf{takeT}(Laura, pcr), \mathsf{requested}(Smith, Laura, pcr)\}$$ □

Definition 4.4.13 (Consistent Transition)**.** *Let* (WM_0, σ) *be a* $\mathcal{T}$*-consistent prestructure and* $\alpha_1 \ldots \alpha_n$ *be a sequence of actions. Suppose that we have the following transitions:*

$$(\mathsf{WM}_0, \sigma) \overset{\alpha_1}{\twoheadrightarrow} (\mathsf{WM}_1, \sigma) \overset{\alpha_2}{\twoheadrightarrow}, \ldots (\mathsf{WM}_{n-1}, \sigma) \overset{\alpha_n}{\twoheadrightarrow} (\mathsf{WM}_n, \sigma)$$

Let WM_i^{cons} *(*$i = 1 \ldots n$*) be the* $\mathcal{T}$*-consistent results of applying* α_i *to* $(\mathsf{WM}_{i-1}^{cons}, \sigma)$ *(where* $\mathsf{WM}_0^{cons} = \mathsf{WM}_0$*). We define the corresponding* $\mathcal{T}$***-consistent transition*** *as*

$$(\mathsf{WM}_0, \sigma) \overset{\alpha_1}{\twoheadrightarrow} (\mathsf{WM}_1^{cons}, \sigma) \overset{\alpha_2}{\twoheadrightarrow}, \ldots (\mathsf{WM}_{n-1}^{cons}, \sigma) \overset{\alpha_n}{\twoheadrightarrow} (\mathsf{WM}_n^{cons}, \sigma)$$ □

Definition 4.4.14 (Intermediate and Cyclic WM)**.** *Let* (WM_0, σ) *be a* $\mathcal{T}$*-consistent prestructure and suppose* r *is a rule with actions* $\alpha_1 \cdots \alpha_n$ *in the head. Suppose there are working memories* $\mathsf{WM}_1 \ldots \mathsf{WM}_n$ *such that*

$$(\mathsf{WM}_0, \sigma) \overset{\alpha_1}{\twoheadrightarrow} (\mathsf{WM}_1, \sigma) \overset{\alpha_2}{\twoheadrightarrow} \ldots \overset{\alpha_{n-1}}{\twoheadrightarrow} (\mathsf{WM}_{n-1}, \sigma) \overset{\alpha_n}{\twoheadrightarrow} (\mathsf{WM}_n, \sigma)$$

In this case we say that WM_0 *and* WM_n *are* ***cycle*** *working memories, whereas* $\mathsf{WM}_1 \ldots \mathsf{WM}_{n-1}$ *are* ***intermediate*** *working memories.*[5] □

Intuitively, cycle working memories are the initial and the final (resulting) working memories, whereas intermediate working memories are the intermediate states produced by the execution of the rule actions.

We say that a transition of the form

$$(\mathsf{WM}_0, \sigma) \overset{\alpha_1}{\twoheadrightarrow} (\mathsf{WM}_1, \sigma) \overset{\alpha_2}{\twoheadrightarrow} \ldots \overset{\alpha_{n-1}}{\twoheadrightarrow} (\mathsf{WM}_{n-1}, \sigma) \overset{\alpha_n}{\twoheadrightarrow} (\mathsf{WM}_n, \sigma)$$

[5] RIF-PRD calls these cycle states because these are the states where cycles of rule applications begin.

is **non-trivial** if $\mathsf{WM}_0 \neq \mathsf{WM}_n$.

The following two definitions formalize the *conflict resolution strategy* for a given rule r. We say that a rule r is eligible for execution in a working memory WM if r's condition holds in WM, and the working memory resulting from applying r is consistent with the ontology. In addition, if r is an `IF-THEN` rule we require that r's action changes WM, and if r is a `FOR` rule we require that r's action is not instantiated twice with the same assignment.

Definition 4.4.15 (Fireable IF-THEN Rule)**.** *Let r be a rule of the form:*

$$r\colon \texttt{Forall}\ \vec{x}\colon \texttt{if}\quad \phi_r(\vec{x})\ \texttt{then}\ \psi_r(\vec{x}) \tag{4.3}$$

*We say that r is **fireable** in a prestructure (WM_0, σ) iff*

1. *(WM_0, σ) is $\mathcal{T}$-consistent.*
2. *$\mathcal{T}, (\mathsf{WM}_0, \sigma) \models \phi_r$*
3. *There is a non-trivial $\mathcal{T}$-consistent transition of the form*

$$(\mathsf{WM}_0, \sigma) \overset{\alpha_1 \ldots \alpha_n}{\twoheadrightarrow} (\mathsf{WM}_n, \sigma)$$

where $\overset{\alpha_1 \ldots \alpha_n}{\twoheadrightarrow}$ is defined in Definition 4.4.10.

*In this case we say that r **causes transition** from WM_0 to WM_n and denote it as $\mathsf{WM}_0 \overset{r}{\hookrightarrow} \mathsf{WM}_n$.* □

Definition 4.4.16 (Fireable FOR Rule)**.** *Let r be a rule of the form:*

$$r\colon \texttt{For}\ \vec{x}\colon \phi_r(\vec{x})\quad \texttt{do}\ \psi_r(\vec{x}) \tag{4.4}$$

*We say that r is **fireable** in a working memory WM_0 iff*

- *WM_0 is $\mathcal{T}$-consistent.*
- *there are prestructures $(\mathsf{WM}_0, \sigma_0), (\mathsf{WM}_1, \sigma_0), (\mathsf{WM}_1, \sigma_1) \ldots (\mathsf{WM}_n, \sigma_{n-1})$ such that there are $\mathcal{T}$-consistent transitions of the form*

$$\begin{array}{c} (\mathsf{WM}_0, \sigma_0) \overset{\alpha_1 \ldots \alpha_m}{\twoheadrightarrow} (\mathsf{WM}_1, \sigma_0) \\ (\mathsf{WM}_1, \sigma_1) \overset{\alpha_1 \ldots \alpha_m}{\twoheadrightarrow} (\mathsf{WM}_2, \sigma_1) \\ \vdots \\ (\mathsf{WM}_{n-1}, \sigma_{n-1}) \overset{\alpha_1 \ldots \alpha_m}{\twoheadrightarrow} (\mathsf{WM}_n, \sigma_{n-1}) \end{array} \tag{4.5}$$

where the following conditions hold:

1. Looping*: $\mathcal{T}, (\mathsf{WM}_i, \sigma_i) \models \phi_r$ for each cycle working memory,* WM_i *(*$0 \leqslant i \leqslant n-1$*)*
2. No repetitions*: For each pair of prestructures* $(\mathsf{WM}_i, \sigma_i), (\mathsf{WM}_j, \sigma_j)$ *(*$0 \leqslant i < j \leqslant n-1$*), we have that* $\sigma_i \neq \sigma_j$
3. Termination*: For every consistent transition of the form* $(\mathsf{WM}_n, \sigma) \overset{\alpha_1 \ldots \alpha_m}{\twoheadrightarrow} (\mathsf{WM}_{n+1}, \sigma)$ *such that* $\mathcal{T}, (\mathsf{WM}_n, \sigma) \models \phi_r$*, there is a prestructure* $(\mathsf{WM}_i, \sigma_i)$ *with (*$0 \leqslant i \leqslant n-1$*), such that* $\sigma = \sigma_i$*.*

In this case we say that r **causes transition** *from* WM_0 *to* WM_n *and denote it as* $\mathsf{WM}_0 \overset{r}{\hookrightarrow} \mathsf{WM}_n$. □

Condition 3 above says that rules are no longer fired once the system reaches a fixpoint. This guarantees that a PS does not have trivial infinite runs.

Recall that a PS applies rules in three steps: (1) pattern matching, (2) conflict resolution, (3) rule execution, and then it loops back to (1). So far we have described only the steps (1) and (3). The next series of definitions describes Step (2) and show how looping is modeled in the semantics. This semantics does not depend on any particular conflict resolution strategy so, for concreteness, in Step (2) we will simply randomly choose a fireable rule from the conflict resolution set.[6] Some other works [7, 28] use the same strategy.

Definition 4.4.17 (Consistent Transition Graph)**.** *The* **transition graph***,* $\mathcal{T}_{\mathsf{PS}}$*, of a production system is a directed labeled graph, whose set of nodes is the set of all working memories. There is an edge between two nodes* WM *and* WM'*, labeled with* α, σ *for some action* α *and variable assignment* σ*, iff* $(\mathsf{WM}, \sigma) \overset{\alpha}{\twoheadrightarrow} (\mathsf{WM}', \sigma)$*. We will use* $\mathcal{P}_{\mathsf{WM}}$ *to denote the set of all paths (sequences of WMs) in the graph* $\mathcal{T}_{\mathsf{PS}}$ *starting at* WM. □

Definition 4.4.18 (Split)**.** *Let* $\pi = \mathsf{WM}_0 \ldots \mathsf{WM}_n$ *be a path in* $\mathcal{P}_{\mathsf{WM}_0}$*. A* split *of* π *is a pair of subpaths,* π_1 *and* π_2*, such that* $\pi_1 = \mathsf{WM}_0 \ldots \mathsf{WM}_i$ *and* $\pi_2 = \mathsf{WM}_i \ldots \mathsf{WM}_n$ *for some* i *(*$1 \leqslant i \leqslant n$*). In this case, we write* $\pi = \pi_1 \circ \pi_2$*.* □

Definition 4.4.19 (Run)**.** *A path* π *in* $\mathcal{P}_{\mathsf{WM}_0}$ *is a* **run** $\mathcal{R}$ *for a production system* PS *iff there are splits* $\pi = \pi_1 \circ \cdots \circ \pi_n$ *and rules* $r_1 \ldots r_n$ *such that for each* $i = 1 \ldots n$*,* $WM_{i,start} \overset{r_i}{\hookrightarrow} WM_{i,end}$*, where* $WM_{i,start}$ *is the first element in* π_i *and* $WM_{i,end}$ *is its last. Note that this implies that every* π_i *is a* $\mathcal{T}$*-consistent transition (see Definition 4.4.13).* □

We will refer to the ith **cycle** working memory in a run as WM_i.

4.5 $\mathcal{TR}^{\text{PAD}}$ and its Semantics

The alphabet of the language $\mathcal{L}_{\mathcal{TR}}$ of $\mathcal{TR}^{\text{PAD}}$ is defined the same way as in the case DL except that now the set of all predicates $\mathcal{P}$ is further partitioned into two subsets, $\mathcal{P}_{fluents}$

[6]Recall that the conflict resolution set contains all the rules that can be fired on a given working memory.

and $\mathcal{P}_{actions}$. The former will be used to represent facts in database states and the latter for transactions that change those states. Querying a fluent can be viewed as an action that does not change the underlying database state. We also add new symbols, $\rhd$ and $\overset{a}{\rightsquigarrow}$, where a is an atom whose predicate symbol is in $\mathcal{P}_{actions}$.

Terms are defined as usual in first order logic. States are referred to with the help of special constants called ***state identifiers***; these will be usually denoted by boldface lowercase letters $\mathbf{d}$, $\mathbf{d}_1$, $\mathbf{d}_2$. The symbol $\mathbf{neg}$ will be used to represent the explicit negation and $\mathbf{not}$ will be used for the *default* negation. These two symbols are applicable to fluents only. A fluent literal is either an atomic fluent or it has one of the following negated forms: $\mathbf{neg}\, f$, $\mathbf{not}\, f$, $\mathbf{not}\ \mathbf{neg}\, f$, where f is an atomic fluent. Literals that do not mention $\mathbf{not}$ are said to be $\mathbf{not}$-free.

Note that in the ontologies one can have both $\mathbf{neg}$- and $\neg$-literals, while $\mathcal{TR}^{PAD}$ uses $\mathbf{neg}$- and $\mathbf{not}$-literals instead. This is because logic programming rules cannot use classical negation, while ontologies do not use default negation.

Like the original Transaction Logic [9, 10], $\mathcal{TR}^{PAD}$ contains logical connectives from the standard FOL ($\wedge, \vee, \forall, \exists$,) plus two additional logical connectives: the ***serial conjunction***, $\otimes$, and the modal operator $\Diamond$ for *hypothetical* execution. Informally, a serial conjunction of the form $\phi \otimes \psi$ represents an action composed of an execution of ϕ followed by an execution of ψ. A ***hypothetical formula***, $\Diamond\phi$, represents an action where ϕ is *tested* whether it can be executed at the current state, but no actual state changes take place. For instance, the first part of the following formula

$$\Diamond(\mathit{insert}(\mathsf{infection}) \otimes \mathit{bill\_insurance} \otimes \mathsf{has\_paid}) \otimes \mathit{insert}(\mathsf{takesT})$$

is a hypothetical test to verify that the patient's insurance company will pay in case of an infection after the blood test. The actual blood test is only performed if the hypothetical test succeeds. We will assume that hypothetical formulas contain only serial conjunctions of literals.

Definition 4.5.1 ($\mathcal{TR}$ with Partially Defined Actions)**.** *$\mathcal{TR}^{PAD}$ consists of* **serial-Horn** *rules,* **partial action definitions** *(PADs), and certain statements about states and actions, which we call* premises. *The syntax for all these is shown below, where c stands for a $\mathbf{not}$-free literal, $c_1, \ldots, c_n$ are* literals *(fluents or actions), f is a $\mathbf{not}$-free fluent literal, b_1, b_2 are conjunctions of* fluent *literals or hypotheticals ($\mathbf{not}$-literals are ok), b_3, b_4 are conjunctions of $\mathbf{not}$-free* fluent *literals, $\boldsymbol{d}_0, \boldsymbol{d}_1 \ldots$ are identifiers, and a is an action atom.*

Rules			***Premises***		
(i)	$c \leftarrow c_1 \otimes \cdots \otimes c_n$	*(a serial-Horn rule)*	*(iii)*	$\boldsymbol{d}_0 \rhd f$	*(a* ***state****-premise)*
(ii)	$b_1 \otimes a \otimes b_2 \rightarrow b_3 \otimes a \otimes b_4$	*(a PAD)*	*(iv)*	$\boldsymbol{d}_1 \overset{a}{\rightsquigarrow} \boldsymbol{d}_2$	*(a* ***run****-premise)*

The serial-Horn rule (i) is a statement that defines the literal c, which can be viewed as a calling sequence for a complex transaction and $c_1 \otimes \cdots \otimes c_n$ can be viewed as a definition for the actual course of action to be performed by that transaction. If c is a fluent literal then we require that c_1, ..., c_n are also fluents. In that case we call c a **defined fluent** *and the rule itself a* **fluent rule**. *Fluent rules are akin to regular Horn rules in logic programming. If* c *is an action, we will say that* c *is a* **compound action**, *as it is defined by a rule. In contrast, actions such as* a *in* (ii) *are called* **partially defined actions**. *They are defined by* **partial action definitions** *(PADs) of the form (ii); they cannot appear in the heads of serial-Horn rules.* $\square$

For instance, the serial-Horn rule

$$\textit{r\_1} \leftarrow \mathsf{requested}(D,P,T) \otimes \mathsf{dnaT}(T) \otimes \textit{insert}(\mathsf{takesT}(P,T))$$

defines a compound action *r_1*. This action behaves in the same way as rule r_1 in Example 4.1.1. The PAD (ii) means that if we know that b_1 holds before executing action a and b_2 holds after, we can conclude that b_3 must have held before executing a and b_4 must hold as a result of a. For instance, the PAD

$$\mathsf{healthy}(P) \otimes \textit{insert}(\mathsf{dnaT}(T)) \rightarrow \textit{insert}(\mathsf{dnaT}(T)) \otimes \mathsf{healthy}(P)$$

states that if a patient is healthy, she remains so after adding a DNA type in the database. This is a simplified version of an inertial law in $\mathcal{TR}^{PAD}$. Note that the serial conjunction $\otimes$ binds stronger than the implication, so the above statement should be interpreted as: $(\mathsf{healthy}(P) \otimes \textit{insert}(\mathsf{dnaT}(T))) \rightarrow (\textit{insert}(\mathsf{dnaT}(T)) \otimes \mathsf{healthy}(P))$. To sum up, we distinguish two kinds of actions: *partially defined actions* (abbr., *pda*) and *compound actions*. Partially defined actions cannot be defined by rules—they are defined by PAD statements only. In contrast, compound actions are defined via serial-Horn rules but not by PADs. Note that *pda*s can appear in the bodies of serial-Horn rules that define compound actions (see *r_1* above) and, in this way, $\mathcal{TR}^{PAD}$ can create larger action theories by composing smaller ones in a modular way.

Premises are statements about the initial and the final database states (state premises) and about possible state transitions caused by partially defined actions (run-premises).

For example, to represent the initial database in our example, we can use the state premises

$$\mathbf{d}_0 \rhd \mathsf{dnaT}(\mathrm{pcr})$$
$$\mathbf{d}_0 \rhd \mathsf{requested}(\mathrm{Smith}, \mathrm{Laura}, \mathrm{pcr})$$
$$\mathbf{d}_0 \rhd \mathsf{flu}(\mathrm{Laura})$$

The run-premise

$$\mathbf{d}_0 \overset{\textit{insert}(\mathsf{takeT}(\vec{t}))}{\rightsquigarrow} \mathbf{d}_1$$

says that executing the *pda* action $\textit{insert}(\mathsf{takeT}(\vec{t}))$ in the state associated with $\mathbf{d}_0$ leads to the state represented by $\mathbf{d}_1$.

A ***transaction*** is a statement of the form $?\!-_{(\mathbf{d}_0)}\, \exists \bar{X}\phi$, where $\phi = l_1 \otimes \cdots \otimes l_k$ is a serial conjunction of literals (both fluent and action literals) and $\bar{X}$ is a list of all the variables that occur in ϕ. Transactions in $\mathcal{TR}$ generalize the notion of queries in ordinary logic programming. For instance,

$$?\!-_{(\mathbf{d}_0)}\mathsf{flu}(\mathrm{Laura}) \otimes \textit{r\_1}$$

is a transaction that first checks if the patient has a flu in the initial state $\mathbf{d}_0$; if so, the compound action *r_1* is executed. Note that if the execution of the transaction cannot proceed the already executed actions are undone and the underlying database state remains unchanged.

A $\mathcal{TR}^{PAD}$ ***transaction base*** is a set of serial-Horn rules. A $\mathcal{TR}^{PAD}$ ***action base*** is a set of PADs. A $\mathcal{TR}^{PAD}$ **specification** is a tuple $(\mathcal{E}, \mathbf{P}, \mathcal{S})$ where $\mathcal{E}$ is a $\mathcal{TR}^{PAD}$ action base, $\mathbf{P}$ is a $\mathcal{TR}^{PAD}$ transaction base, and $\mathcal{S}$ is a set of premises.

4.5.0.0.1 Semantics. This semantics uses three truth values, **u**, **t** and **f**, which stand for *true*, *false*, and *undefined* and are ordered as follows: $\mathbf{f} < \mathbf{u} < \mathbf{t}$. In addition, we will use the following operator $\sim$: $\sim \mathbf{t} = \mathbf{f}$, $\sim \mathbf{f} = \mathbf{t}$, $\sim \mathbf{u} = \mathbf{u}$.

A **database state** $\mathbf{D}$ (or just a *state,* for short) is a set of ***ground*** (i.e., variable-free) fluent literals.

The semantics is based on the notion of *path structures*.

Definition 4.5.2 (Three-valued Partial Herbrand Interpretation). *A* ***partial Herbrand interpretation*** *is a mapping $\mathcal{H} : \mathcal{B} \mapsto \{\boldsymbol{f}, \boldsymbol{u}, \boldsymbol{t}\}$ that assigns a truth value, $\boldsymbol{f}$, $\boldsymbol{u}$, or $\boldsymbol{t}$, to every formula ϕ in $\mathcal{B}$.* □

A central feature in the semantics of $\mathcal{TR}$ is the notion of *execution paths*, since $\mathcal{TR}$ formulas are evaluated over paths and *not* over states like in temporal logics

Definition 4.5.3 (Three-valued Herbrand Path Structure). *A* ***Herbrand path structure*** *is a mapping $\mathbf{I}$ that assigns a partial Herbrand interpretation to every path. That is, for any path π, $\mathbf{I}(\pi)$ is an interpretation. So, for instance, $\mathbf{I}(\pi)(f)$ is a truth value for any literal f. This mapping must satisfy the restriction that for each ground base fluent f and database state $\mathbf{D}$:*

$$\mathbf{I}(\langle \mathbf{D} \rangle)(f) = \boldsymbol{t} \textit{ if } f \in \mathbf{D} \quad \textit{and} \quad \mathbf{I}(\langle \mathbf{D} \rangle)(f) = \boldsymbol{f} \textit{ if } \mathbf{neg}\, f \in \mathbf{D}$$

where $\langle \mathbf{D} \rangle$ is a path that contains only one state, $\mathbf{D}$.

In addition, $\mathbf{I}$ includes a mapping of the form:

$$\Delta_{\mathbf{I}} : \text{ State identifiers} \longrightarrow \text{Database states}$$

which associates states (i.e., sets of atomic formulas) to state identifiers. We will usually omit the subscript. □

Intuitively, Herbrand path structures in $\mathcal{TR}$ play a role similar to transition functions in temporal logics by providing a link between states and actions.

An ***execution path*** of length k, or a ***k-path***, is a finite sequence of states, $\pi = \langle \mathbf{D}_1 \ \ldots \ \mathbf{D}_k \rangle$, where $k \geqslant 1$. A ***path abstraction*** is a finite sequence of state identifiers. If $\langle \mathbf{d}_1 \ \ldots \ \mathbf{d}_k \rangle$ is a path abstraction then $\langle \mathbf{D}_1 \ \ldots \ \mathbf{D}_k \rangle$, where $\mathbf{D}_i = \Delta(\mathbf{d}_i)$, is an execution path. We will also sometimes write $\mathcal{M}(\langle \mathbf{d}_1 \ \ldots \ \mathbf{d}_k \rangle)$ meaning $\mathcal{M}(\langle \Delta(\mathbf{d}_1) \ \ldots \ \Delta(\mathbf{d}_k) \rangle)$.

Definition 4.5.4 (Split). *Let π be a path. A* split *of π is a pair of subpaths, π_1 and π_2, such that $\pi_1 = \langle \mathbf{D}_1 \ldots \mathbf{D}_i \rangle$ and $\pi_2 = \langle \mathbf{D}_i \ldots \mathbf{D}_k \rangle$ for some i ($1 \leqslant i \leqslant k$). In this case, we write $\pi = \pi_1 \circ \pi_2$.* □

In the remainder of this section we will consider ground rules and PADs only. We can make this assumption without loosing generality because all the variables in a rule are considered to be universally quantified. In addition, we assume that the language includes the distinguished propositional constants $\mathbf{t}^{\pi}$, and $\mathbf{u}^{\pi}$ for each $\mathcal{TR}$ path π. Observe that since there is an infinite number of paths, there is an infinite number of such constants. Informally, $\mathbf{t}^{\pi}$ ($\mathbf{u}^{\pi}$) is a proposition that has the truth value **t**(respectively **u**) only on the path π, and it is false on every other path. That is, $\mathbf{I}(\pi')(\mathbf{t}^{\pi}) = \mathbf{t}$ (respectively $\mathbf{I}(\pi')(\mathbf{u}^{\pi}) = \mathbf{u}$) if and only if $\pi = \pi'$.

The following definition formalizes the idea that truth of $\mathcal{TR}$ formulas is defined on paths.

Definition 4.5.5 (Satisfaction). *Let $\mathbf{I}$ be a Herbrand path structure, π be a path, f a ground* **not** *-free literal, and G, G_1, G_2 ground serial goals. We define* ***truth valuations*** *with respect to the path structure $\mathbf{I}$ as follows:*

- $\mathbf{I}(\pi)(f)$ *was already defined as part of the definition of Herbrand path structures.*
- $\mathbf{I}(\pi)(\phi \otimes \psi) = max\{min(\mathbf{I}(\pi_1)(\phi), \mathbf{I}(\pi_2)(\psi) \mid \pi = \pi_1 \circ \pi_2\}$
- $\mathbf{I}(\pi)(G_1 \wedge G_2) = min(\mathbf{I}(\pi)(G_1), \mathbf{I}(\pi)(G_2))$
- $\mathbf{I}(\pi)(\mathbf{not}\, \phi) = \sim \mathbf{I}(\pi)(\phi)$[7]
- $\mathbf{I}(\pi)(\diamond\phi) = \begin{cases} max\{\mathbf{I}(\pi')(\phi) \mid \pi' \textit{ is a path that starts at } \mathbf{D}, \} & \textit{if } \pi = \langle \mathbf{D} \rangle \\ \boldsymbol{f} & \textit{otherwise} \end{cases}$
- $\mathbf{I}(\pi)(f \leftarrow G) = \boldsymbol{t}$ *iff* $\mathbf{I}(\pi)(f) \geqslant \mathbf{I}(\pi)(G)$
- $\mathbf{I}(\pi)(b_1 \otimes \alpha \otimes b_2 \rightarrow b_3 \otimes \alpha \otimes b_4) = \boldsymbol{t}$ *iff* π *has the form* $\langle \mathbf{D}_1, \mathbf{D}_2 \rangle$, $\mathbf{I}(\langle \mathbf{D}_1, \mathbf{D}_2 \rangle)(\alpha) = \boldsymbol{t}$*, and the following holds:*
$$min\{min\{\mathbf{I}(\langle \mathbf{D}_1 \rangle)(f) \mid f \in b_1\}, min\{\mathbf{I}(\langle \mathbf{D}_2 \rangle)(f) \mid f \in b_2\}\}$$
$$\leqslant$$
$$min\{min\{\mathbf{I}(\langle \mathbf{D}_1 \rangle)(f) \mid f \in b_3\}, min\{\mathbf{I}(\langle \mathbf{D}_2 \rangle)(f) \mid f \in b_4\}\}$$

We write $\mathbf{I}, \pi \models \phi$ and say that ϕ is ***satisfied*** *on path π in the path structure $\mathbf{I}$ if $\mathbf{I}(\pi)(\phi) =$* ***t****.* □

Definition 4.5.6 (Model). *A path structure, $\mathbf{I}$, is a* ***model of a formula*** *ϕ if $\mathbf{I}, \pi \models \phi$ for every path π. In this case we write $\mathbf{I} \models \phi$. A path structure, $\mathbf{I}$, is a model of a set of formulas if it is a model of every formula in the set. A path structure, $\mathbf{I}$, is a* ***model of a premise-statement*** *σ iff:*

[7]Recall that $\sim \mathbf{t} = \mathbf{f}$, $\sim \mathbf{f} = \mathbf{t}$, $\sim \mathbf{u} = \mathbf{u}$.

- *σ is a run-premise of the form $\boldsymbol{d}_1 \overset{\alpha}{\rightsquigarrow} \boldsymbol{d}_2$ and $\mathbf{I}, \langle \boldsymbol{d}_1\boldsymbol{d}_2\rangle \models \alpha$; or*
- *σ is a state-premise $\boldsymbol{d} \rhd f$ and $\mathbf{I}, \langle \boldsymbol{d}\rangle \models f$.*

$\mathbf{I}$ *is a* ***model*** *of a specification $(\mathcal{E}, \mathbf{P}, \mathcal{S})$ if $\mathbf{I}$ is an interpretation for $\mathcal{E}$ and it satisfies every rule in $\mathbf{P}$ and every premise in $\mathcal{S}$.* □

Example 4.5.7. *Consider again Example 4.1.1. Let us present the specification $\Lambda = (\mathcal{E}, \mathbf{P}, \mathcal{S})$, that intuitively encodes the ontology and part of the PS. A complete encoding will be described in Section 4.6. Assume that $\mathcal{S}$ contains the premises already introduced in the previous section. The transaction base $\mathbf{P}$ contains the following rules encoding the ontology*

$$\begin{array}{ll} \mathbf{neg}\ \mathsf{virusT}(T) \leftarrow \mathsf{dnaT}(T) & \\ \mathbf{neg}\ \mathsf{healthy}(P) \leftarrow \mathsf{flu}(P) & \mathsf{healthy}(P) \leftarrow \mathsf{takesT}(P,T), \mathbf{neg}\ \mathsf{virusT}(T) \end{array}$$

The action base $\mathcal{E}$ contains two PADs encoding the (simplified) inertia laws, and the definition of the action $\mathsf{insert}_{\mathsf{takeT}}$.

$$\begin{array}{rl} \mathsf{dnaT}(P) \otimes \mathsf{insert}_{\mathsf{takeT}}(P,T) \otimes \mathbf{not}\ \mathsf{inconsistent} & \rightarrow \mathsf{insert}_{\mathsf{takeT}}(P,T) \otimes \mathsf{dnaT}(P) \\ \mathsf{flu}(P) \otimes \mathsf{insert}_{\mathsf{takeT}}(P,T) \otimes \mathbf{not}\ \mathsf{inconsistent} & \rightarrow \mathsf{insert}_{\mathsf{takeT}}(P,T) \otimes \mathsf{flu}(P) \\ \mathsf{insert}_{\mathsf{takeT}}(P,T) & \rightarrow \mathsf{insert}_{\mathsf{takeT}}(P,T) \otimes \mathsf{takeT}(P,T) \end{array}$$

From the premises and the rules in Λ, we can see that any path structure $\mathcal{I}$ that models Λ satisfies

$$\begin{array}{ll} \mathcal{I}(\boldsymbol{d}_0)(\mathsf{dnaT}(pcr)) = \boldsymbol{t} & \\ \mathcal{I}(\boldsymbol{d}_0)(\mathsf{flu}(Laura)) = \boldsymbol{t} & \mathcal{I}(\boldsymbol{d}_0\boldsymbol{d}_1)(\mathsf{insert}_{\mathsf{takeT}}(Laura, pcr) = \boldsymbol{t} \end{array}$$

Now take an interpretation, $\mathcal{I}_1$, such that $\mathcal{I}_1(\boldsymbol{d}_1)(\mathsf{inconsistent}) = \boldsymbol{f}$. From the PADs in $\mathcal{E}$ instantiated with pcr, Laura, and Smith, we can conclude that:

$$\begin{array}{ll} \mathcal{I}_1(\boldsymbol{d}_1)(\mathsf{dnaT}(pcr)) = \boldsymbol{t} & \\ \mathcal{I}_1(\boldsymbol{d}_1)(\mathsf{flu}(Laura)) = \boldsymbol{t} & \mathcal{I}_1(\boldsymbol{d}_1)(\mathsf{takeT}(Laura, pcr) = \boldsymbol{t} \end{array}$$

and from the rules in the ontology it follows that $\mathcal{I}_1(\boldsymbol{d}_1)(\mathsf{healthy}) = \boldsymbol{t}$ and $\mathcal{I}_1(\boldsymbol{d}_1)(\mathbf{neg}\ \mathsf{healthy}) = \boldsymbol{t}$. Thus, $\boldsymbol{d}_1$ is inconsistent in $\mathcal{I}_1$. □

In classical logic programming based on three-valued models, given two Herbrand *partial* interpretations $\mathbf{N}_1$ and $\mathbf{N}_2$, we say that

- $\mathbf{N}_1 \leqslant^c \mathbf{N}_2$ iff all $\mathbf{not}$-free literals that are true in $\mathbf{N}_1$ are true in $\mathbf{N}_2$ and all $\mathbf{not}$-literals that are true in $\mathbf{N}_1$ are true in $\mathbf{N}_2$. This coincides with set-theoretic *inclusion* and is called the *information ordering*.
- $\mathbf{N}_1 \preceq^c \mathbf{N}_2$ iff all $\mathbf{not}$-free literals that are true in $\mathbf{N}_1$ are true in $\mathbf{N}_2$ and all $\mathbf{not}$-literals that are true in $\mathbf{N}_2$ are true in $\mathbf{N}_1$. This is called the *truth ordering*.

Definition 4.5.8 (Order on Path Structures). *Let $\mathbf{M}_1$ and $\mathbf{M}_2$ be two Herbrand path structures, then:*

- Information ordering: $\mathbf{M}_1 \leqslant \mathbf{M}_2$ *if for every path, π, it holds that* $\mathbf{M}_1(\pi) \leqslant^c \mathbf{M}_2(\pi)$.

- Truth ordering: $\mathbf{M}_1 \preceq \mathbf{M}_2$ *if for every path, π, it holds that* $\mathbf{M}_1(\pi) \preceq^c \mathbf{M}_2(\pi)$. □

These two orderings are considerably different. The *truth ordering* minimize the *amount of truth*, by minimizing the atoms that are true and maximizing the atoms that are false on each path. In contrast, the *information ordering* minimizes the amount of information by minimizing both the atoms that are true and false in each path. For instance, the smallest model with respect to $\leqslant$ for the program $\{\alpha \to \alpha\}$ is one where α is *undefined* on every path; in contrast, the least model with respect to $\preceq$ is one when α is false on every path.

Example 4.5.9. *Consider a path structure $\mathcal{I}_2$ for the specification Λ in Example 4.5.7 that coincides with $\mathcal{I}_1$ in the path $\langle \boldsymbol{d}_0 \rangle$ but differs in $\langle \boldsymbol{d}_1 \rangle$ as follows:*

$$\mathcal{I}_2(\boldsymbol{d}_1)(\mathsf{dnaT}(pcr)) = \boldsymbol{u} \qquad \mathcal{I}_2(\boldsymbol{d}_1)(\mathsf{inconsistent}) = \boldsymbol{u}$$
$$\mathcal{I}_2(\boldsymbol{d}_1)(\mathsf{flu}(Laura)) = \boldsymbol{u} \qquad \mathcal{I}_2(\boldsymbol{d}_1)(\mathsf{takeT}(Laura, pcr) = \boldsymbol{t}$$

It is not hard to see that $\mathcal{I}_2$ is also a model of Λ, and moreover $\mathcal{I}_2 \preceq \mathcal{I}_1$. □

Definition 4.5.10 (Least Model). *A model* $\mathbf{M}$ *of a specification $(\mathcal{E}, \mathbf{P}, \mathcal{S})$ is **minimal** with respect to $\preceq$ iff for any other model,* $\mathbf{N}$*, of $(\mathcal{E}, \mathbf{P}, \mathcal{S})$, if* $\mathbf{N} \preceq \mathbf{M}$ *then* $\mathbf{N} = \mathbf{M}$*. The **least** model of $(\mathcal{E}, \mathbf{P}, \mathcal{S})$, denoted LPM$(\mathcal{E}, \mathbf{P}, \mathcal{S})$, is a minimal model that is unique.* □

The following definition is key to the notion of well-founded $\mathcal{TR}^{PAD}$ models. It is modeled after [43] with appropriate extensions for PADs.

Definition 4.5.11 ($\mathcal{TR}^{PAD}$-quotient). *Let $(\mathcal{E}, \mathbf{P}, \mathcal{S})$ be a $\mathcal{TR}^{PAD}$ specification, and* $\mathbf{I}$ *a Herbrand path structure. By $\mathcal{TR}^{PAD}$**-quotient** of $(\mathcal{E}, \mathbf{P}, \mathcal{S})$ **modulo*** $\mathbf{I}$ *we mean a new specification, $\frac{(\mathcal{E},\mathbf{P},\mathcal{S})}{\mathbf{I}}$, which is obtained from $(\mathcal{E}, \mathbf{P}, \mathcal{S})$ by*

- *Then replacing every literal of the form* $\mathbf{not}\, b$ *in* $\mathbf{P} \cup \mathcal{E}$ *with*

  $\boldsymbol{t}^{\pi}$ *for every path π such that* $\mathbf{I}(\pi)(\mathbf{not}\, b) = \boldsymbol{t}$
  $\boldsymbol{u}^{\pi}$ *for every path π such that* $\mathbf{I}(\pi)(\mathbf{not}\, b) = \boldsymbol{u}$

- *And finally removing all the remaining rules and PADs that have a literal of the form* $\mathbf{not}\, b$ *in the body such that* $\mathbf{I}(\pi)(\mathbf{not}\, b) = \boldsymbol{f}$ *for some path π.* □

Example 4.5.12. *Consider the specification $(\mathcal{E}, \mathbf{P}, \mathcal{S})$ consisting in the following PADs and rules:*

(1)	*PAD* :	$\alpha \to \alpha \otimes \mathbf{neg}\, f_1$
(2)	*PAD* :	$c_1 \otimes \alpha \otimes \mathbf{not}\, \mathsf{inconsistent} \to \alpha \otimes c_1$
(3)	*PAD* :	$c_2 \otimes \alpha \otimes \mathbf{not}\, \mathsf{inconsistent} \to \alpha \otimes c_2$
(4)	*PAD* :	$\beta \to \beta \otimes f_1$
(5)	*Fluent rule:*	$f_1 \leftarrow c_1 \wedge c_2$
(6)	*Fluent rule:*	$\mathsf{inconsistent} \leftarrow f_1 \wedge \mathbf{neg}\, f_1$
(7)	*Action rule:*	$\gamma \leftarrow \alpha \otimes \mathbf{not}\, \mathsf{inconsistent} \otimes \beta$

where $\mathsf{inconsistent}$*, f_1, c_1 and c_2 are fluents, and α, β, and γ are actions. In plain English, PAD (1) describes the effect of α on $\mathbf{neg}\, f_1$. PADs (2) and (3) encode the frame axioms for the fluents c_1 and c_2 with respect to α with further qualification that they must not cause inconsistency. PAD (4) describes the effect of β on f_1. The fluent rule (5) defines the fluent f_1 in terms of the fluents c_1 and c_2, and (6) defines the fluent* $\mathsf{inconsistent}$*. Rule (7) defines the action γ. Let $\mathbf{I}$ be the Herbrand path where everything is undefined in every path. The quotient $\frac{(\mathcal{E},\mathbf{P},\mathcal{S})}{\mathbf{I}}$ is then as follows:*

$(1)\quad \alpha \rightarrow \alpha \otimes \mathbf{neg}\, f_1$
$(2)\quad c_1 \otimes \alpha \otimes \boldsymbol{u}^{\pi} \rightarrow \alpha \otimes c_1$ *(multiple copies for all possible paths π)*
$(3)\quad c_2 \otimes \alpha \otimes \boldsymbol{u}^{\pi} \rightarrow \alpha \otimes c_2$ *(again, one per path π)*
$(4)\quad \beta \rightarrow \beta \otimes f_1$
$(5)\quad f_1 \leftarrow c_1 \wedge c_2$
$(6)\quad \mathsf{inconsistent} \leftarrow f_1 \wedge \mathbf{neg}\, f_1$
$(7)\quad \gamma \leftarrow \alpha \otimes \boldsymbol{u}^{\pi} \otimes \beta$

□

Next, we give a constructive definition of *well-founded models* for $\mathcal{TR}^{PAD}$ specifications in terms of a consequence operator.

Definition 4.5.13 ($\mathcal{TR}^{PAD}$ Immediate Consequence Operator)**.** *The consequence operator, Γ, for a $\mathcal{TR}^{PAD}$ specification is defined by analogy with the classical case:*

$$\Gamma(\mathbf{I}) = \mathit{LPM}(\frac{(\mathcal{E},\mathbf{P},\mathcal{S})}{\mathbf{I}})$$

Suppose $\mathbf{I}_{\emptyset}$ is the path structure that maps each path π to the empty Herbrand interpretation in which all atoms are undefined. That is, for every path π and literal f, we have $\mathbf{I}_{\emptyset}(\pi)(f) = \boldsymbol{u}$. The ordinal powers of the consequence operator Γ are then defined inductively as follows:

- $\Gamma^{\uparrow 0}(\mathbf{I}_{\emptyset}) = \mathbf{I}_{\emptyset}$
- $\Gamma^{\uparrow n}(\mathbf{I}_{\emptyset}) = \Gamma(\Gamma^{\uparrow n-1}(\mathbf{I}_{\emptyset}))$, *if n is a successor ordinal*
- $\Gamma^{\uparrow n}(\mathbf{I}_{\emptyset})(\pi) = \bigcup_{j \leqslant n} \Gamma^{\uparrow j}(\mathbf{I}_{\emptyset})(\pi)$, *if n is a limit ordinal* □

The operator Γ is monotonic with respect to the $\leqslant$-order when $(\mathcal{E},\mathbf{P},\mathcal{S})$ is fixed. Because of this, the sequence $\{\Gamma^{\uparrow n}(\mathbf{I}_{\emptyset})\}$ has a least fixed point and is computable via transfinite induction.

Definition 4.5.14 (Well-founded Model)**.** *The **well-founded** model of a $\mathcal{TR}^{PAD}$ specification $(\mathcal{E},\mathbf{P},\mathcal{S})$, written $\mathit{WFM}((\mathcal{E},\mathbf{P},\mathcal{S}))$, is defined as a limit of the sequence $\{\Gamma^{\uparrow n}(\mathbf{I}_{\emptyset})\}$.*
□

Example 4.5.15. *Consider the specification,* $(\mathcal{E}, \mathbf{P}, \mathcal{S})$*, in Example 4.5.12. Suppose that we have the following set of premises in* $\mathcal{S}$*:*

$$\begin{aligned}
&\boldsymbol{d}_1 \rhd c_1 \\
&\boldsymbol{d}_1 \rhd c_2 \\
&\boldsymbol{d}_1 \rhd f_1 \\
&\boldsymbol{d}_1 \overset{\alpha}{\rightsquigarrow} \boldsymbol{d}_2 \\
&\boldsymbol{d}_2 \overset{\beta}{\rightsquigarrow} \boldsymbol{d}_3
\end{aligned}$$

Then, the least model $\mathcal{M}$ *of* $\frac{(\mathcal{E},\mathbf{P},\mathcal{S})}{\mathbf{I}}$ *has the following properties:*

1. $\mathcal{M}(\langle \boldsymbol{d}_1\boldsymbol{d}_2\rangle)(\alpha) = \boldsymbol{t}$
2. $\mathcal{M}(\langle \boldsymbol{d}_2\boldsymbol{d}_3\rangle)(\beta) = \boldsymbol{t}$
3. $\mathcal{M}(\langle \boldsymbol{d}_1\rangle)(c_1) = \boldsymbol{t}$
4. $\mathcal{M}(\langle \boldsymbol{d}_1\rangle)(c_2) = \boldsymbol{t}$
5. $\mathcal{M}(\langle \boldsymbol{d}_1\rangle)(f_1) = \boldsymbol{t}$
6. $\mathcal{M}(\langle \boldsymbol{d}_2\rangle)(\mathbf{neg}\, f_1) = \boldsymbol{t}$
7. $\mathcal{M}(\langle \boldsymbol{d}_1\rangle)(\mathbf{neg}\, f_1) = \boldsymbol{f}$
8. $\mathcal{M}(\langle \boldsymbol{d}_0\boldsymbol{d}_1\rangle)(\alpha) = \boldsymbol{f}$
9. $\mathcal{M}(\langle \boldsymbol{d}_2\rangle)(c_1) = \boldsymbol{u}$
10. $\mathcal{M}(\langle \boldsymbol{d}_2\rangle)(c_2) = \boldsymbol{u}$
11. $\mathcal{M}(\langle \boldsymbol{d}_2\rangle)(f_1) = \boldsymbol{u}$
12. $\mathcal{M}(\langle \boldsymbol{d}_2\rangle)(\mathsf{inconsistent}) = \boldsymbol{u}$
13. $\mathcal{M}(\langle \boldsymbol{d}_1\boldsymbol{d}_2\boldsymbol{d}_3\rangle)(\gamma) = \boldsymbol{u}$

Items 1,2,3, 4, and 5 hold due to the premises in $\mathcal{S}$*. Item 6 holds because of the effect of* α*. Items 7 and 8 are false because they can be safely assumed to be false in the minimal model. Items 9 and 10 are undefined because the postcondition of* α *is undefined. Item 11 is undefined because* c_1 *and* c_2 *are undefined. Item 12 is undefined because* f_1 *is undefined. And item 13 is undefined because part of its definition is undefined. Note that it is possible for* f_1 *to be undefined and* $\mathbf{neg}\, f_1$ *to be true in a path.* □

Example 4.5.16. *Consider the specification in Example 4.5.7 together with the following rule defining the fluent* inconsistent*.*

$$\mathsf{inconsistent} \leftarrow \mathsf{healthy}(P),\ \mathbf{neg}\, \mathsf{healthy}(P)$$

In the specification $\frac{\Lambda}{\mathbf{I}_\emptyset}$*, the sets* $\mathbf{P}$*, remain the same since they all are* **not** *-free. In* $\mathcal{E}$*, only the frame axioms change as follows*

$$\begin{aligned}
\mathsf{dnaT}(P) \otimes \mathsf{insert}_{\mathsf{takeT}}(P,T) \otimes \boldsymbol{u}^{\pi} &\rightarrow \mathsf{insert}_{\mathsf{takeT}}(P,T) \otimes \mathsf{dnaT}(P) \\
\mathsf{flu}(P) \otimes \mathsf{insert}_{\mathsf{takeT}}(P,T) \otimes \boldsymbol{u}^{\pi} &\rightarrow \mathsf{insert}_{\mathsf{takeT}}(P,T) \otimes \mathsf{flu}(P)
\end{aligned}$$

Since $\frac{\Lambda}{\mathbf{I}_\emptyset}$ *is* **not** *-free, it has a minimal model [88]* $\Gamma^{\uparrow 1}(\mathbf{I}_\emptyset) = \mathcal{I}_1$*. It follows from the construction of* $\mathcal{I}_1$ *that* $\mathcal{I}_1(\langle \boldsymbol{d}_1\rangle)(\mathsf{inconsistent}) = \boldsymbol{u}$*. It is not hard to see that in the WFM of* Λ*,* inconsistent *is also undefined in* $\mathcal{I}_1(\langle \boldsymbol{d}_1\rangle)$*. This is because the frame axioms are preventing the inconsistency from occurring, but it is still detected. Without the rules encoding the ontology,* inconsistent *would be false in* $WFM(\langle \boldsymbol{d}_1\rangle)$*.* □

Theorem 4.5.17. $WFM((\mathcal{E}, \mathbf{P}, \mathcal{S}))$ *is the least model of* $(\mathcal{E}, \mathbf{P}, \mathcal{S})$*.*

4.6 Production Systems in $\mathcal{TR}^{PAD}$

In this section we present the reduction of production systems augmented with Datalog-rewritable ontologies to $\mathcal{TR}^{PAD}$. Given an alphabet $\mathcal{L}_{\mathsf{PS}}$ for a production system PS, the corresponding language $\mathcal{L}_{\mathcal{TR}}$ of the target $\mathcal{TR}^{PAD}$ formulation will consist of symbols for rule labels, constants, and predicates. In addition, $\mathcal{L}_{\mathcal{TR}}$ has the following symbols:

- the *pdas* *add_used* and *clean_used*, and for every predicate $p \in \mathcal{L}_{\mathsf{PS}}$, *Ins_*$p$, and *del_*$p$;
- the compound action *act*;
- the defined fluent inconsistent, and for every rule label r the defined fluent fireable_r;
- the fluents inertial and used.

Intuitively, the *pdas* *Ins_*p, and *del_*p above represent the actions ***assert*** and ***retract***. The *pdas* *add_used* and *clean_used*, and the fluent used, are used to keep track of the assignments that has already been used to instantiate a `FOR`-production rule. The compound action *act* represents a generic production rule. The defined fluent fireable_r is true if the condition of the rule r holds and the action produces no inconsistencies. The defined fluent inconsistent is true, if there is an inconsistency in the state. The fluent inertial is used to distinguish inertial from non-inertial fluents.

Intuitively, the *pdas* *Ins_*p and *del_*p above represent the RIF actions ***assert*** and ***retract***. The *pdas* *add_used* and *clean_used*, and the fluent used, are used to keep track of the assignments that have already been used to instantiate a `For`-production rule.

The compound action *act* represents a generic rule. The defined fluent fireable_r is true if the condition of the rule r holds and the action produces no inconsistencies. The fluent inertial is used to distinguish inertial from non-inertial fluents.

Let $\psi = \alpha_1 \ldots \alpha_n$ be a sequence of atomic actions. We use $\widehat{\psi}$ to denote the $\mathcal{TR}$ -serial conjunction $\widehat{\psi} = \widehat{\alpha_1} \otimes \cdots \otimes \widehat{\alpha_n}$ where

$$\widehat{\alpha_i} = \begin{cases} \mathit{Ins\_}p(t_1 \ldots t_n) & \text{if } \alpha_i = \textbf{\textit{assert}}(p(t_1 \ldots t_n)) \\ \mathit{del\_}p(t_1 \ldots t_n) & \text{if } \alpha_i = \textbf{\textit{retract}}(p(t_1 \ldots t_n)) \end{cases}$$

Let $\phi = f_1 \wedge \cdots \wedge f_n \wedge l_1 \ldots l_m$ be a conjunction of atoms (f_i) and negative literals (l_j). Then let $\widehat{\phi}$ denote the $\mathcal{TR}$ -serial conjunction $\widehat{\phi} = f_1 \wedge \cdots \wedge f_m \wedge \sim l_1 \wedge \cdots \wedge \sim l_m$, where

$$\sim l_j = \begin{cases} \textbf{not}\, f(\vec{c}) & \text{if } l_j = \neg f(\vec{c}) \wedge f \in \mathcal{P}_{PS} \\ \textbf{neg}\, f(\vec{c}) & \text{if } l_j = \textbf{neg}\, f(\vec{c}) \wedge f \in \mathcal{P}_{DL} \end{cases}$$

In the following, let $\mathsf{PS} = (\mathcal{T}, \mathsf{L}, R)$ be a production system. For simplicity we assume that conditions in production rules are conjunction of fluent literals. In addition, we assume we have an ***initial working memory***, WM_0, that represents the knowledge we have about the initial state of the system. A production system coupled with a working memory is called a ***configuration***.

The reduction, Λ_{PS}, of a configuration $(\mathsf{PS}, \mathsf{WM}_0)$ to $\mathcal{TR}^{PAD}$ is a $\mathcal{TR}^{PAD}$ knowledge-base $(\mathcal{E}, \mathbf{P}, \mathcal{S})$ composed of the following PADs ($\mathcal{E}$), rules for defined fluents ($\mathbf{P}$) and premises ($\mathcal{S}$). From now on we assume that the ontology $\mathcal{T}$ is *Datalog-expressible* (e.g., $\mathcal{L}DL^{+}$—see Section 4.3.2).

1. **State identifiers**: There is an *initial* state identifier $\mathbf{d}_0$. The rest of the state identifiers are indexed by sequences of actions that need to be applied to $\mathbf{d}_0$ in order to reach those other states. That is, they have the form $\mathbf{d}_{0,\alpha_1,\ldots,\alpha_n}$, where $n \geqslant 0$ and each α_i is a ground instance of a $\mathcal{TR}^{PAD}$'s partially defined action *add_used*, *clean_used*, *Ins_p*, or *del_p*, for some p.

2. **Ontology $\mathcal{T}$:** $\mathbf{P}$ contains all the rules from the Datalog rendering of $\mathcal{T}$.

3. **Initial Database:** The premises below characterize the content of the initial working memory WM_0.

   - For each atomic literal $p(t_1, \ldots, t_n)$ in WM_0

     $\mathbf{d}_0 \rhd p(t_1, \ldots, t_n) \in \mathcal{S}$
     $\mathbf{d}_0 \rhd \mathsf{inertial}(p(t_1, \ldots, t_n)) \in \mathcal{S}$[8]

4. **Frame Axioms:** The following frame axioms encode the laws of inertia. In addition, they take care of the actual "removal" of $\mathcal{L}_{\mathsf{PS}}$ atoms from the working memory, and the cleaning of the used assignments.

   - For each predicate p and action α_q that involves assertion or retraction of a predicate q, where $p \neq q$:

$$\left\{\begin{array}{c} (\mathsf{inertial}(p(\vec{X})) \wedge p(\vec{X})) \otimes \alpha_q(\vec{Y}) \otimes \mathbf{not}\ \mathsf{inconsistent} \rightarrow \\ \alpha_q(\vec{Y}) \otimes (p(\vec{X}) \wedge \mathsf{inertial}(p(\vec{X}))) \end{array}\right\} \in \mathcal{E}$$

   - For each predicate p and action α_p on that predicate:

$$\left\{\begin{array}{c} (\mathsf{inertial}(p(\vec{X})) \wedge p(\vec{X}) \wedge \vec{X} \neq \vec{Y}) \otimes \alpha_p(\vec{Y}) \otimes \mathbf{not}\ \mathsf{inconsistent} \rightarrow \\ \alpha_p(\vec{Y}) \otimes (p(\vec{X}) \wedge \mathsf{inertial}(p(\vec{X}))) \end{array}\right\} \in \mathcal{E}$$

   - For each predicate p

$$\left\{\begin{array}{l} (\mathsf{inertial}(p(\vec{X})) \wedge p(\vec{X})) \otimes \mathit{add\_used}(\vec{Y}) \rightarrow \\ \qquad \mathit{add\_used}(\vec{Y}) \otimes (p(\vec{X}) \wedge \mathsf{inertial}(p(\vec{X}))) \\ (\mathsf{inertial}(p(\vec{X})) \wedge p(\vec{X}) \wedge p \neq \mathsf{used}) \otimes \mathit{clean\_used} \rightarrow \\ \qquad \mathit{clean\_used} \otimes (p(\vec{X}) \wedge \mathsf{inertial}(p(\vec{X}))) \end{array}\right\} \in \mathcal{E}$$

[8] We could have written this as $\mathsf{inertial}(p, t_1, \ldots, t_n)$ to avoid the appearance of being second order or that the use of function symbols here is essential.

5. **RIF Actions:** The following rules encode the RIF actions ***assert*** and ***retract*** in $\mathcal{TR}^{PAD}$:

   - *Insert:* For each predicate $p \in \mathcal{L}_{PS}$ (whether in $\mathcal{P}_{DL}$ or $\mathcal{P}_{PS}$):

$$\left\{ \begin{array}{l} \mathit{Ins\_p}(t_1, \ldots, t_n) \rightarrow \mathit{Ins\_p}(t_1, \ldots, t_n) \otimes \\ \qquad (p(t_1, \ldots, t_n) \wedge \mathsf{inertial}(p(t_1, \ldots, t_n))) \end{array} \right\} \in \mathcal{E}$$

   - *Retract:* For each predicate $p \in \mathcal{L}_{PS}$,

$$\text{if } p \in \mathcal{P}_{DL} \text{ then } \left\{ \begin{array}{l} \mathit{del\_p}(t_1, \ldots, t_n) \rightarrow \mathit{del\_p}(t_1, \ldots, t_n) \otimes \\ \qquad (\mathbf{neg}\, p(t_1 \ldots t_n) \wedge \mathsf{inertial}(\mathbf{neg}\, p(t_1 \ldots t_n))) \end{array} \right\} \in \mathcal{E}$$

   Recall that the effect of the *pda* $\mathit{del\_p}$ for PS atoms is given by the interaction with the frame axioms. For instance, if applying $\mathit{del}_{dnaT}(pcr)$ in $\mathbf{d}_1$ results in a state $\mathbf{d}_2$, it holds that $\mathbf{d}_2$ is equal to $\mathbf{d}_1$ except for $\mathsf{dnaT}(pcr)$, which is not carried to $\mathbf{d}_2$ by the frame axioms. This is equivalent to remove $\mathsf{dnaT}(pcr)$ from $\mathbf{d}_2$.

6. **Production rules:** The following rules encode the production rules.

   - For each `IF-THEN`-rule of the form "r: `Forall` $\vec{x}$: `if` $\phi_r(\vec{x})$ `then` $\psi_r(\vec{x})$"

$$r \leftarrow \mathsf{fireable\_r}(\vec{X}) \otimes \widehat{\psi}_r(\vec{X}) \quad \in \mathbf{P}$$

   - For each `FOR`-rule of the form "r: `For` $\vec{x}$: $\phi_r(\vec{x})$ `do` $\psi_r(\vec{x})$"

$$\left. \begin{array}{l} r \leftarrow \mathsf{fireable\_r}(\vec{X}) \otimes \widehat{\psi}_i(\vec{X}) \otimes \mathit{add\_used}(\vec{X}) \otimes \mathit{loop\_r} \\ \mathit{loop\_r} \leftarrow r \\ \mathit{loop\_r} \leftarrow (\mathit{not}\ \exists \vec{X} : \mathsf{fireable\_r}(\vec{X})) \otimes \mathit{clean\_used} \end{array} \right\} \in \mathbf{P}$$

   where $\mathit{not}\ \exists \vec{X} : \widehat{\phi}(\vec{X})$ above is a shorthand for $\mathbf{not}\ \mathsf{p}'$ such that p' is a new predicate defined as $\mathsf{p}' \leftarrow \widehat{\phi}(\vec{X})$.

7. **Auxiliary Actions and Premises:**

   - *Run-Premises*: Then for each pda α and a sequence ξ of actions *Ins*, *del*, *add_used*, or *clean_used*, the set of premises $\mathcal{S}$ contains the following run-premise:

$$\mathbf{d}_\xi \overset{a}{\rightsquigarrow} \mathbf{d}_{\xi,a}$$

   For example, $\mathbf{d}_{0,\mathit{Ins\_p}(c)} \overset{\mathit{Ins}_q(d)}{\rightsquigarrow} \mathbf{d}_{0,\mathit{Ins\_p}(c),\mathit{Ins}_q(d)}$.

- *Inconsistency*: For each predicate $p \in \mathcal{L}_{PS}$, $\mathbf{P}$ contains a rule of the form:

$$\mathsf{inconsistent} \leftarrow p(\vec{X}),\ \mathbf{neg}\, p(\vec{X})$$

- *Adding* used *assignments:*

$$\{\ \mathit{add\_used}(\vec{Y}) \rightarrow \mathit{add\_used}(\vec{Y}) \otimes \mathsf{used}(\vec{X})\ \} \in \mathcal{E}$$

- *Fireability*. The following rules are in $\mathbf{P}$:

If r is an `IF-THEN` rule $\begin{cases} \mathsf{fireable\_r}(\vec{X}) \leftarrow \widehat{\phi}_r(\vec{X}) \wedge \bigvee_{p \in \mathcal{P}} inertial(p(Y)) \wedge \\ (\lozenge \widehat{\psi}_r(\vec{X}) \otimes \mathbf{not}\, \mathsf{inconsistent} \wedge \mathbf{not}\, inertial(p(Y))) \end{cases}$

If r is a `FOR` rule $\begin{cases} \mathsf{fireable\_r}(\vec{X}) \leftarrow \widehat{\phi}_r(\vec{X}) \wedge \mathbf{not}\, \mathsf{used}(\vec{X})) \wedge (\lozenge \widehat{\psi}_r(\vec{X}) \otimes \\ \mathbf{not}\, \mathsf{inconsistent}) \end{cases}$

Observe that in the definition of fireable_r for `IF-THEN` rules, we also require that the effect of the action must produce a change, in particular in the rule above we require that it must retract some inertial fact. We omit the analogous definition requiring that the change consists of inserting a new inertial fact.

- *Random choice of action*: Suppose $\{r_1 \dots r_n\} = \mathsf{L}$

$$\left.\begin{array}{l} \mathit{act} \leftarrow r_1 \\ \vdots \\ \mathit{act} \leftarrow r_n \end{array}\right\} \in \mathbf{P}$$

To run k rules of the production system we use the transaction:

$$?-\ (\mathbf{d}_0)\ \underbrace{\mathit{act} \otimes \cdots \otimes \mathit{act}}_{k}$$

Theorem 4.6.1 (Soundness)**.** *Let $(\mathcal{E}, \mathbf{P}, \mathcal{S})$ be the $\mathcal{TR}^{\text{PAD}}$ embedding of a PS configuration. Suppose*

$$\mathcal{E}, \mathbf{P}, \mathcal{S}, \boldsymbol{d}_0 \dots \boldsymbol{d}_k \models \underbrace{\mathit{act} \otimes \cdots \otimes \mathit{act}}_{m}$$

Then there are working memories $\mathsf{WM}_1 \dots \mathsf{WM}_m$, and RIF rules $r_1 \dots r_m$ such that

$$\begin{array}{c} \mathsf{WM}_0 \overset{r_1}{\hookrightarrow} \mathsf{WM}_1 \\ \vdots \\ \mathsf{WM}_{m-1} \overset{r_m}{\hookrightarrow} \mathsf{WM}_m \end{array}$$

4.7 Summary

In this chapter we described a new semantics for the combination of production systems with *arbitrary* DL ontologies. Unlike previous approaches [90, 28, 26, 7, 67, 102], the semantics presented here supports extensions, like the *FOR*-loops or *while*-loops, that are not included in RIF-PRD, but are found in commercial production systems such as IBM's *JRules*[58]. In addition, our approach can handle inconsistencies produced by the interaction of rule actions and the ontology.

We also defined a sound embedding of such semantics, restricted to rule-based DL ontologies, into Transaction Logic with partial action definitions ($\mathcal{TR}^{PAD}$). This reduction gives a declarative semantics to the combination, and is considerably simpler and compact that other approaches, including [90, 67, 102, 26, 62].

To model production systems in $\mathcal{TR}^{PAD}$, we extended $\mathcal{TR}^{PAD}$ with default negation and defined the well-founded semantics [96] for it. It is worth noting that this $\mathcal{TR}^{PAD}$ embedding can be used as an implementation vehicle for the combination of PS and rule-based ontologies.

Chapter 5

Generalized Ontology-based Production Systems

5.1 Introduction

The goal of this chapter is twofold. On the one hand, we want to provide a very general framework for the combination of ontologies and production rules. On the other hand, we want to define specific systems combining production rules and ontologies and to define effective algorithms for the static analysis of such systems.

To reach the first of the above goals, in the first part of this chapter we define *generalized ontology-based production systems (GOPSs)*, which formalize a very general and powerful combination of ontologies and production systems. We show that GOPSs capture and generalize many existing formal notions of production systems. We introduce a powerful verification query language for GOPSs, which is able to express the most relevant formal properties of production systems previously considered in the literature. We establish a general sufficient condition for the decidability of answering verification queries over GOPSs.

Then, we turn our attention to the second of the above goals. Specifically,we define *Lite-GOPSs*, a particular class of GOPSs based on the use of a light-weight ontology language (*DL-Lite*$_A$), a light-weight ontology query language (*EQL-Lite(UCQ)*), and a tractable semantics for updates over Description Logic ontologies. We show decidability of all the above verification tasks over Lite-GOPSs, and prove tractability of some of such tasks.

5.1.1 Motivation and contribution

5.1.1.1 Motivation

The integration of ontologies and production rules is a challenging task. As illustrated by the survey presented in Chapter 4, many approaches have (either directly or indirectly) dealt with this problem in the recent literature [85, 27, 7, 62, 91, 29]. However, known approaches suffer from the following limitations: (i) there is no unifying approach that is able to capture all the above proposals within a coherent formal setting; (ii) no approach seems to be flexible enough to allow for the combination of *arbitrary* ontologies with production rules; (iii) there are very few results concerning the computational properties of such extended forms of production systems (an exception is [29]).

The goal of this chapter is to identify and explore a very general way of combining ontologies and production rules. We argue that there are at least two strong motivations for pursuing such a goal.

On the one hand, the existence of a framework which is general enough to capture the main existing proposals for the combination of ontologies and production rules makes it possible to easily and effectively study and compare the different proposals in a coherent formal setting.

On the other hand, this general framework makes it possible to identify and study decidability and complexity of reasoning in classes of systems combining ontologies and production rules. In particular, in this chapter we are interested in identifying classes of systems combining ontologies and production rules which allow for decidable static analysis of such systems. Indeed, as in other fields like databases, information systems, and software engineering, the availability of effective methods for static analysis would be an invaluable tool for the design and optimization of production systems.

5.1.1.2 Contribution

Our approach is based on a very abstract vision of an ontology, whose roots lie in the principles of knowledge representation [70]: we see the ontology as a knowledge base defined through a *functional specification*. More precisely, the ontology is equipped with a *query language* and an *update language*. Such languages are provided with a semantics given by a function ASK and a function $TELL$, respectively. The function ASK provides the semantics of queries posed to an ontology; the function $TELL$ provides the semantics of updates over an ontology.

According to this view of an ontology, the integration of ontologies with production rules is very simple and natural. Roughly speaking, production rules are "if *condition* then *action*" statements. Now, the functional specification of an ontology makes it very simple to define a combination of ontologies and production systems. This combination is based on the following, almost straightforward, considerations:

1. when combining ontologies and production rules, it is natural to use ontology query languages to express rule conditions and ontology update languages to express rule actions;

2. to interpret the meaning of such conditions and of such actions, it is natural to use the semantics of ontology queries and ontology updates, respectively.

Thus, production rules are executed on an ontology: the condition language corresponds to the ontology query language, while the action language corresponds to the ontology update language. The production system uses the ontology as a working memory, and the semantics of the system is then given by the usual operational semantics of production rules, using the ASK and $TELL$ functions to interpret conditions and actions, respectively.

Hence, this approach to the integration of ontologies and production rules is very natural. However, to effectively exploit such an approach, we have to face a big issue: in fact, very few ontology languages are equipped with a satisfactory query language and/or a satisfactory update language.[1] In particular, few results are available in the field of updating ontologies, in particular Description Logic (DL) ontologies.

So, this approach might seem just elegant but of no use: fortunately, this is not true, for at least two reasons. First of all, the functional view of ontologies makes it clear that the real technical obstacles towards the combination of ontologies and production systems are only due to the ontology component, in the sense that such obstacles are due to the fact that the specifications of ontologies are often still incomplete (they lack a proper query functionality and/or a proper update functionality). Moreover, some recent results allow us to identify ontology specifications that actually match the requirements for a meaningful combination with production rules. In fact, many expressive (and decidable) query languages have been defined for DL ontologies (e.g., [94, 20, 46]), and some recent approaches have proposed interesting semantics for updates over DL ontologies, as well as algorithms for effectively computing such updates (e.g., [30, 23, 68, 74]).

Concerning production rules, essentially we stick to the RIF-PRD specification [31], which has already been introduced in Chapter 4.

We formalize the above vision of the combination of ontologies and production systems as follows.

- We define *generalized ontology-based production systems (GOPSs)*, which formalize a very general and powerful combination of ontologies and production systems based on the functional specification of ontologies.

[1]Here, by *satisfactory* language we mean that both syntax (i.e., the expressiveness) and semantics of the language should be adequate. For instance, we argue that a semantics for the update action of inserting an axiom which corresponds to the simple syntactic addition of the axiom to the DL ontology is to be considered as unsatisfactory, since it is not coherent with the semantics of the ontology itself.

- We introduce a powerful verification query language for GOPSs, which is able to express the most relevant formal properties of production systems previously considered in the literature.
- We study reasoning over GOPSs and establish a general sufficient condition for the decidability of answering verification queries over GOPSs.
- Next, we turn our attention to specific ontology languages. More specifically, we define *Lite-GOPSs*, a particular class of GOPSs based on the use of a light-weight ontology language ($DL\text{-}Lite_A$), a light-weight ontology query language (*EQL-Lite(UCQ)*), and a tractable semantics for updates over Description Logic ontologies.
- We show decidability of all the above verification tasks over Lite-GOPSs, and prove tractability of some of such tasks.

With respect to the approach presented in Chapter 4, we highlight two main differences. First, the present approach has a more general flavour, since any decription logic ontology can be combined with production rules, while the approach of Chapter 4 is tailored for Datalog-rewritable ontologies. Second, and most important, the two approaches are based on different principles. In the approach of Chapter 4, the problem of defining a language for expressing conditions and effects of production rules and evaluating such conditions and effects on the ontology is solved *within* the approach itself. Conversely, in our approach, we start from the postulate that production rule conditions are ontology queries and production rule effects are ontology updates, therefore we are not allowed to define them: we must resort to (and reuse) semantics for ontology queries and updates to define the meaning of such conditions and effects. In other words, according to our view, an ontology comes equipped with its own languages and semantics for queries and updates, and the most natural and intuitive combination of the ontology with production rules is the one that uses the equipped languages and semantics for queries and updates over the ontology.

Notice also that, differently from both the approach described in Chapter 4 and analogous existing approaches (e.g., [27, 7, 91, 29]), we do not aim at reconstructing the whole production system into a logic: that is, our approach does not provide a declarative semantics for production systems through a logical representation of such systems. Nevertheless, we make use of logic in the verification of formal properties of production systems. In fact, as will be clear in Section 5.1.4, we will use model checking for the static analysis of production systems.

5.1.1.3 Structure of the chapter

The chapter is structured as follows. In the first part, we define generalized ontology-based production systems (GOPSs) and study such systems in their generality, without making any assumption on the ontology languages used to specify the ontology component of the GOPSs. In particular:

- In Section 5.1.2, we briefly introduce production systems and Description Logic (DL) ontologies.
- In Section 5.1.3, we formally define syntax and semantics of generalized ontology-based production system (GOPS).
- In Section 5.1.4, we define the verification query language which allows for analyzing the formal properties of GOPSs.
- In Section 5.1.5 we use the above verification query language to express the most relevant formal properties of GOPSs.
- In Section 5.1.6 we study reasoning over GOPSs. In particular, we provide sufficient conditions for the decidability of the problem of answering verification queries over GOPSs.

In the second part of the paper, we study a specific class of GOPSs, which we call Lite-GOPSs, based on a specific ontology language (*DL-Lite*$_{\mathcal{A}}$), a specific ontology query language (*EQL-Lite(UCQ)*) and a specific ontology update language ($\mathcal{UL}$*-Lite*). In particular:

- in Section 5.1.7 we introduce the ontology query language *EQL-Lite(UCQ).*
- In Section 5.1.8 we present a recent semantics for ontology updates over DL ontologies and the ontology update language $\mathcal{UL}$*-Lite*.
- In Section 5.1.9 we define the class of Lite-GOPSs, based on the choice of the above languages to specify the ontology component of a GOPS; moreover, we study reasoning over Lite-GOPSs, and present some decidability and complexity results for the problem of answering verification queries over Lite-GOPSs.
- Finally, in Section 5.1.10 we draw some conclusions and directions for further work.

5.1.2 Preliminaries

In this section we recall production systems, Description Logic ontologies, and the description logic *DL-Lite*$_{\mathcal{A}}$.

5.1.2.1 Production Systems

In the following, we briefly informally recall the notions of production system and production rule given in the RIF-PRD specification (for more details see [31]).

A production rule system (or production system) is a pair $\langle F_0, P\rangle$ where F_0 is a state of a *fact base* and P is a set of *production rules*. A state of a fact base is simply a set of facts (ground atomic formulas). A production rule is an expression of the form

$$\mathit{FORALL}\, \vec{x} : \mathit{IF}\ \phi(\vec{x})\ \mathit{THEN}\ \alpha_1(\vec{x}), \ldots, \alpha_n(\vec{x})$$

such that:

- $\phi(\vec{x})$ is a first-order formula, called *condition*, with free variables $\vec{x}$;
- every $\alpha_i(\vec{x})$ is such that for every *ground substitution* $\langle \vec{x}, \vec{c}\rangle$, i.e., a substitution of the rule variables with constants, $\alpha_i(\vec{c})$ is an *action*, i.e., an expression of the form `Assert`(f) or `Retract`(f), where f is a fact.

A *rule instance* is the variable-free rule obtained applying a ground substitution to a production rule. A *priority* is associated to every rule instance.

The semantics of production systems is expressed in terms of a transition system. Such a notion is based on the semantics of execution of a production rule over a fact base and on the notion of *conflict resolution strategy*.

Given a production rule p of the above form, a ground substitution $\langle \vec{x}, \vec{c}\rangle$, and a state of the fact base F, the rule instance $p(\vec{c})$ *matches* F if the formula $\phi(\vec{c})$ is satisfied by F.

The set of rule instances matching a state F of a fact base is called the *conflict set* of F.

The *execution* of $p(\vec{c})$ over F is a state of the fact base F' obtained from F by applying the sequence of fact addition (`Assert`) and fact deletion (`Retract`) actions corresponding to $\alpha_1(\vec{c}), \ldots, \alpha_n(\vec{c})$.

A *conflict resolution strategy* is a function that, given a conflict set and information on the previous history of the system, picks one rule instance among the ones in the conflict set.

The operational semantics of a production system is given by a *transition system*: roughly, every state[2] of such a transition system represents a state of the fact base plus the information needed by the conflict resolution strategy, and there is a transition from one state s to another state s' (labeled by the sequence of actions of $p(\vec{c})$) if the rule $p(\vec{c})$ is the one picked by the conflict resolution strategy in s and s' is the state resulting by the execution of rule $p(\vec{c})$ in s: in particular, the state F' of the fact base of s' is the state of the fact base resulting from the execution of rule $p(\vec{c})$ on the state F of the fact base of s. For more details on the semantics, we refer the reader to [31].

[2]This description is relative to the so-called *cyclic states* of the transition system: actually, there is another kind of states, the *transitional states*, in the transition system.

As above explained, the conflict resolution strategy is a central aspect of the semantics of production systems. The RIF-PRD specification formalizes a particular conflict resolution strategy, whose principles are the same as the conflict resolution principles of real prodution rule systems: the so-called `rif:forwardChaining` strategy.

The `rif:forwardChaining` strategy can be described as follows. Given a conflict set:

1. (*refraction step*) if a rule instance has been executed in a given state of the system, it is no longer eligible for execution as long as it satisfies the states of facts associated to all the subsequent system states; this property is called *refraction*. Therefore, all the refracted rule instances are removed from further consideration;

2. (*priority step*) the remaining rule instances are ordered by decreasing priority, and only the rule instances with the highest priority are kept for further consideration;

3. (*recency step*) the rule instances are ordered by the number of consecutive system states in which they have been in the conflict set, and only the most recent rule instances are kept for further consideration;

4. (*tie-break step*) any remaining tie is broken is some way, and a single rule instance is kept for firing.

5.1.2.2 Description Logic ontologies

Description Logics (DLs) [5] allow for expressing knowledge in terms of *atomic concepts*, i.e., unary predicates, and *atomic roles*, i.e., binary predicates. General concepts and roles are built through the constructs allowed in the DL: such constructs are usually expressible in first-order logic (FOL). A *DL ontology* is formed by a set of *assertions*, typically divided into a *TBox*, expressing intensional knowledge, and an *ABox*, expressing extensional knowledge. Again, usually such assertions can be expressed as FOL sentences (i.e., closed FOL formulas). Thus, in most cases DL ontologies can be seen as FOL theories (of specific forms). The only notable exceptions are those DLs that include some form of second-order constructs, such as transitive closure or fixpoints [5].

In the following, we will speak about a *specification language* for ontologies, about *queries* to an ontology, and about *updates* to an ontology. There are several approaches that define and study query languages over DL ontologies (e.g. [72, 22, 21, 76]) and updates over DL ontologies (e.g., [73, 30, 23, 68]).

5.1.2.3 The description logic *DL-Lite*$_{\mathcal{A}}$

The description logic *DL-Lite*$_{\mathcal{A}}$ [84] is a member of the *DL-Lite* family of tractable Description Logics. Such a family of DLs is characterized by the nice computational properties of all the main DL reasoning tasks, in particular query answering [21].

DL-Lite$_{\mathcal{A}}$ distinguishes concepts from *value-domains*, which denote sets of (data) values, and roles from *attributes*, which denote binary relations between objects and values. Concepts, roles, attributes, and value-domains in this DL are formed according to the following syntax:

$$\begin{array}{llll}
B \longrightarrow A \mid \exists Q \mid \delta(U) & & E \longrightarrow \rho(U) \\
C \longrightarrow B \mid \neg B & & F \longrightarrow \top_D \mid T_1 \mid \cdots \mid T_n \\
Q \longrightarrow P \mid P^- & & V \longrightarrow U \mid \neg U \\
R \longrightarrow Q \mid \neg Q & &
\end{array}$$

In such rules, A, P, and U respectively denote an atomic concept (i.e., a concept name), an atomic role (i.e., a role name), and an attribute name, P^- denotes the inverse of an atomic role, whereas B and Q are called basic concept and basic role, respectively. Furthermore, $\delta(U)$ denotes the *domain* of U, i.e., the set of objects that U relates to values; $\rho(U)$ denotes the *range* of U, i.e., the set of values that U relates to objects; $\top_D$ is the universal value-domain; $T_1, \ldots, T_n$ are n pairwise disjoint unbounded value-domains.

A *DL-Lite*$_{\mathcal{A}}$ ontology is a pair $\mathcal{O} = \langle \mathcal{T}, \mathcal{A} \rangle$, where $\mathcal{T}$ is the TBox and $\mathcal{A}$ the ABox. A *DL-Lite*$_{\mathcal{A}}$ TBox $\mathcal{T}$ is a finite set of assertions of the form

$$B \sqsubseteq C \qquad Q \sqsubseteq R \qquad E \sqsubseteq F \qquad U \sqsubseteq V \qquad (\mathsf{funct}\ Q) \qquad (\mathsf{funct}\ U)$$

From left to right, the first four assertions respectively denote inclusions between concepts, roles, value-domains, and attributes. In turn, the last two assertions denote functionality on roles and on attributes. In fact, in *DL-Lite*$_{\mathcal{A}}$ TBoxes we further impose that roles and attributes occurring in functionality assertions cannot be specialized (i.e., they cannot occur in the right-hand side of inclusions). Let B_1 and B_2 be basic concepts, and let Q_1 and Q_2 be basic roles. We call *positive inclusions (PIs)* assertions of the form $B_1 \sqsubseteq B_2$, and of the form $Q_1 \sqsubseteq Q_2$, whereas we call *negative inclusions (NIs)* assertions of the form $B_1 \sqsubseteq \neg B_2$ and $Q_1 \sqsubseteq \neg Q_2$.

A *DL-Lite*$_{\mathcal{A}}$ ABox $\mathcal{A}$ is a finite set of membership assertions of the forms $A(a)$, $P(a, b)$, and $U(a, v)$, where A, P, and U are as above, a and b belong to Γ_O, the subset of Γ_C containing object constants, and v belongs to Γ_V, the subset of Γ_C containing value constants, where $\{\Gamma_O, \Gamma_V\}$ is a partition of Γ_C.

The semantics of a *DL-Lite*$_{\mathcal{A}}$ ontology is given in terms of first-order logic (FOL) interpretations $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$. $\Delta^{\mathcal{I}}$ is a non-empty domain such that $\Delta^{\mathcal{I}} = \Delta_V \cup \Delta_O^{\mathcal{I}}$, where $\Delta_O^{\mathcal{I}}$ is the domain used to interpret object constants in Γ_O, and Δ_V is the fixed domain (disjoint from $\Delta_O^{\mathcal{I}}$) used to interpret data values. $\cdot^{\mathcal{I}}$ is an interpretation function defined as follows:

$$\begin{array}{lcl@{\qquad}lcl}
A^{\mathcal{I}} & \subseteq & \Delta_O^{\mathcal{I}} & P^{\mathcal{I}} & \subseteq & \Delta_O^{\mathcal{I}} \times \Delta_O^{\mathcal{I}} \\
(\delta(U))^{\mathcal{I}} & = & \{\, o \mid \exists v.\, (o, v) \in U^{\mathcal{I}} \,\} & (P^-)^{\mathcal{I}} & = & \{\, (o, o') \mid (o', o) \in P^{\mathcal{I}} \,\} \\
(\exists Q)^{\mathcal{I}} & = & \{\, o \mid \exists o'.\, (o, o') \in Q^{\mathcal{I}} \,\} & (\neg Q)^{\mathcal{I}} & = & (\Delta_O^{\mathcal{I}} \times \Delta_O^{\mathcal{I}}) - Q^{\mathcal{I}} \\
(\neg B)^{\mathcal{I}} & = & \Delta_O^{\mathcal{I}} - B^{\mathcal{I}} & U^{\mathcal{I}} & \subseteq & \Delta_O^{\mathcal{I}} \times \Delta_V \\
\top_D^{\mathcal{I}} & = & \Delta_V & (\neg U)^{\mathcal{I}} & = & (\Delta_O^{\mathcal{I}} \times \Delta_V) - U^{\mathcal{I}} \\
(\rho(U))^{\mathcal{I}} & = & \{\, v \mid \exists o.\, (o, v) \in U^{\mathcal{I}} \,\} & & &
\end{array}$$

Notice that each $(T_i)^{\mathcal{I}}$ and each $(v)^{\mathcal{I}}$ are the same in every interpretation. An interpretation $\mathcal{I}$ satisfies a concept (resp., role) inclusion assertion $B \sqsubseteq C$ (resp., $Q \sqsubseteq R$) if $B^{\mathcal{I}} \subseteq C^{\mathcal{I}}$ (resp., $Q^{\mathcal{I}} \subseteq R^{\mathcal{I}}$). Furthermore, a role functionality assertion $(\mathsf{funct}\ Q)$ is satisfied by $\mathcal{I}$ if, for each $o, o', o'' \in \Delta_O^{\mathcal{I}}$, we have that $(o, o') \in Q^{\mathcal{I}}$ and $(o, o'') \in Q^{\mathcal{I}}$ implies $o' = o''$. The semantics for attribute and value-domain inclusion assertions, and for functionality assertions over attributes can be defined analogously. As for the semantics of ABox assertions, we say that $\mathcal{I}$ satisfies the ABox assertions $A(a)$, $P(a, b)$ and $U(a, v)$ if $a^{\mathcal{I}} \in A^{\mathcal{I}}$, $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in P^{\mathcal{I}}$ and $(a^{\mathcal{I}}, v^{\mathcal{I}}) \in U^{\mathcal{I}}$, respectively. Furthermore, in *DL-Lite*$_{\mathcal{A}}$ the Unique Name Assumption (UNA) is adopted, i.e., in every interpretation $\mathcal{I}$, and for every pair $c_1, c_2 \in \Gamma_C$, if $c_1 \neq c_2$ then $c_1^{\mathcal{I}} \neq c_2^{\mathcal{I}}$.

We denote with $\mathit{Mod}(\mathcal{O})$ the set of models of an ontology $\mathcal{O}$, i.e., the set of FOL interpretations that satisfy both TBox and ABox assertions in $\mathcal{O}$. As usual, an ontology $\mathcal{O}$ entails a FOL sentence ϕ, denoted $\mathcal{O} \models \phi$, if $\phi^{\mathcal{I}}$ is true in every $\mathcal{I} \in \mathit{Mod}(\mathcal{O})$.

An atomic concept A in $\mathcal{T}$ is unsatisfiable if $\mathcal{T} \models A \sqsubseteq \neg A$, i.e., if the interpretation of A is empty in every model of $\mathcal{T}$. Analogously, we say that an atomic role P is unsatisfiable in $\mathcal{T}$ if $\mathcal{T} \models P \sqsubseteq \neg P$, and a concept attribute U is unsatisfiable in $\mathcal{T}$ if $\mathcal{T} \models U \sqsubseteq \neg U$.

5.1.3 Generalized Ontology-Based Production Systems

In this section we define syntax and semantics of generalized ontology-based production systems (GOPSs).

5.1.3.1 Syntax

We start from the following pairwise disjoint alphabets: an alphabet of predicates Pred, an alphabet of constants Const, an alphabet of variables Var and an alphabet of production rule identifiers RuleID.

Our notion of production system builds on three languages over the above alphabets Pred, Const and Var:

- an *ontology specification language* $\mathcal{OL}$ which specifies the syntax of the ontology;
- an *ontology query language* $\mathcal{QL}$ which defines the queries over the ontology;
- an *ontology update language* $\mathcal{UL}$ which defines the updates over the ontology.

An *ontology* is a set of formulas from $\mathcal{OL}$. An *ontology query* is a formula of $\mathcal{QL}$. An *ontology update* is a formula of $\mathcal{UL}$.

An ontology query with no occurrences of free variables is called a *Boolean* ontology query. An ontology update with no occurrences of free variables is called a *ground* ontology update.

We denote by $\phi(\vec{x})$ a formula ϕ containing the free variables $\vec{x}$ ($\vec{x}$ is a tuple of variable symbols).

Given an n-tuple of variables $\vec{x}$, a *ground substitution* of $\vec{x}$ is a pair $\langle\vec{x},\vec{c}\rangle$, where $\vec{c}$ is an n-tuple of constants from $Const$. Given a formula with n free variables $\phi(\vec{x})$, and an n-tuple of constants $\vec{c}$, the formula $\phi(\vec{c})$ obtained from ϕ by applying the ground substitution $\langle\vec{x},\vec{c}\rangle$, i.e., by replacing the variables in $\vec{x}$ with the corresponding constants in $\vec{c}$, is called a *grounding* of $\phi(\vec{x})$.

Definition 5.1.1. *A* Generalized Ontology-Based Production System (GOPS) $\mathcal{G}$ *is a pair* $\langle\mathcal{O}_{in},\mathcal{P}\rangle$ *where:*

- $\mathcal{O}_{in}$ *is an ontology;*
- $\mathcal{P}$ *is a set of* production rules. *A production rule is an expression of the form*

$$FORALL\,\vec{x} : IF\ \phi(\vec{x})\ THEN\ \alpha_1(\vec{x}),\ldots,\alpha_n(\vec{x}) \tag{5.1}$$

  such that:

  - $\phi(\vec{x})$ *is an ontology query over* $\mathcal{O}$ *with free variables* $\vec{x}$*. The variables in* $\vec{x}$ *are symbols from* Var *and are called the* variables *of the production rule;*
  - *every* $\alpha_i(\vec{x})$ *is such that every grounding* $\alpha_i(\vec{c})$ *of* $\alpha_i(\vec{x})$ *is an ontology update.*

  Every production rule has an associated rule identifier*, i.e., a symbol from* $RuleID$ *which is associated with no other production rule.*

A *ground* production rule is a production rule without variables, i.e., an expression of the form

$$IF\ \phi\ THEN\ \alpha_1,\ldots,\alpha_n \tag{5.2}$$

where ϕ is an ontology query without free variables and every α_i is an ontology update.

Given a production rule with identifier p of the form (5.1), an *instance* of rule p is a ground production rule, with identifier $p(\vec{c})$, obtained from (5.1) by applying a ground substitution $\langle\vec{x},\vec{c}\rangle$, i.e., $p(\vec{c})$ is a ground production rule of the form

$$IF\ \phi(\vec{c})\ THEN\ \alpha_1(\vec{c}),\ldots,\alpha_n(\vec{c})$$

5.1.3.2 Semantics

The semantics of GOPSs is based on three functions: (i) the function ASK, which provides the semantics of ontology queries; (ii) the function $TELL$, which provides the semantics of ontology updates; (iii) the function Φ, which governs the execution of production rules. These functions are introduced in the following.

5.1.3.2.1 Ontology queries

The semantics of ontology queries is defined by the function $ASK(\mathcal{O}, \phi(\vec{x}))$. For every ontology $\mathcal{O}$ and for every ontology query $\phi(\vec{x})$ with free variables $\vec{x}$, $ASK(\mathcal{O}, \phi(\vec{x}))$ is a set of ground substitutions for $\vec{x}$.

Notice that, if ϕ has no free variables, i.e., ϕ is a Boolean query, then $ASK(\mathcal{O}, \phi)$ is either the set of ground substitutions containing the empty ground substitution $\{\emptyset\}$ (which means that the query ϕ is entailed by $\mathcal{O}$) or the empty set of ground substitutions $\emptyset$ (which means that ϕ is not entailed by $\mathcal{O}$).

5.1.3.2.2 Ontology updates

The semantics of ontology updates is defined by the partial function $TELL(\mathcal{O}, \alpha)$. More precisely, given an ontology $\mathcal{O}$ and an ontology update α, $TELL(\mathcal{O}, \alpha)$ is either undefined or is equal to one ontology $\mathcal{O}'$.

Intuitively, the case when $TELL(\mathcal{O}, \alpha)$ is undefined encodes those situations in which it is impossible (according to the intended semantics of ontology updates) to update the ontology $\mathcal{O}$ according to the update action α.

5.1.3.2.3 Production rules

Given a ground production rule p_g of the form (5.2) and an ontology $\mathcal{O}$, we say that p_g is *fireable* in $\mathcal{O}$ if the following conditions hold:

1. $ASK(\phi, \mathcal{O}) \neq \emptyset$;
2. there exists a sequence of ontologies $\mathcal{O}_0, \ldots, \mathcal{O}_n$ such that $\mathcal{O}_0 = \mathcal{O}$ and, for every i such that $0 \leqslant i \leqslant n-1$, $TELL(\mathcal{O}_i, \alpha_i)$ is defined and is equal to the ontology $\mathcal{O}_{i+1}$.

Given a GOPS $\mathcal{G}$ and an ontology $\mathcal{O}$, we denote by $CS(\mathcal{O}, \mathcal{G})$ the *conflict set* for $\mathcal{O}$ and $\mathcal{G}$, i.e., the set of instances p_g of the production rules from $\mathcal{P}$ such that p_g is fireable in $\mathcal{O}$.

5.1.3.2.4 GOPS

A *GOPS graph* $GG = \langle N, S^{in}, E, L^e \rangle$ is a directed graph where: N is the set of nodes S, called ***GOPS states***, which are pairs of the form $\langle id, \mathcal{O} \rangle$ where id is the *state identifier* and $\mathcal{O}$ is an ontology; S^{in} is a node of N, called the *initial state* of GG; E is the set of edges, i.e., pairs of GOPS states; and L^e is function which labels the edges with rule instance identifiers.

A *GOPS path* is a path of a GOPS graph.

A *(partial) conflict resolution function* is a function Φ over paths of a GOPS graph. Specifically, let $\mathcal{G}$ be a GOPS and let π be a GOPS path. Let $S_e = \langle id, \mathcal{O} \rangle$ be the ending state of π. Then, the value of $\Phi(\mathcal{G}, \pi)$ is $\emptyset$ if $CS(\mathcal{O}, \mathcal{G}) = \emptyset$; otherwise, the value of $\Phi(\mathcal{G}, \pi)$ is a non-empty set of ground production rules $\mathcal{P}_g$ such that $\mathcal{P}_g \subseteq CS(\mathcal{O}, \mathcal{G})$. If the value of the function Φ is always either the empty set or a singleton set (i.e., Φ selects at most one rule among the ones in $CS(\mathcal{O}, \mathcal{G})$), then we call Φ a *total conflict resolution function*.

Informally, a partial conflict resolution function selects a subset of the rule instances in the conflict resolution set $CS(\mathcal{O}, \mathcal{G})$. The intuitive meaning of such a function is that it chooses a subset of the rules in the conflict set: any of the rules in such a subset could be chosen for execution. On the other hand, a total conflict resolution function actually resolves the conflict among the rules, since it just selects one rule. While real production systems only adopt total conflict resolution functions, we introduce partial conflict resolution functions in our framework because this allows us to formally study the properties of *families* of conflict resolution stategies. For instance, the `rif:forwardChaining` strategy described in Section 5.1.2.1 can be seen as a partial conflict resolution function, since it does not actually specify the final *tie-break* step. So, studying GOPS under this partial conflict resolution function makes it possible to verify the formal properties of the `rif:forwardChaining` strategy independently of the particular implementation of the tie-break step.

Notice also that the conflict resolution function does not only depend on the current state of the ontology (i.e., the ontology labeling the final state of the GOPS path), but depends on a GOPS path, which represents the whole "history" of the GOPS evolution. This reflects the conflict resolution strategies adopted in real systems: e.g., the above described `rif:forwardChaining` strategy depends not only on the current state of the fact base, but also on the previous states of the transition system.

The semantics of a GOPS $\mathcal{G}$ is defined by the notion of transition system.

Definition 5.1.2. *Given a GOPS $\mathcal{G} = \langle \mathcal{O}_{in}, \mathcal{P} \rangle$, the* transition system of $\mathcal{G}$*, denoted by $TS(\mathcal{G})$, is the GOPS graph $\langle N, S^{in}, E, L^e \rangle$ defined inductively as follows:*

1. *the initial state S^{in} is the state $\langle S^{\mathcal{G}}_{in}, \mathcal{O}_{in} \rangle$;*
2. *for every state S of the form $\langle id, \mathcal{O} \rangle$ belonging to N, let $\pi(S)$ be the path of $TS(\mathcal{G})$ starting in $S^{\mathcal{G}}_{in}$ and ending at S. If p_g is the identifier of a rule instance such that $p_g \in \Phi(\mathcal{G}, \pi(S))$ then $\langle S, \langle p_g(id), \mathcal{O}' \rangle \rangle \in N$ with $\mathcal{O}' = EXEC(p_g, \mathcal{O})$, $\langle S, \langle p_g(id), \mathcal{O}' \rangle \rangle \in E$, and $L^e(\langle S, \langle p_g(id), \mathcal{O}' \rangle \rangle = p_g$.*

Informally, $TS(\mathcal{G})$ is the GOPS graph built starting from the initial state and adding, for every state S, an edge (transition) and a new state for every rule instance selected by the conflict resolution function $\Phi((\mathcal{G}, \pi(S))$: the new state is the state obtained by the execution of the rule instance on state S.

We call *run* of $\mathcal{G}$ any path of $TS(\mathcal{G})$ starting at the state whose identifier is $S^{\mathcal{G}}_{in}$ and such that the path either is infinite or ends at a sink node (i.e., a node which does not have any outcoming edge). Of course, if Φ is a total conflict resolution function, then $TS(\mathcal{G})$ contains only one run.

Intuitively, the transition system $TS(\mathcal{G})$ represents all the possible runs of $\mathcal{G}$, i.e., all the possible sequences of execution of production rule instances starting from the initial ontology.

Of course, the transition system may be infinite, since there may be infinite runs of $\mathcal{G}$.

5.1.4 Verification query language

In this section we define the verification query language $\mathcal{V}(\mathcal{QL})$. The language $\mathcal{V}(\mathcal{QL})$ is an extension of μ-calculus [36] where state formulas are formulas of the ontology query language $\mathcal{QL}$. This language builds from an analogous previous proposal [24] in the field of artifact-centric services.

To specify dynamic properties we will use μ-calculus [36] which is one of the most powerful temporal logics for which model checking has been investigated, and indeed is able to express both linear time logics, as LTL, and branching time logics such as CTL or CTL* [25]. In particular, we need to introduce a variant of μ-calculus that conforms with the basic assumption of our formalism: the use of ontologies and ontology queries to describe the properties of a state.

To define verification queries, we need a further alphabet, the alphabet of predicate variables $\mathcal{PV}$, which is pairwise disjoint with all the alphabets introduced in Section 5.1.3.

Definition 5.1.3. *A* verification query *is specified by the following abstract syntax:*

$$\psi ::= \phi \mid \neg\psi \mid \psi_1 \wedge \psi_2 \mid [p]\psi \mid \langle p\rangle\psi \mid \mu X.\psi \mid \nu X.\psi$$

where ϕ is a Boolean ontology query (i.e., a formula from $\mathcal{QL}$ without free variables), X is a predicate variable symbol from $\mathcal{PV}$, and p is: (i) a rule instance identifier; or (ii) a rule identifier; or (iii) the symbol $-$.

The verification query language $\mathcal{V}(\mathcal{QL})$ *is the language of verification queries.*

The symbols μ and ν can be considered as quantifiers, and we make use of the notions of scope, bound and free occurrences of variables, closed formulas, etc. The definitions of these notions are the same as in first-order logic, treating μ and ν as quantifiers. For formulas of the form $\mu X.\psi$ and $\nu X.\psi$, we require the *syntactic monotonicity* of ψ wrt X: Every occurrence of the variable X in ψ must be within the scope of an even number of negation signs. In μ-calculus, given the requirement of syntactic monotonicity, the least fixpoint $\mu X.\psi$ and the greatest fixpoint $\nu X.\psi$ always exist.

Let $GG = \langle N, S^{in}, E, L^e\rangle$ be a GOPS graph. A valuation on GG is a mapping from the predicate variables appearing in ψ to subsets of N.

We assign meaning to $\mathcal{V}(\mathcal{QL})$ formulas by an *evaluation function* $Eval_{GG}(\cdot)$, which maps $\mathcal{V}(\mathcal{QL})$ formulas to subsets of N.

In the following, id represents a rule identifier, and, as explained in Section 5.1.3, the symbol $id(\vec{c})$ represents the identifier of an instance of the rule id.

Definition 5.1.4. *The* evaluation function $Eval_{GG}(\cdot)$ *for a GOPS graph* $GG = \langle N, S^{in}, E, L^e\rangle$ *is defined inductively as follows:*

$$\begin{array}{l}
Eval_{GG}(\phi) = \{s = \langle s_{id}, \mathcal{O}\rangle \mid s \in N \textit{ and } ASK(\phi, \mathcal{O}) \neq \emptyset\} \\
Eval_{GG}(X) = N' \subseteq N \\
Eval_{GG}(\neg\psi) = N - Eval_{GG}(\psi) \\
Eval_{GG}(\psi_1 \wedge \psi_2) = Eval_{GG}(\psi_1) \cap Eval_{GG}(\psi_2) \\
Eval_{GG}(\langle id(\vec{c})\rangle\psi) = \{s \in N \mid \exists s'.\ \langle s, s'\rangle \in E \textit{ and } L^e(s, s') = id(\vec{c}) \textit{ and } s' \in Eval_{GG}(\psi)\} \\
Eval_{GG}([id(\vec{c})]\psi) = \{s \in N \mid \forall s'.\ (\langle s, s'\rangle \in E \textit{ and } L^e(s, s') = id(\vec{c})) \textit{ implies } s' \in Eval_{GG}(\psi)\} \\
Eval_{GG}(\langle id\rangle\psi) = \{s \in N \mid \exists s', \vec{c}.\ \langle s, s'\rangle \in E \textit{ and } L^e(s, s') = id(\vec{c}) \textit{ and } s' \in Eval_{GG}(\psi)\} \\
Eval_{GG}([id]\psi) = \{s \in N \mid \forall s', \vec{c}.\ (\langle s, s'\rangle \in E \textit{ and } L^e(s, s') = id(\vec{c})) \textit{ implies } s' \in Eval_{GG}(\psi)\} \\
Eval_{GG}(\langle -\rangle\psi) = \{s \in N \mid \exists s'.\ \langle s, s'\rangle \in E \textit{ and } s' \in Eval_{GG}(\psi)\} \\
Eval_{GG}([-]\psi) = \{s \in N \mid \forall s'.\ \langle s, s'\rangle \in E \textit{ implies } s' \in Eval_{GG}(\psi)\} \\
Eval_{GG}(\mu X.\psi) = \bigcap\{N' \subseteq N \mid Eval_{GG}[X|N'](\psi) \subseteq N'\ \} \\
Eval_{GG}(\nu X.\psi) = \bigcup\{N' \subseteq N \mid N' \subseteq\ \ Eval_{GG}[X|N'](\psi)\}
\end{array}$$

where $Eval_{GG}[X|N'](\psi)$ *represents the value* $Eval_{GG}(()\psi)$ *when every occurrence of the predicate variable* X *is evaluated as the set of states* N'.

Intuitively, the evaluation function $Eval_{GG}(\cdot)$ assigns to the various constructs of μ-calculus the following meanings:

- The boolean connectives have the expected meaning.
- The evaluation of $\langle id(\vec{c})\rangle\psi$ includes the states $s \in N$ such that at state s there is an execution of the rule instance $id(\vec{c})$ that leads to a state s' included in the evaluation of ψ. Thus, the intuitive meaning of $\langle id(\vec{c})\rangle\psi$ is "there exists an execution of rule instance $id(\vec{c})$ that leads to a state where ψ holds".
- The evaluation of $[id(\vec{c})]\psi$ includes the states $s \in N$ such that each execution of the rule instance $id(\vec{c})$ at state s leads to some state s' included in the evaluation of ψ. Thus, the intuitive meaning of the operator $[id(\vec{c})]$ is "every execution of the rule instance $id(\vec{c})$ leads to a state where ψ holds".[3]
- The evaluation of $\langle id\rangle\psi$ includes the states $s \in N$ such that at state s there is an execution of any instance of the rule id that leads to a state s' included in the evaluation of ψ. Thus, the intuitive meaning of $\langle id\rangle\psi$ is "there exists an execution of an instance of rule id that leads to a state where ψ holds".
- The evaluation of $[id]\psi$ includes the states $s \in N$ such that each execution of any instance of the rule id at state s leads to some state s' included in the evaluation of ψ. Thus, the intuitive meaning of the operator $[id]$ is "every execution of an instance of rule id leads to a state where ψ holds".

[3]Notice that the present framework does not actually allow for *nondeterministic* execution of rule instances: i.e., the execution of a rule instance in a state always produces one successor state. Consequently, no transition system of a GOPS may contain two outcoming edges labeled by the same rule instance identifier, and therefore the formula $[id(\vec{c})]\psi$ appears of no use. However, the semantics of the verification query language is defined over generic GOPS graphs, in which the formula $[id(\vec{c})]\psi$ might make sense.

- The evaluation of $\langle - \rangle \psi$ includes the states $s \in N$ such that at state s there is an execution of an arbitrary rule instance that leads to a state s' included in the evaluation of ψ. Thus, the intuitive meaning of $\langle - \rangle \psi$ is "there exists an execution of a rule instance that leads to a state where ψ holds".
- The evaluation of $[-]\psi$ includes the states $s \in N$ such that each execution of any arbitrary rule instance at state s leads to some state s' included in the evaluation of ψ. Thus, the intuitive meaning of the operator $[-]$ is "every execution of rule instances leads to a state where ψ holds".
- The evaluation of $\mu X.\psi$ is the *smallest subset* N' of N such that, assigning to X the evaluation N', the resulting evaluation of ψ is contained in N'.
- Similarly, the evaluation of $\nu X.\psi$ is the *greatest subset* N' of N such that, assigning to X the evaluation N', the resulting evaluation of ψ contains N'.

The reasoning problem we are interested in is *model checking*. Let $GG = \langle N, S^{in}, E, L^e \rangle$ be a GOPS graph, let $s \in N$ be one of its states, and let ψ be a verification query. The related model checking problem is to verify whether $s \in \mathit{Eval}_{GG}(\psi)$.

In particular, in the following we focus on the Boolean problem of verifying whether the *initial state* S^{in} is in the evaluation of a verificaton query ψ. Thus, for every verification query ψ and for every GOPS graph $GG = \langle N, S^{in}, E, L^e \rangle$, we define $\mathit{Entailed}(\psi, GG)$ as true if $S^{in} \in \mathit{Eval}_{GG}(\psi)$, and false otherwise.

It is immediate to verify that model checking of verification queries is decidable if answering ontology queries is decidable. In particular, the following property holds.

Theorem 5.1.5. *Checking a verification query ψ over a finite GOPS graph $GG = \langle N, S^{in}, E, L^e \rangle$ can be done in time*

$$O((|GG| \cdot |\psi|)^k)$$

where $|GG| = |N| + |E|$, i.e., the number of states plus the number of transitions of GG, $|\psi|$ is the size of formula ψ (in fact, considering propositional formulas as atomic), and k is the number of nested fixpoints, i.e., fixpoints whose variables are one within the scope of the other, using an oracle for deciding $ASK(\phi, \mathcal{O}) \neq \emptyset$ for every verification query ϕ and ontology $\mathcal{O}$.

Proof. We can use the standard μ-calculus model checking algorithm [36], with the proviso that atomic formulas are now ontology queries ϕ, therefore for evaluating an atomic formula Q in a state $\langle id, \mathcal{O} \rangle$ we use an oracle that decides whether $ASK(\phi, \mathcal{O}) \neq \emptyset$. $\square$

Finally, we introduce a notion of bisimilarity for GOPS graph which will be very important for establishing decidability results on answering verification queries over GOPSs.

Let $S_1 = \langle id_1, \mathcal{O}_1 \rangle$, $S_2 = \langle id_2, \mathcal{O}_2 \rangle$ be two GOPS states. We say that S_1 and S_2 are *locally bisimilar* if, for every ontology query ϕ, $ASK(\phi, \mathcal{O}_1) = ASK(\phi, \mathcal{O}_2)$.

Given two GOPS graphs $GG_1 = \langle N_1, S_1^{in}, E_1, L_1^e \rangle$, $GG_2 = \langle N_2, S_2^{in}, E_2, L_2^e \rangle$, we say that GG_1 and GG_2 are *bisimilar* if there exists a function β from the states of GG_1 to the states of GG_2 such that $\beta(S_1^{in}) = S_2^{in}$ and, for every state S of GG_1, the following conditions hold:

1. S and $\beta(S)$ are locally bisimilar;
2. for each state S' such that $\langle S, S' \rangle$ is an edge of GG_1, $\langle \beta(S), \beta(S') \rangle$ is an edge of GG_2, and $L_1^e(\langle S, S' \rangle) = L_2^e(\langle \beta(S), \beta(S') \rangle)$;
3. for each state S'' such that $\langle \beta(S), S'' \rangle$ is an edge of GG_2, there exists a state S' of GG_1 such that $\langle S, S' \rangle$ is an edge of GG_1, $S'' = \beta(S')$, and $L_1^e(\langle S, S' \rangle) = L_2^e(\langle \beta(S), \beta(S') \rangle)$.

Theorem 5.1.6. *If GG_1 and GG_2 are bisimilar GOPS graphs, then for every verification query ψ, Entailed$(\psi, GG_1) =$ Entailed(ψ, GG_2).*

Proof. The proof is analogous to the standard proof of bisimulation invariance of μ-calculus, see, e.g., [11]. □

Given a GOPS $\mathcal{G}$ and a verification query ψ, we say that *ψ is entailed by $\mathcal{G}$* if $\mathit{Entailed}(\psi, TS(\mathcal{G})) = \mathit{true}$. Moreover, we define $Ans_{GOPS}(\psi, \mathcal{G})$ as $\mathit{Entailed}(\psi, TS(\mathcal{G}))$. The problem of *answering a verification query ψ over a GOPS $\mathcal{G}$* amounts to establishing whether $Ans_{GOPS}(\psi, \mathcal{G}) = \mathit{true}$.

5.1.5 Formal properties of GOPS

In this section we show that the verification query language defined in Section 5.1.4 is a very powerful tool for the static analysis of GOPSs. In particular, using the verification query language $\mathcal{V}(\mathcal{QL})$, we provide a formalization of some interesting and complex formal properties of GOPSs.

In the following, we refer to a GOPS $\mathcal{G}$ and use the symbol $\mathcal{I}_{\mathcal{G}}$ to denote the set of production rule identifiers of the GOPS $\mathcal{G}$.

Moreover, without loss of generality, we assume that the language allows for expressing a Boolean ontology query which holds in every ontology, and we denote by $TRUE$ such a query (and denote by $FALSE$ the formula $\neg TRUE$). In fact, even in the case when the ontology query language does not allow for such a tautological query, it is possible to easily modify the specification of the GOPS $\mathcal{G}$ in a way such that there exists a Boolean ontology query that holds in every state of the transition system of $\mathcal{G}$: this kind of "locally tautological" ontology query for $\mathcal{G}$ is enough for our purposes.

The following list shows the formalization in terms of $\mathcal{V}(\mathcal{QL})$ queries of a set of interesting properties of a GOPS $\mathcal{G}$:

- *Every run terminates in a state where the ontology query ϕ holds* if and only if $\mathit{Entailed}(\psi, \mathit{TS}(\mathcal{G})) = \mathit{true}$, where ψ is the formula
$$\psi = \mu X.(([-]\mathit{FALSE} \wedge \phi) \vee (\langle - \rangle \mathit{TRUE} \wedge [-]X))$$
- *Every run is finite* if and only if $\mathit{Entailed}(\psi, \mathit{TS}(\mathcal{G})) = \mathit{true}$, where ψ is the formula
$$\psi = \mu X.[-]X$$
- *An ontology query ϕ eventually holds forever in some run* if and only if $\mathit{Entailed}(\psi, \mathit{TS}(\mathcal{G})) = \mathit{true}$, where ψ is the formula
$$\psi = \mu X.((\nu Y.\phi \wedge \langle - \rangle Y) \vee \langle - \rangle X)$$
- *An ontology query ϕ eventually holds forever in every run* if and only if $\mathit{Entailed}(\psi, \mathit{TS}(\mathcal{G})) = \mathit{true}$, where ψ is the formula
$$\psi = \mu X.((\nu Y.\phi \wedge [-]Y) \vee (\langle - \rangle \mathit{TRUE} \wedge [-]X))$$
- *The production rule id is applied in every run* if and only if $\mathit{Entailed}(\psi, \mathit{TS}(\mathcal{G})) = \mathit{true}$, where ψ is the formula
$$\psi = \mu X.(\langle id \rangle.\mathit{TRUE} \vee (\langle - \rangle \mathit{TRUE} \wedge [-]X))$$
- *Every production rule is applied in every run* if and only if $\mathit{Entailed}(\psi, \mathit{TS}(\mathcal{G})) = \mathit{true}$, where ψ is the formula
$$\psi = \bigwedge_{id \in \mathcal{I}_{\mathcal{G}}} \mu X.(\langle id \rangle.\mathit{TRUE} \vee (\langle - \rangle \mathit{TRUE} \wedge [-]X))$$
- *Rule id is never applied* if and only if $\mathit{Entailed}(\psi_1^{id}, \mathit{TS}(\mathcal{G})) = \mathit{true}$, where ψ_1^{id} is the formula
$$\psi_1^{id} = \mu X.([-].\mathit{FALSE} \vee ([id]\mathit{FALSE} \wedge [-]X))$$
- *Rule id is applied at most once in every run* if and only if $\mathit{Entailed}(\psi_2^{id}, \mathit{TS}(\mathcal{G})) = \mathit{true}$, where ψ_2^{id} is the formula
$$\psi_2^{id} = \mu Y.((\psi_1^{id}) \vee (([id](\psi_1^{id}) \wedge \bigwedge_{id' \in \mathcal{I}_G - \{id\}} [id']Y))$$
and ψ_1^{id} is the formula defined in the previous point.

- *Every rule is applied at most once in every run* if and only if $\mathit{Entailed}(\psi, \mathit{TS}(\mathcal{G})) = \mathit{true}$, where ψ is the formula
$$\psi = \bigwedge_{id \in \mathcal{I}_G} (\psi_2^{id})$$
and ψ_2^{id} is the formula defined in the previous point.

5.1.6 Reasoning over GOPSs

In this section we study reasoning over GOPSs. In particular, we focus on the reasoning task of answering verification queries over GOPSs.

We start from the following, trivial, undecidability result.

Theorem 5.1.7. *If either the function ASK is undecidable or the function TELL is undecidable,*[4] *then answering verification queries over GOPSs is undecidable.*

As a consequence of the above result, we have that, for instance, we must carefully choose the ontology query language that must be coupled with a DL ontology language: in fact, all the typical relational database query languages are actually undecidable over DL ontologies, and even fragments of such languages are undecidable for many DL ontologies (see e.g. [92]). Indeed, the language of unions of conjunctive queries (a subset of FOL queries) is one of the most expressive languages which is decidable over (almost all) DL ontologies (see e.g. [75]). Analogous considerations hold in principle for updates over DL ontologies, even though research in this field is at an earlier stage than research in query answering, and a significant classification of decidable and undecidable update languages and update semantics for DLs is not available yet.

We now turn our attention to *decidable* classes of GOPS. Our aim is to provide sufficient conditions for the decidability of answering verification queries over GOPSs.

Our analysis starts from the fact that, according to Theorem 5.1.5, answering verification queries over a GOPS is decidable if the transition system $\mathit{TS}(\mathcal{G})$ of $\mathcal{G}$ can be built in a finite amount of time. This is not the case, of course, when $\mathit{TS}(\mathcal{G})$ is infinite: however, it might still be possible to construct in a finite amount of time a GOPS graph GG that is bisimilar to $\mathit{TS}(\mathcal{G})$. This implies decidability of our reasoning task, since, due to Theorem 5.1.6, we can evaluate verification queries using GG instead of $\mathit{TS}(\mathcal{G})$.

In the following, we thus look for sufficient conditions (on the specification of $\mathcal{G}$) for the construction of a GOPS graph that is bisimilar to $\mathit{TS}(\mathcal{G})$.

[4]More precisely, when we say that the function ASK is undecidable, we mean that the problem of establishing whether $\mathit{ASK}(\phi, \mathcal{O}) \neq \emptyset$ for a Boolean ontology query ϕ is undecidable; also, when we say that TELL is undecidable we mean that the following problem is undecidable: given two ontologies $\mathcal{O}, \mathcal{O}'$ and an ontology update α, establish whether $\mathcal{O}' = \mathit{TELL}(\mathcal{O}, \alpha)$.

We start with some auxiliary definitions.

Let $\mathcal{GO}$ be the set of all GOPS, let $\mathcal{GP}$ be the set of GOPS paths, let $\mathcal{OO}$ be the set of all ontologies, and let $\mathcal{D}$ be an arbitrary set. A *finite transformation* of a conflict resolution function Φ is a pair of functions $\langle \tau_\pi, \tau_\Phi \rangle$ where $\tau_\pi : \mathcal{GP} \rightarrow \mathcal{D}$, $\tau_\Phi : \mathcal{GO} \times \mathcal{D} \rightarrow \mathcal{OO}$, such that, for every GOPS $\mathcal{G}$: (i) for every path $\pi \in TS(\mathcal{G})$, $\Phi(\mathcal{G}, \pi) = \tau_\Phi(\mathcal{G}, \tau_\pi(\pi))$; (ii) the set $\{d \mid d = \tau_\pi(\pi)$ and π is a path of $TS(\mathcal{G})\}$ is finite.

The idea behind a finite transformation of a conflict resolution function is that it is possible to formulate the conflict resolution function without using all the information on the previous history of the system (represented by the whole GOPS path ending in the current state), but using only an approximation of such information, such that the number of possible instances of such an approximation which is relevant for a given transition system is finite. Notice that the number of possible GOPS paths is infinite, and that in general a conflict resolution function may admit no finite transformations.

This idea of finite transformation is crucial to prove a general result on the finite representability of the transition system of a GOPS, and hence a general decidability result on answering verification queries over GOPSs.

Given a GOPS $\mathcal{G}$ and a finite transformation $T = \langle \tau_\pi, \tau_\Phi \rangle$ for Φ, the T*-core* of $\mathcal{G}$, denoted by *T-core*$(\mathcal{G})$, is the GOPS graph obtained from $TS(\mathcal{G})$ by collapsing every pair of states S, S' such S and S' are locally bisimilar and $\tau_\pi(\pi_S) = \tau_\pi(\pi_{S'})$, where π_S and $\pi_{S'}$ are the paths of $TS(\mathcal{G})$ ending in S and S', respectively.

Lemma 5.1.8. *Given a GOPS $\mathcal{G}$ and a finite transformation $T = \langle \tau_\pi, \tau_\Phi \rangle$ for Φ, the T-core of $TS(\mathcal{G})$ is a finite GOPS graph.*

The following lemma states that a T-core of $\mathcal{G}$ constitutes a correct representation of $TS(\mathcal{G})$ with respect to the verification query language $\mathcal{V}(\mathcal{QL})$.

Lemma 5.1.9. *For every GOPS $\mathcal{G}$ and finite transformation T for Φ, $TS(\mathcal{G})$ and T-core$(\mathcal{G})$ are bisimilar.*

Proof (sketch). From the definition of finite transformation, it follows that the outcoming edges of two states s, s' such that $\tau_\pi(\pi_s) = \tau_\pi(\pi_{s'})$ are the same, and since s and s' are locally bisimilar, every pair of corresponding successor states of s and s' are locally bisimilar as well. $\Box$

Theorem 5.1.10. *Suppose $\langle \tau_\pi, \tau_\Phi \rangle$ is a finite transformation for Φ such that both τ_π and τ_Φ are decidable. Then, answering verification queries over a GOPS $\mathcal{G}$ is decidable.*

Proof (sketch). First, we prove that the hypotheses and Lemma 5.1.8 imply that the T-core of $\mathcal{G}$ can be computed in a finite amount of time. Then, by Lemma 5.1.9 and since the evaluation of a verification query over a finite GOPS graph is decidable, the thesis follows. $\Box$

We now prove that the conflict resolution function corresponding to the `rif:forwardChaining` strategy (without the definition of a tie-break rule) admits a finite transformation, under the condition that the number of different states of the ontology is finite.

Given an ontology $\mathcal{O}$, we define the update-closure of $\mathcal{O}$ as the set $\mathbf{C}$ inductively defined as follows: (i) $\mathcal{O} \in \mathbf{C}$; (ii) if $\mathcal{O}' \in \mathbf{C}$ and the formula ϕ is an ontology update using the constants occurring in $\mathcal{O}$, then $\mathit{TELL}(\mathcal{O}', \phi) \in \mathbf{C}$.

We say that *updates have a finite evolution* if, for every ontology $\mathcal{O}$, the update-closure of $\mathcal{O}$ contains a finite number of locally-bisimilar equivalence classes.

Theorem 5.1.11. *Suppose Φ corresponds to the `rif:forwardChaining` conflict resolution strategy, and suppose that ASK is decidable, TELL is decidable, and updates have a finite evolution. Then, answering verification queries over GOPSs is decidable.*

Proof (sketch). The definition of the conflict resolution strategy `rif:forwardChaining` implies that the corresponding conflict resolution function Φ actually uses a small amount of the information on the previous history of the system. This property, together with the hypothesis, allows for defining a function τ_π and a function τ_Φ which satisfy the hypotheses of Theorem 5.1.10. Consequently, the thesis follows. $\square$

We conclude by considering the case when the ontology is a relational database without integrity constraints. More precisely, we call DB-GOPSs the GOPSs defined based on the following assumptions:

- the ontology language is the language of ground atoms: that is, ontologies correspond to databases (sets of facts);
- the ontology update language consists of assert and retract actions of single ground atoms;
- the ontology query language consists of domain-independent FOL queries (i.e., a subclass of SQL queries);
- the ASK function evaluates queries according to the standard semantics of domain-independent FOL queries in relational databases (i.e., a database is considered as an interpretation over which the FOL queries are evaluated according to the standard FOL semantics);
- the TELL function corresponds to the syntactic additions and deletions of facts in the database.

Now, it is immediate to verify that, in DB-GOPSs, updates have a finite evolution. Therefore, as a corollary of Theorem 5.1.11, we get the following property.

Corollary 5.1.12. *Answer verification queries over DB-GOPSs under the `rif:forwardChaining` conflict resolution strategy is decidable.*

5.1.7 The EQL-Lite ontology query language

In this section we briefly recall the ontology query languages *EQL* and *EQL-Lite(UCQ)*, originally presented in [20]. Such languages are based on the addition of an epistemic operator to the first-order logic query language. The semantics of *EQL* queries and *EQL-Lite(UCQ)* queries are defined with respect to a DL ontology.

5.1.7.1 The ontology query language *EQL*

We interpret DL ontologies on interpretations sharing the *same infinite countable domain* Δ, and we assume that our language includes an infinitely countable set of disjoint constants corresponding to elements of Δ, also known as *standard names* [71]. This allows us to blur the distinction between such constants (which are syntactic objects) and the elements of Δ that they denote (which are semantic objects).

As a query language, we make use of a variant of the well-known first-order modal logic of knowledge/belief [70, 87, 71, 55], here called *EQL*. The language *EQL* is a first-order modal language with equality and with a single modal operator $\mathbf{K}$, constructed from concepts (i.e., unary predicates) and roles (i.e., binary predicates) and the constants introduced above (i.e., the standard names corresponding to Δ). In *EQL*, the modal operator is used to formalize the epistemic state of the DL ontology, according to the minimal knowledge semantics (see later). Informally, the formula $\mathbf{K}\phi$ should be read as "ϕ is known to hold (by the ontology)".

In the following, we use c to denote a constant, $\vec{c}$ to denote a tuple of constants, x to denote a variable, $\vec{x}$ to denote a tuple of variables, ϕ, ψ to denote arbitrary formulas, and $\psi[x|c]$ to denote a formula where each x is replaced by c.

A *world* is a FOL interpretation over Δ. An *epistemic interpretation* is a pair E, w, where E is a (possibly infinite) set of worlds, and w is a world in E. We inductively define when a sentence (i.e., a closed formula) ϕ *is true in an interpretation* E, w (or, is true in w and E), written $E, w \models \phi$, as follows:

$$\begin{array}{lll}
E,w \models c_1 = c_2 & \text{iff} & c_1 = c_2 \\
E,w \models P(\vec{c}) & \text{iff} & w \models P(\vec{c}) \\
E,w \models \phi_1 \wedge \phi_2 & \text{iff} & E,w \models \phi_1 \text{ and } E,w \models \phi_2 \\
E,w \models \neg\phi & \text{iff} & E,w \not\models \phi \\
E,w \models \exists x.\psi & \text{iff} & E,w \models \psi[x|c] \text{ for some constant } c \\
E,w \models \mathbf{K}\psi & \text{iff} & E,w' \models \psi \text{ for every } w' \in E
\end{array}$$

Therefore, an epistemic interpretation corresponds to a Kripke structure satisying the axioms of modal logic S5.

Formulas without occurrences of $\mathbf{K}$ are said to be *objective*, since they talk about what is true. Observe that, to check whether $E, w \models \phi$, where ϕ is an objective formula, we have to look at w but not at E: we only need the FOL interpretation w. All assertions in the

DL ontology are indeed objective sentences. Instead, formulas where each occurrence of predicates and of the equality is in the scope of the $\mathbf{K}$ operator are said to be *subjective*, since they talk about what is known to be true. Also, observe that, for a subjective sentence ϕ, in order to establish whether $E, w \models \phi$ we do not have to look at w but only at E. We use such formulas to query what the ontology knows. In other words, through subjective sentences we do not query information about the world represented by the ontology; instead, we query the epistemic state of the ontology itself. Obviously, there are formulas that are neither objective nor subjective. For example $\exists x.P(x)$ is an objective sentence, $\mathbf{K}(\exists x.P(x) \wedge \neg\mathbf{K}P(x))$ is a subjective sentence, while $\exists x.P(x) \wedge \neg\mathbf{K}P(x)$ is neither objective nor subjective.

Among the various epistemic interpretations, only the ones that represent a *minimal epistemic state* of the DL ontology, i.e., the state in which the ontology has minimal knowledge, are considered. Formally: let $\mathcal{O}$ be a DL ontology (TBox and ABox), and let $Mod(\mathcal{O})$ be the set of all FOL-interpretations that are models of $\mathcal{O}$. Then a *$\mathcal{O}$-EQL-interpretation* is an epistemic interpretation E, w for which $E = Mod(\mathcal{O})$.

A sentence ϕ is *EQL-logically implied* by $\mathcal{O}$, written $\mathcal{O} \models_{EQL} \phi$, if for every $\mathcal{O}$-EQL-interpretation E, w we have $E, w \models \phi$. Observe that for objective formulas such a definition becomes the standard one, namely $w \models \phi$ for all $w \in Mod(\mathcal{O})$, denoted by $\mathcal{O} \models \phi$.

We are now ready to define *EQL*-queries: An *EQL-query* is simply an *EQL*-formula, possibly an open one.

Let ϕ be an *EQL*-query with free variables $\vec{x}$, where the arity of $\vec{x}$ is $n \geqslant 0$, and is called the arity of ϕ. We denote such a query by $\phi[\vec{x}]$. We will use the notation $\phi[\vec{c}]$ to denote $\phi[\vec{x}|\vec{c}]$ (i.e., the formula obtained from ϕ by substituting each free occurrence of the variable x_i in $\vec{x}$ with the constant c_i in $\vec{c}$, where obviously $\vec{x}$ and $\vec{c}$ must have the same arity). Since we are dealing with all the models of the ontology, as usual, query answering should return those tuples of constants that make the query true in every model of the ontology: the so-called certain answers. Formally, the *certain answers* to a query $\phi(\vec{x})$ over an ontology $\mathcal{O}$ are the set

$$ans_{EQL}(\phi, \mathcal{O}) = \{\vec{c} \in \Delta \times \cdots \times \Delta \mid \mathcal{O} \models_{EQL} \phi[\vec{c}]\}$$

Example 5.1.13. *Consider the DL ontology $\mathcal{O}$ constituted by the following TBox $\mathcal{T}$ and ABox $\mathcal{A}$:*

$$\begin{array}{rcl} \mathcal{T} & = & \{ \ \mathsf{Male} \sqsubseteq \neg\mathsf{Female} \ \} \\ \mathcal{A} & = & \{ \ \mathsf{Female(mary)}, \mathsf{Female(ann)}, \mathsf{Female(jane)}, \\ & & \ \ \mathsf{Male(bob)}, \mathsf{Male(john)}, \mathsf{Male(paul)}, \\ & & \ \ \mathsf{PARENT(bob, mary)}, \mathsf{PARENT(bob, ann)}, \\ & & \ \ \mathsf{PARENT(john, paul)}, \mathsf{PARENT(mary, jane)} \ \} \end{array}$$

Suppose we want to know the set of males that do not have female children. This corresponds to the following FOL query ϕ_1:

$$\phi_1[x] \ = \ \mathsf{Male}(x) \wedge \neg\exists y.\mathsf{PARENT}(x, y) \wedge \mathsf{Female}(y)$$

It is easy to verify that the set of certain answers to ϕ_1 over $\mathcal{O}$ is empty. In particular, neither john *nor* paul *are certain answers to the above query, since (due to the open-world semantics of DLs) there are models of $\mathcal{O}$ in which the interpretation of* PARENT *contains pairs of elements of the form* (john, x) *or* (paul, x) *and the interpretation of* Female *contains the element x.*

Suppose now that we want to know who are the known *males that are not* known *to be parents of a female. This can be expressed by the following EQL-query ϕ_2:*

$$\phi_2[x] \;=\; \mathbf{K}\mathsf{Male}(x) \wedge \neg\mathbf{K}(\exists y.\mathsf{PARENT}(x,y) \wedge \mathsf{Female}(y))$$

It is immediate to verify that the certain answers to ϕ_2 over $\mathcal{O}$ are john *and* paul*, since they are the only known males that are not in the answer to the query* $\exists y.\mathsf{PARENT}(x,y) \wedge \mathsf{Female}(y)$.

Suppose now that we want to know who are the single children according to what is known, i.e., the known children who have no known sibling. This can be expressed by the following EQL-query ϕ_3:

$$\begin{aligned}\phi_3[x] \;=\; & \exists y.(\mathbf{K}\mathsf{PARENT}(y,x)) \wedge \\ & \qquad \forall z.(\mathbf{K}\mathsf{PARENT}(y,z)) \rightarrow z = x\end{aligned}$$

It is immediate to verify that the certain answers to ϕ_3 over $\mathcal{O}$ are paul *and* jane. ■

Notice that, in an *EQL*-query, we can apply a form of closed world reasoning: for example, in query ϕ_2 above, the evaluation of $\neg\mathbf{K}(\exists y.\mathsf{PARENT}(x,y) \wedge \mathsf{Female}(y))$ corresponds to the evaluation of $\neg\exists y.\mathsf{PARENT}(x,y) \wedge \mathsf{Female}(y)$ under the closed world assumption.

5.1.7.2 The ontology query language *EQL-Lite(UCQ)*

We introduce now the query language *EQL-Lite(UCQ)*. Such a language is a particularly well-behaved fragment of *EQL*, and is based on the language of unions of conjunctive queries (UCQ) over a DL signature.

Informally, *EQL-Lite(UCQ)* is the FOL query language with equality whose atoms are epistemic formulas of the form $\mathbf{K}q$, where q is a UCQ. Formally, an *EQL-Lite(UCQ)* query is a possibly open *EQL*-formula built according to the following syntax:

$$\phi \;::=\; \mathbf{K}q \mid x_1 = x_2 \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \exists x.\phi$$

where q is a UCQ. We call *epistemic atoms* the formulas $\mathbf{K}q$ occurring in an *EQL-Lite(UCQ)* query.

Observe that in *EQL-Lite(UCQ)* we do not allow the $\mathbf{K}$ operator to occur outside of the epistemic atoms $\mathbf{K}q$. Indeed, allowing for occurrences of the $\mathbf{K}$ outside such atoms does not actually increase the expressive power of *EQL-Lite(UCQ)*, as shown in [20].

EQL-Lite(UCQ) queries enjoy a very interesting computational property: one can decouple the reasoning needed for answering the epistemic atoms from the reasoning needed for answering the whole query. Formally, let $\mathcal{O}$ be a DL ontology, and $\phi[\vec{x}]$ be an *EQL-Lite(UCQ)* query over $\mathcal{O}$, whose epistemic atoms are $\mathbf{K}q_1, \ldots, \mathbf{K}q_m$. We denote by $q_{\scriptscriptstyle FOL}[\vec{x}]$ the FOL query obtained from ϕ by replacing each epistemic atom $\mathbf{K}q_i$ by a new predicate $R_{\mathbf{K}q_i}$ whose arity is the number of free variables in q_i. Also, we denote by $\mathcal{I}_{\phi,\mathcal{O}}$ the FOL interpretation for the predicates $R_{\mathbf{K}q_i}$ defined as follows: (*i*) the interpretation domain is $\Delta^{\mathcal{I}_{\phi,\mathcal{O}}} = \Delta$; (*ii*) the extension of the predicates $R_{\mathbf{K}q_i}$ is $R_{\mathbf{K}q_i}^{\mathcal{I}_{\phi,\mathcal{O}}} = ans_{EQL}(q_i, \mathcal{O})$. Finally, we denote by $Eval(q_{\scriptscriptstyle FOL}[\vec{x}], \mathcal{I}_{\phi,\mathcal{O}}) = \{\vec{c} \in \Delta \times \cdots \times \Delta \mid \mathcal{I}_{\phi,\mathcal{O}} \models q_{\scriptscriptstyle FOL}[\vec{c}]\}$ the result of evaluating $q_{\scriptscriptstyle FOL}$ over $\mathcal{I}_{\phi,\mathcal{O}}$.

The following property has been stated in [20]. If $\mathcal{O}$ is a DL ontology, ϕ an *EQL-Lite(UCQ)* query over $\mathcal{O}$, and $q_{\scriptscriptstyle FOL}$ and $\mathcal{I}_{q,\mathcal{O}}$ the FOL query and the FOL interpretation defined above, then $ans_{EQL}(\phi, \mathcal{O}) = Eval(q_{\scriptscriptstyle FOL}, \mathcal{I}_{\phi,\mathcal{O}})$. This property tells us that, in order to compute the certain answers of an *EQL-Lite(UCQ)* query ϕ, we can compute the certain answers of queries q_i of the embedded query language $\mathcal{Q}$ occurring in the epistemic atoms of ϕ, and then evaluate the query ϕ as a FOL query, where we consider such certain answers as the extensions of the epistemic atoms.

The theorem above suggests a procedure to compute certain answers in *EQL-Lite(UCQ)*. However, for such a procedure to be effective, we need to address two issues: (*i*) the extension of the predicates $R_{\mathbf{K}q_i}$ in the FOL interpretation $\mathcal{I}_{\phi,\mathcal{O}}$ needs to be finite, otherwise $\mathcal{I}_{\phi,\mathcal{O}}$ would be infinite and the evaluation of $q_{\scriptscriptstyle FOL}$ impossible in practice; (*ii*) since Δ itself is infinite, the evaluation of $q_{\scriptscriptstyle FOL}$ must not directly deal with Δ.

We start by looking at the second issue first. Such an issue has a long tradition in relational databases where indeed one allows only for FOL queries that are "domain independent" [2]. In our context, a FOL query ϕ is *domain independent* if for each pair of FOL interpretations $\mathcal{I}_1$ and $\mathcal{I}_2$, respectively over domains $\Delta^{\mathcal{I}_1} \subseteq \Delta$ and $\Delta^{\mathcal{I}_2} \subseteq \Delta$, for which $R_{\mathbf{K}q_i}^{\mathcal{I}_1} = R_{\mathbf{K}q_i}^{\mathcal{I}_2}$ for all atomic relations $R_{\mathbf{K}q_i}$, we have that $Eval(\phi, \mathcal{I}_1) = Eval(\phi, \mathcal{I}_2)$. We say that an *EQL-Lite(UCQ)* query ϕ is *domain independent* if its corresponding query $q_{\scriptscriptstyle FOL}$ is so. Domain independent FOL queries correspond to relational algebra queries (i.e., SQL queries) and several syntactic sufficient conditions have been devised in order to guarantee domain independence, see e.g., [2]. Such syntactic conditions can be directly translated into syntactic conditions on *EQL-Lite(UCQ)* queries.

As for the other issue, let $\mathcal{O}$ be a DL ontology and let q be a UCQ. We say that q is *$\mathcal{O}$-range-restricted* if $ans_{EQL}(q, \mathcal{O})$ is a *finite* set of tuples. By extension, an *EQL-Lite(UCQ)* query is $\mathcal{O}$-range-restricted if each of its epistemic atoms involves a $\mathcal{O}$-range-restricted query. It can be shown that, if $\mathcal{O}$ is a DL ontology and q is a $\mathcal{O}$-range-restricted query, then $ans_{EQL}(q, \mathcal{O}) \subseteq adom(\mathcal{O}) \times \cdots \times adom(\mathcal{O})$, where $adom(\mathcal{O})$ is the set of all constants explicitly appearing in $\mathcal{O}$.

Example 5.1.14. *Queries ϕ_2 and ϕ_3 in Example 5.1.13 are EQL-Lite(UCQ) queries. It is easy to verify that both such queries are domain independent, and that both are $\mathcal{O}$-range-restricted for the ontology $\mathcal{O}$ given in Example 5.1.13. In fact ϕ_2 and ϕ_3 are $\mathcal{O}$-range-*

restricted for ontologies $\mathcal{O}$ expressed (in practice) in any standard DL (indeed, the set $ans_{EQL}(\mathsf{PARENT}(x, y), \mathcal{O})$ *may never be infinite).*

5.1.7.3 Answering *EQL-Lite(UCQ)* queries over *DL-Lite*$_{\mathcal{A}}$ ontologies

Finally, we recall the complexity of answering *EQL-Lite(UCQ)* queries over ontologies of a specific DL, *DL-Lite*$_{\mathcal{A}}$, introduced in Section 5.1.2.

It is known [20] that the problem of answering *EQL-Lite(UCQ)* queries over *DL-Lite*$_{\mathcal{A}}$ ontologies is decidable and is even tractable with respect to *data complexity*, i.e., with respect to the size of the ABox of the ontology.

First, it is easy to see that for every *DL-Lite*$_{\mathcal{A}}$ ontology $\mathcal{O}$, and for every UCQ q, q is $\mathcal{O}$-range-restricted. So, range-restrictedness is guaranteed in *DL-Lite*$_{\mathcal{A}}$ for *EQL-Lite(UCQ)* queries.

The following property, shown in [20], states that query answering in *DL-Lite*$_{\mathcal{A}}$ is tractable with respect to data complexity.

Theorem 5.1.15. *Answering domain independent EQL-Lite(UCQ) queries in DL-Lite is in PTIME (in particular, in* AC^0*) with respect to data complexity.*

5.1.8 Instance-level DL ontology update semantics

In this section we recall the semantics for the evolution of DL knowledge bases presented in [68].

This approach has three key aspects. First, it concerns the so-called *instance-level updates* to DL ontologies [30]. In fact, this approach only considers update operations corresponding to the assertion or retraction of sets of ABox assertions, i.e., only the extensional component (ABox) of the DL ontology is updated, while the intensional component (TBox) is unchanged. Second, it proposes a semantics for ontology updates which is closed with respect to the DL language. In other words, given a DL Ł and an Ł ontology $\mathcal{O}$, the result of any update operation can always be expressed in terms of a DL ontology $\mathcal{O}'$ in the language Ł. Finally, the proposed semantics has nice computational properties.

In the rest of this section, Ł is a DL, and every ABox $\mathcal{A}$ is a set of ground atoms (also called atomic ABox assertions). Moreover, we assume that the initial ontology $\mathcal{O} = \langle T, \mathcal{A} \rangle$ is a satisfiable Ł ontology (according to the semantics of Ł). In other words, we do not consider the evolution of unsatisfiable ontologies. In addition, F is a finite set of atomic ABox assertions in Ł. Finally, we denote by $\mathsf{cl}_{\mathcal{T}}(\mathcal{A})$ the $\mathcal{T}$-closure of $\mathcal{A}$, i.e., the set of atomic ABox assertions that are entailed by $\langle \mathcal{T}, \mathcal{A} \rangle$ according to the semantics of Ł.

5.1.8.1 WIDTIO-based semantics for instance-level ontology update

First, we specify when a set of ABox assertions "realizes" the insertion or deletion of a set of ABox assertions with respect to $\mathcal{O}$.

Let $\mathcal{A}'$ be an ABox. Then, $\mathcal{A}'$ *accomplishes the insertion of* F *into* $\langle\mathcal{T},\mathcal{A}\rangle$ if $\mathcal{A}'$ is $\mathcal{T}$-consistent, and $\langle\mathcal{T},\mathcal{A}'\rangle \models F$ (i.e., $F \subseteq \mathsf{cl}_{\mathcal{T}}(\mathcal{A}')$). Similarly, $\mathcal{A}'$ *accomplishes the deletion of* F *from* $\langle\mathcal{T},\mathcal{A}\rangle$ if $\mathcal{A}'$ is $\mathcal{T}$-consistent, and $\langle\mathcal{T},\mathcal{A}'\rangle \not\models F$ (i.e., $F \not\subseteq \mathsf{cl}_{\mathcal{T}}(\mathcal{A}')$).

Obviously, we are interested in ontologies which accomplish the evolution of an ontology with a *minimal change*. In order to formalize the notion of *minimal change*, we first need to provide some definitions.

Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two ABoxes. Then, we say that $\mathcal{A}_1$ has fewer deletions than $\mathcal{A}_2$ with respect to $\langle\mathcal{T},\mathcal{A}\rangle$ if $\mathsf{cl}_{\mathcal{T}}(\mathcal{A}) - \mathsf{cl}_{\mathcal{T}}(\mathcal{A}_1) \subset \mathsf{cl}_{\mathcal{T}}(\mathcal{A}) - \mathsf{cl}_{\mathcal{T}}(\mathcal{A}_2)$. Similarly, we say that $\mathcal{A}_1$ and $\mathcal{A}_2$ have the same deletions with respect to $\langle\mathcal{T},\mathcal{A}\rangle$ if $\mathsf{cl}_{\mathcal{T}}(\mathcal{A}) - \mathsf{cl}_{\mathcal{T}}(\mathcal{A}_1) = \mathsf{cl}_{\mathcal{T}}(\mathcal{A}) - \mathsf{cl}_{\mathcal{T}}(\mathcal{A}_2)$. Finally, we say that $\mathcal{A}_1$ has fewer insertions than $\mathcal{A}_2$ with respect to $\langle\mathcal{T},\mathcal{A}\rangle$ if $\mathsf{cl}_{\mathcal{T}}(\mathcal{A}_1) - \mathsf{cl}_{\mathcal{T}}(\mathcal{A}) \subset \mathsf{cl}_{\mathcal{T}}(\mathcal{A}_2) - \mathsf{cl}_{\mathcal{T}}(\mathcal{A})$.

Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two ABoxes. Then, $\mathcal{A}_1$ has *fewer changes* than $\mathcal{A}_2$ with respect to $\langle\mathcal{T},\mathcal{A}\rangle$ if $\mathcal{A}_1$ has fewer deletions than $\mathcal{A}_2$ with respect to $\langle\mathcal{T},\mathcal{A}\rangle$, or $\mathcal{A}_1$ and $\mathcal{A}_2$ have the same deletions with respect to $\langle\mathcal{T},\mathcal{A}\rangle$, and $\mathcal{A}_!$ has fewer insertions than $\mathcal{A}_2$ with respect to $\langle\mathcal{T},\mathcal{A}\rangle$.

Now that we have defined the relation of *fewer changes* between two ontologies w.r.t. another one, we can define the notion of an ontology which accomplishes the insertion (resp. deletion) of a set of facts into (resp. from) another ontology minimally.

Let $\mathcal{A}'$ be an ABox. Then $\mathcal{A}'$ accomplishes the insertion (deletion) of F into (from) $\langle\mathcal{T},\mathcal{A}\rangle$ *minimally* if $\mathcal{A}'$ accomplishes the insertion (deletion) of F into (from) $\langle\mathcal{T},\mathcal{A}\rangle$, and there is no $\mathcal{A}''$ that accomplishes the insertion (deletion) of F into (from) $\langle\mathcal{T},\mathcal{A}\rangle$, and has fewer changes than $\mathcal{A}'$ with respect to $\langle\mathcal{T},\mathcal{A}\rangle$.

With these notions in place, we can now define the evolution operator.

Definition 5.1.16. *Let* $\mathcal{U} = \{\mathcal{A}_1,\ldots,\mathcal{A}_n\}$ *be the set of all ABoxes accomplishing the insertion (deletion) of* F *into (from)* $\langle\mathcal{T},\mathcal{A}\rangle$ *minimally, and let* $\mathcal{A}'$ *be an ABox. Then,* $\langle\mathcal{T},\mathcal{A}'\rangle$ *is the result of changing* $\langle\mathcal{T},\mathcal{A}\rangle$ *with the insertion (deletion) of* F *if* (1) $\mathcal{U}$ *is empty, and* $\langle\mathcal{T},\mathit{cl}_{\mathcal{T}}(\mathcal{A}')\rangle = \langle\mathcal{T},\mathit{cl}_{\mathcal{T}}(\mathcal{A})\rangle$*, or* (2) $\mathcal{U}$ *is nonempty, and* $\langle\mathcal{T},\mathit{cl}_{\mathcal{T}}(\mathcal{A}')\rangle = \langle\mathcal{T},\bigcap_{1\leqslant i\leqslant n}\mathit{cl}_{\mathcal{T}}(\mathcal{A}_i)\rangle$.

It is immediate to verify that, up to logical equivalence, the result of changing $\langle\mathcal{T},\mathcal{A}\rangle$ with the insertion or the deletion of F is unique. The result of changing $\mathcal{O} = \langle\mathcal{T},\mathcal{A}\rangle$ with the insertion of F according to the above semantics will be denoted by *LS-assert*$(\mathcal{O},F)$. Moreover, the result of changing $\mathcal{O} = \langle\mathcal{T},\mathcal{A}\rangle$ with the deletion of F according to the above semantics will be denoted by *LS-retract*$(\mathcal{O},F)$.

Notice that, in the case where F is $\mathcal{T}$-inconsistent, the result of changing $\langle\mathcal{T},\mathcal{A}\rangle$ with both the insertion and the deletion of F is logically equivalent to $\langle\mathcal{T},\mathcal{A}\rangle$ itself.

It can be shown [38, 68] that: (i) an ABox $\mathcal{A}'$ accomplishes the deletion of F from $\langle\mathcal{T},\mathcal{A}\rangle$ minimally if and only if $\mathsf{cl}_{\mathcal{T}}(\mathcal{A}')$ is a maximal $\mathcal{T}$-closed subset of $\mathsf{cl}_{\mathcal{T}}(\mathcal{A})$ such that $F \not\subseteq \mathsf{cl}_{\mathcal{T}}(\mathcal{A}')$; (ii) $\mathcal{A}'$ accomplishes the insertion of F from $\langle\mathcal{T},\mathcal{A}\rangle$ minimally if and only if $\mathsf{cl}_{\mathcal{T}}(\mathcal{A}') = \mathcal{A}'' \cup \mathsf{cl}_{\mathcal{T}}(F)$, where $\mathcal{A}''$ is a maximal $\mathcal{T}$-closed subset of $\mathsf{cl}_{\mathcal{T}}(\mathcal{A})$ such that $\mathcal{A}'' \cup F$ is $\mathcal{T}$-consistent.

From now on, we call $\mathcal{UL}$*-Lite* the instance-level ontology update language which consists of every formula of the form *Assert*(F) or *Retract*(F), where F is a finite set of atomic ABox assertions (i.e., a conjunction of ground atoms).

5.1.8.2 Computing ontology updates in *DL-Lite*$_{\mathcal{A}}$

In the following we consider the computational properties of the above update operators in the DL *DL-Lite*$_{\mathcal{A}}$ [84].

As already mentioned in Section 5.1.2, a *DL-Lite*$_{\mathcal{A}}$ ABox is a set of ground atoms. Therefore, in *DL-Lite*$_{\mathcal{A}}$ the above ontology updates described above correspond to the assertion and retraction of ABoxes.

The following theorem immediately derives from the results presented in [68].

Theorem 5.1.17. *Let $\mathcal{O} = \langle\mathcal{T},\mathcal{A}\rangle$ be a DL-Lite$_{\mathcal{A}}$ ontology, and let $\mathcal{A}'$ be a DL-Lite$_{\mathcal{A}}$ ABox. Then:*

1. *LS-assert$(\mathcal{O},\mathcal{A}')$ can be computed in PTIME.*
2. *LS-retract$(\mathcal{O},\mathcal{A}')$ can be computed in PTIME.*

Also, [68] presents polynomial algorithms for computing the results of the above form of ontology updates in *DL-Lite*$_{\mathcal{A}}$.

We point out that the above theorem is a very interesting result, since it states that computing updates according to the semantics above illustrated can be done in polynomial time, although such a notion of updates does not boil down to purely syntactic insertion/deletion operations.

5.1.9 Lite-GOPS: a tractable combination of DL ontologies and PSs

After introducing the ontology query language *EQL-Lite(UCQ)* and the ontology update language $\mathcal{UL}$*-Lite*, we are ready to present a subclass of GOPSs which enjoys nice computational properties.

5.1.9.1 Lite-GOPS

Lite-GOPSs are a particular form of GOPSs based on *DL-Lite*$_{\mathcal{A}}$ ontologies, *EQL-Lite(UCQ)* ontology queries, $\mathcal{UL}$*-Lite* ontology updates under the semantics defined in Section 5.1.8.

A *Lite-GOPS* is a GOPS as in Definition 5.1.2 under the following assumptions:

- the ontology language $\mathcal{OL}$ is *DL-Lite*$_{\mathcal{A}}$;
- the ontology query language $\mathcal{QL}$ is *EQL-Lite(UCQ)*;
- the ontology update language $\mathcal{UL}$ is $\mathcal{UL}$*-Lite*;
- the function ASK which provides the semantics for ontology queries corresponds to the semantics for *EQL* queries described in Section 5.1.7. More precisely, for every *DL-Lite*$_{\mathcal{A}}$ ontology $\mathcal{O}$, and for every *EQL-Lite(UCQ)* query ϕ, $ASK(\phi, \mathcal{O}) = ans_{EQL}(\phi, \mathcal{O})$;
- the function $TELL$ which provides the semantics for ontology updates corresponds to the semantics for ontology update described in Section 5.1.8. More precisely:
  - for every ontology update α of the form *Assert*(Γ), $TELL(\mathcal{O}, \alpha) =$ *LS-assert*$(\mathcal{O}, \alpha)$;
  - for every ontology update α of the form *Retract*(Γ), $TELL(\mathcal{O}, \alpha) =$ *LS-retract*$(\mathcal{O}, \alpha)$;
- the conflict resolution strategy Φ admits a finite transformation.

5.1.9.2 Reasoning in Lite-GOPSs

In this section we show that answering verification queries over Lite-GOPSs is decidable.

We also identify some subclasses of queries of the verification query language which can be answered in polynomial time when evaluated over Lite-GOPSs.

First, it is immediate to verify that the chosen semantics of ontology updates and the adoption of the *DL-Lite*$_{\mathcal{A}}$ language imply the following property.

Lemma 5.1.18. *Updates have a finite evolution in Lite-GOPSs.*

We are now ready to prove the following decidability result.

Theorem 5.1.19. *Answering verification queries over Lite-GOPSs is decidable.*

Proof. The proof immediately follows from Lemma 5.1.18, Theorem 5.1.15, Theorem 5.1.17 and Theorem 5.1.11. □

Then, we study the complexity of reasoning over Lite-GOPSs under the `rif:forwardChaining` conflict resolution strategy.

We call *data complexity* of answering verification queries over a Lite-GOPS $\mathcal{G} = \langle \mathcal{O}_{in}, \mathcal{P} \rangle$, where $\mathcal{O}_{in}$ is a *DL-Lite*$_{\mathcal{A}}$ ontology $\langle \mathcal{T}, \mathcal{A} \rangle$, the complexity of the above problem measured with respect to the size of the ABox $\mathcal{A}$.

Theorem 5.1.20. *Answering verification queries over Lite-GOPSs under the* `rif:forwardChaining` *conflict resolution strategy is in EXPTIME with respect to data complexity.*

Proof (sketch). The thesis follows from Theorem 5.1.5, Theorem 5.1.15 and Theorem 5.1.17. □

We now prove a tractability result for reasoning over Lite-GOPSs. In particular, we prove that reasoning over GOPSs is tractable for verification queries without fixpoints,

In the following, we say that a verification query is *simple* if it does not contain fixpoint operators.

Theorem 5.1.21. *Answering simple verification queries over Lite-GOPSs is in PTIME with respect to data complexity.*

Proof (sketch). The key property is that it is sufficient to build a small (polynomial) part of the transition system. In particular, if k is the size of the simple verification query (actually k should represent the maximum nesting level of the modalities), then it is sufficient to build the paths of the transition system that start at the initial state and have a length less than or equal to k. Now, the number of outcoming edges from a state for the same production rule p is polynomial with respect to data complexity, since there is only a polynomial number of ground substitutions for the free variables of p, consequently the number of outcoming edges from a state is polynomial. This in turn implies that the number of the above paths of length $\leqslant k$ is polynomial with respect to data complexity (since k does not depend on the size of the ABox), therefore this portion of the transition system can be built in polynomial time. Thus, from Theorem 5.1.5 it follows that the evaluation of the query over such a polynomial model is polynomial with respect to data complexity, which implies the thesis. □

5.1.10 Summary

In this chapter we have presented generalized ontology-based production systems (GOPSs), which constitute a very general framework for the combination of ontologies and production rules. The GOPS approach is based on a functional specification of ontologies, which views ontologies as knowledge bases which can be accessed through a query service and an update service. In this way, the semantics of the execution of production rules over ontologies is straightforward, and fully relies on the semantics of queries and updates over ontologies.

Then, we have defined an expressive language for formalizing verification tasks over GOPS specifications, and have shown that typical static analysis tasks can easily be expressed through such a language.

Moreover, we have studied the computational properties of reasoning in the framework of

GOPSs, providing very general sufficient conditions for undecidability and decidability of reasoning.

Finally, we have analyzed a specific combination of ontologies and production rules, called Lite-GOPSs. We have established decidability and complexity results for reasoning in such a class of GOPS. In particular, we have shown that Lite-GOPSs enjoy very nice computational properties, thus they constitute a very good trade-off between the complexity of reasoning and the expressive power of the ontology component of the GOPSs.

This approach can be further extended in several directions. First of all, in a way analogous to Lite-GOPSs, other notable specific combinations of ontologies and production rules can be defined and studied within the GOPS framework. Then, it would be very interesting to further extend the GOPS framework, by adding other form of production rules beyond those considered in the RIF-PRD specification (as the one taken into account in Chapter 4). Finally, it would also be interesting to extend this approach to more powerful verification languages. For instance, it would be very easy to extend $\mathcal{V}(\mathcal{QL})$ to allow for the presence of free variables.

Chapter 6

Discussion Regarding Convergence

While the previous chapters of this deliverable presented results on optimizing, refining, and improving combinations of logical rules and ontologies and combinations of production rules and ontologies, in this chapter we discuss some issues related to the theoretical convergence of all three knowledge representation paradigms: logical rules, production rules, and ontologies.

We start with the observation that all combinations explored in the project had ontologies as a component. There has also been extensive work devoted to the issue of rewriting ontologies to logical rules: this makes it possible to see logical programming formalisms as tight combinations between themselves and such rule-based/translatable ontologies. As such, when talking about combinations of all three paradigms, the onus lies on combining logical rules with production rules. There has already been some work in this area, in particular the integration via $\mathcal{TR}^{PAD}$ described in chapter 4. Since it is a trivial task to accommodate logical rules within $\mathcal{TR}^{PAD}$, the Integration via Transaction Logic with Partially Defined Actions is de facto an approach which integrates production rules, logical rules, and ontologies. As concerns practical integration, ontoprise worked on a first version of converging the logical programming approach ObjectLogic with ideas coming from the area of production rules. The work is related to the requirements resulting from the AUDI Business Orchestration Use Case and is described within the ONTORULE deliverable D4.3 [93].

Here we present some alternative ideas about how one can bring together production rules and logical rules, and ultimately integrate them with ontologies.

ACTHEX [8] is an extension of HEX programs [34] with *action atoms*: unlike dl-atoms in dl-programs [33], which only send and receive inputs to/from ontologies, action atoms are associated to functions capable of actually changing the state of external environments. Such atoms can appear (only) in heads of rules and as such they can be part of answer sets. An action atom is of the form $\#g\,[Y_1,\dots Y_n]\,\{o,r\}\,[w:l]$ where $Y_1,\dots,Y_n$ is a list of terms (called input list), and $\#g$ is an action predicate name. We assume that $\#g$ has fixed length $in(\#g)=n$ for its input list. $o\in\{b,c,c_p\}$ is called the *action option*.

Depending on the value of o, the action atom is called *brave, cautious, preferred cautious*, respectively. r, w and l range over positive integers and variables, and are called *action precedence*, *action weight* and *action level* respectively.

Action atoms are executed according to *execution schedules*. Every answer set is associated with one or more *execution schedules*, where an execution schedule is an ordered list containing all action atoms in that particular answer set. The order of execution within a schedule depends on the actions *precedence* attribute. Action atoms allow to specify whether they have to be executed *bravely*, *cautiously* or *preferred cautiously*, respectively, meaning that the atom can get executed if it appears in at least one, all, or all *best cost* answer sets.

A potential usage of ACTHEX programs is for updating knowledge bases by asserting or removing facts to these. [8] describes how the abstract action constructs $assert(kb, f)$ and $retract(kb, f)$, whose effect is to add or remove a statement f from a given knowledge base kb, can be grounded to ACTHEX programs, by introducing two action predicates $\#assert_k$ and $\#retract_k$, for $k > 0$. An atom $\#assert_k[kb, a_1, \ldots, a_k]\{o, p\}$, (resp. $\#retract_k[kb, a_1, \ldots, a_k]\{o, p\}$) adds to (resp. removes from) the knowledge base kb the assertion $a_1| \ldots |a_k$, for $a_i|a_j$, being the string concatenation of a_i and a_j. Further on, in order to be able to simulate a sequence of updates, the action atom $\#execute[kb]\{o, p\}$ is introduced, where kb is an ACTHEX program, and o, and p are as in the definition of action atoms: the execution of such an atom consists in evaluating kb and executing the resulting execution schedule.

Notice that when an ACTHEX program is such that every open answer set contains one and only one $execute$ action atom, as described above, this atom has the highest priority (i.e. it is the last in every execution schedule) and the kb which is supposed to be updated is the extensional part of the original ACTHEX program, the effect is to have a stateful update formalism. Intuitively, this is exactly the device needed for simulating the operational semantics of a production rule system. Rules with update action atoms in the heads act like production rules, the extensional part of an ACTHEX program can be seen as a working memory, while the ‘execute’ atom triggers a new firing step in the execution. The logic programming nature of ACTHEX, together with the presence of action precedences and priorities, makes it easy to encode declaratively different conflict resolution strategies.

Let’s consider a scenario where: (1) all production rules have equal (or no) priorities, (2) they always add or remove a single atom to the working memory, and (3) the conflict resolution strategy is to randomly pick for execution one of the firable rules. Assume production rules are labeled with natural numbers from 1 to n. The scenario could be encoded in ACTHEX as follows (P is the name of the resulted ACTHEX program):

- Selection of one and only one production rule to fire:
  - for every production rule (i) : *if* p *then remove/add* q, add the following rule to P:
    $c_i \leftarrow p$ (c_i holds in an answer set iff rule i is firable);

  - for every production rule (i) : *if p then remove/add q*, add the following rule to P:

    $d_i \leftarrow c_i, not\ d_1, \ldots, not\ d_{i-1}, not\ d_{i+1}, \ldots, not\ d_n$ (d_i holds in an answer set iff rule i is firable, and is selected for execution; at most one d_i holds in every answer set);

- specification of the production rules using action atoms: for every production rule (i) : *if p then remove/add q*, add the following rule to the ACTHEX program:

  $retract/assert[WM, “p”, “.”]\{b, 1\} \leftarrow d_i$ (if production rule i is firable and selected for execution, i.e. d_i holds, the action atom $retract/assert[WM, “p”, “.”]$ will be part of the answer set with precedence 1 and mode of execution $brave$; note that, due to the switch implemented by d_i-s, at most one such atom will be part of any answer set);

- check whether the execution terminates (no rule is firable) and if not, trigger the next execution step:

  $terminates \leftarrow not\ c_1, \ldots not\ c_n$ (the production rule is firable)

  $\#execute[P]\{b, 2\} \leftarrow not\ terminates$ (if $terminates$ is false, i.e. some production rules fires, $execute[P]$ will be part of the answer set with precedence 2, i.e. it will be executed after any update action $retract/assert[WM, “p”, “.”]$).

The encoding described above is quite rudimentary, as a production rule could fire infinitely often, in case its prerequisite is met and the update caused by the rule does not change its firability. Assume we want the system to have the following property: a rule fires only if it causes change to the WM (the atom which it adds/removes is not/is part of the WM). In order to access the WM (which we assume it is stored in a different file than P), we can use an external atom. External atoms are a feature of HEX programs [34]: the generic form of such an atom is $\&g[Y_1, \ldots, Y_n](X_1, \ldots, X_m)$, where $Y_1, \ldots, Y_n$ and $X_1, \ldots, X_m$ are two lists of terms (called input and output lists, respectively), and $\&g \in \mathcal{G}$ is an external predicate name. We assume that $\&g$ has fixed lengths $in(\&g) = n$ and $out(\&g) = m$ for input and output lists, respectively. One can use such an atom to query the working memory: $\&check(WM, p(X_1, \ldots, X_q))$ is evaluated as true if the fact ‘$p(X_1, \ldots, X_q).'$ is part of WM, and false, otherwise. The rules encoding firability of production rules have to be changed:

- for every production rule (i) : *if p then remove q*:

  $c_i \leftarrow p, \&check(WM, q)$ (rule i is firable if its prerequisite p is fulfilled and q is part of the WM).

- for every production rule (i) : *if p then add q*:

  $c_i \leftarrow p, \&check(WM, q)$ (rule i is firable if its prerequisite p is fulfilled and q is not part of the WM).

Note that external atoms can serve as a loosely-coupled interface to DL ontologies. Such atoms are generalizations of dl-atoms: they can be used for querying arbitrary DL ontologies. Thus, the formalism offers a smooth integration of all three worlds: production rules, logical rules, and ontologies (where the ontologies do not have to be Datalog rewritable). An implementation of ACTHEX programs has been realized and is available[1] as an extension to the dlvhex system[2]. As concerns the update actions, the `KBModaddon` library implements a generalization of such update action atoms. The downside with using ACTHEX for encoding production rule systems is that there is no guarantee that such programs which have recursive evaluation calls using the action predicate $execute$ terminate. As such, it is not possible, in the general case, to check static properties of production rule systems like termination, or whether a certain fact holds in some execution/all executions, etc. Finding appropriate restrictions for termination is subject of future work.

Alternatively, one can use a logic programming based formalism which can simulate a forward notion of time and for which decidability of different reasoning tasks together with algorithms which implement such tasks are well-investigated. Such a formalism is $\mathbb{FDNC}$ [35, 98].

$\mathbb{FDNC}$ is a decidable extension of Answer Set Programming with function symbols. Decidability is achieved, similarly to FoLPs, by ensuring that the language has the forest model property: the fragment allows only for unary and binary predicates and the syntax of the rules is restricted. However, the shape of $\mathbb{FDNC}$ programs is such that they are amenable to bottom-up reasoning (as opposed to FoLPs for which we have top-down tableau-like reasoning procedure). As such, FoLPs can simulate forward branching time and they are a good device encoding for action languages, planning problems, etc. In order to offer a richer representation device for such problems, the syntax of $\mathbb{FDNC}$ has been relaxed to allow for higher-arity predicates: some syntactical restrictions regarding the usage of variables have been imposed in order to maintain decidability. The new fragment is called *higher-arity* $\mathbb{FDNC}$. Standard reasoning tasks like deciding consistency of a program, cautious/brave entailments of atoms, etc. are proved to be decidable and algorithms for checking each of these tasks are provided.

In [35] a translation of the action language $\mathcal{K}$ to $\mathbb{FDNC}$ is provided. Using a similar translation, one could encode hypothetical runs of production rule systems. An advantage of this approach is that the existent reasoning support for $\mathbb{FDNC}$ allows for checking static properties of such encoded systems. However, due to the pure declarative nature of $\mathbb{FDNC}$, one cannot materialize the results of execution runs. Also, the lack of external atoms makes it impossible to have a loosely-coupled interaction with external information sources like ontologies. However, as previously mentioned, Datalog-rewritable ontologies can always be tightly-coupled with such a translation, due to their inherent property: rewritability. It remains to be explored how and if $\mathbb{FDNC}$ could be extended with external atoms which would allow sending inputs, querying, and modifying external sources.

[1] `http://www.kr.tuwien.ac.at/research/systems/dlvhex/actionplugin.html`

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

Another logic programming based formalism which can simulate a forward time dimension is Splittable Temporal Logic Programs (STLP) [12] which are a fragment of Temporal Equilibrium Logic, an extension of ASP with modal temporal operators. An algorithm for reasoning with such programs is provided in [12]: temporal equilibrium models are captured by an LTL formula by using two well-known techniques for reasoning with ASPs: splitting and loop formulas. The algorithm has been implemented using an LTL model checker, SPOT [13]. Similar considerations as for $\mathbb{FDNC}$ apply for encodings of production rule systems using STLP.

Chapter 7

Conclusions

In this deliverable we took further the work on execution performed during Year 2 of the project and improved it by :

1. exploring a more expressive language, in the area of Datalog rewritable Description Logics: Horn-$\mathcal{SHIQ}$ is a fragment of the DL $\mathcal{SHIQ}$ with features identified to be desirable in the analysis of case studies like existentials on the right-hand side of the axioms and inverse properties. A new algorithm for reasoning with the fragment has been devised and a prototype reasoner, KAOS, which can answer conjunctive queries over Horn-$\mathcal{SHIQ}$ ontologies, has been developed;
2. devising an optimized (as compared to last year) and worst-case optimal algorithm for reasoning with Forest Logic Programs: the algorithm runs in exponential time, one exponential level than the previous algorithm and it shows that FoLPs are ExpTime-complete.
3. introducing a richer combination of Production Rule Systems and ontologies which allow for FOR-loops as typical in commercial systems, and whose semantics deals with inconsistency. The combination is given a model theoretic semantics by embedding it into $\mathcal{TR}^{PAD}$ for rule-based ontologies. Finally, $\mathcal{TR}^{PAD}$ was extended with default negation under the well-founded semantics.
4. defining a general framework for combining Production Rule Systems with ontologies: Generalized Ontology-based Production Systems (GOPSs), together with a powerful query verification language for verifying static properties of such systems. A specific type of GOPSs, *Lite*-GOPSs are identified, which use $DL-Lite_{\mathcal{A}}$ as an ontology language, $EQL-Lite(UCQ)$ as a verification language, and an ontology update language $\mathcal{UL}-Lite$, and for which verification tasks are decidable, and in particular cases, even tractable.

We also explored how logical rules, production rules, and ontologies can be brought together by considering ways of embedding production rule systems in logic programming based formalisms: a practical approach, ACTHEX, that extends ASP with external atoms and executable action atoms (atoms which change the state of external environments) has been considered, as well as a pure declarative decidable extension of ASP with function symbols: $\mathbb{FDNC}$. Both approaches offer some desirable features for such an embedding, but not in totality: more work is needed in the case of each formalism in order to be able to use them for achieving full convergence of the three paradigms.

Acknowledgement

The authors would like to thank the reviewers of this document, Dr. Eva Kiss and Prof. François Levy, for their helpful comments and suggestions.

Bibliography

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison Wesley, 1995.

[3] Guray Alsac and Chitta Baral. Characterizing production systems using logic programming and situation calculus, 2001. Available from `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.72.2728&rep=rep1&type=pdf`.

[4] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, second edition, 2007.

[5] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003.

[6] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook*. Cambridge University Press, 2003.

[7] C. Baral and J. Lobo. Characterizing production systems using logic programming and situation calculus. `http://www.public.asu.edu/~cbaral/papers/char-prod-systems.ps`, 1995.

[8] Selen Basol, Ozan Erdem, Michael Fink, and Giovambattista Ianni. Hex programs with action atoms. In *Technical Communications of the 26th International Conference on Logic Programming*, volume 7 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24–33, 2010.

[9] A.J. Bonner and M. Kifer. Transaction logic programming (or a logic of declarative and procedural knowledge). Technical Report CSRI-323, University of Toronto, November 1995. `http://www.cs.sunysb.edu/~kifer/TechReports/transaction-logic.pdf .`

[10] A.J. Bonner and M. Kifer. A logic for programming database transactions. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, chapter 5, pages 117–166. Kluwer Academic Publishers, March 1998.

[11] Julien Bradfield and Colin Stirling. Modal mu-calculi. In *Handbook of Modal Logic*, volume 3, pages 721–756. Elsevier, 2007.

[12] P. Cabalar. Loop formulas for splitable temporal logic programs. In James Delgrande and Wolfgang Faber, editors, *Proc. of the 11th Intl. Conf. on Logic Programming and Nonmonotonic Reasoning (LPNMR11)*, volume 6645 of *Lecture Notes in Computer Science*, pages 80–92. Springer Berlin / Heidelberg, 2011.

[13] Pedro Cabalar and Martn Diguez. Stelp a tool for temporal answer set programming. In James Delgrande and Wolfgang Faber, editors, *Proc. of the 11th Intl. Conf. on Logic Programming and Nonmonotonic Reasoning*, volume 6645 of *Lecture Notes in Computer Science*, pages 370–375. Springer Berlin / Heidelberg, 2011.

[14] Andrea Calì, Georg Gottlob, and Michael Kifer. Taming the Infinite Chase: Query Answering under Expressive Relational Constraints. In Franz Baader, Carsten Lutz, and Boris Motik, editors, *Description Logics'08*, volume 353. CEUR-WS.org, 2008.

[15] Andrea Calì, Georg Gottlob, and Thomas Lukasiewicz. A General Datalog-Based Framework for Tractable Query Answering over Ontologies. In *In Proc. PODS-2009*, pages 77–86. ACM Press, 2009.

[16] Andrea Calì, Georg Gottlob, and Thomas Lukasiewicz. Datalog : A Unified Approach to Ontologies and Integrity Constraints. volume 9, pages 14–30, 2009.

[17] Andrea Calì, Georg Gottlob, and Andreas Pieris. Advanced Processing for Ontological Queries. *Proceedings of the VLDB Endowment*, 3(1):554–565, 2010.

[18] Andrea Calì, Georg Gottlob, and Andreas Pieris. Query Answering under Non-Guarded Rules in Datalog. In *Proceedings of the 4th International Conference on Web Reasoning and Rule Systems (RR 2010)*, pages 1–17, 2010.

[19] 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.

[20] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. EQL-Lite: Effective first-order query processing in description logics. In *Proc. 20th International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 274–279, 2007.

[21] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Tractable reasoning and efficient query answering in description logics: The *DL-Lite* family. *Journal of Automated Reasoning*, 39(3):385–429, 2007.

[22] Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. On the decidability of query containment under constraints. In *Proceedings of the Seventeenth ACM SIGACT SIGMOD SIGART Symposium on Principles of Database Systems (PODS'98)*, pages 149–158, 1998.

[23] Diego Calvanese, Evgeny Kharlamov, Werner Nutt, and Dmitriy Zheleznyakov. Evolution of *DL-Lite* knowledge bases. In *Proceedings of the Ninth International Semantic Web Conference (ISWC 2010)*, volume 6496 of *LNCS*, pages 112–128. Springer, 2010.

[24] Piero Cangialosi, Giuseppe De Giacomo, Riccardo De Masellis, and Riccardo Rosati. Conjunctive artifact-centric services. In *Proceedings of the Eighth International Joint Conference on Service Oriented Computing (ICSOC 2010)*, volume 6470 of *LNCS*, pages 318–333. Springer, 2010.

[25] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. The MIT Press, Cambridge, MA, USA, 1999.

[26] Carlos Viegas Damásio, José Júlio Alferes, and João Leite. Declarative semantics for the rule interchange format production rule dialect. In *Proceedings of the 9th international semantic web conference on The semantic web - Volume Part I*, ISWC'10, pages 798–813, Berlin, Heidelberg, 2010. Springer-Verlag.

[27] Carlos Viegas Damasio, Jose Julio Alferes, and Joao Leite. Declarative semantics for the rule interchange format production rule dialect. In *Proceedings of the Ninth International Semantic Web Conference (ISWC 2010)*, pages 798–813, 2010.

[28] Jos de Bruijn and Martín Rezk. A logic based approach to the static analysis of production systems. In *RR*, pages 254–268, 2009.

[29] Jos de Bruijn and Martin Rezk. A logic based approach to the static analysis of production systems. In *Proceedings of the Third International Conference on Web Reasoning and Rule Systems (RR 2009)*, pages 254–268, 2009.

[30] Giuseppe De Giacomo, Maurizio Lenzerini, Antonella Poggi, and Riccardo Rosati. On instance-level update and erasure in description logic ontologies. *Journal of Logic and Computation, Special Issue on Ontology Dynamics*, 19(5):745–770, 2009.

[31] C. de Sainte Marie, G. Hallmark, and A. Paschke (editors). RIF production rule dialect. W3C Recommendation, `http://www.w3.org/TR/rif-prd/`, 2010.

[32] Thomas Eiter, Giovambattista Ianni, Thomas Krennwallner, and Roman Schindlauer. Exploiting conjunctive queries in description logic programs. *Ann. Math. Artif. Intell.*, 53(1-4):115–152, 2008.

[33] Thomas Eiter, Giovambattista Ianni, Thomas Lukasiewicz, Roman Schindlauer, and Hans Tompits. Combining answer set programming with description logics for the semantic web. *Artif. Intell.*, 172:1495–1539, August 2008.

[34] Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In *Proceedings IJCAI-2005*, pages 90–96. Professional Book Center, 2005.

[35] Thomas Eiter and Mantas Šimkus. $\mathbb{FDNC}$: Decidable non-monotonic disjunctive logic programs with function symbols. *Transactions on Computational Logic (TOCL)*, 11(2), April 2010. Article 14 (50 + 16 pages).

[36] E. Allen Emerson. Model checking and the mu-calculus. In *Descriptive Complexity and Finite Models*, pages 185–214, 1996.

[37] 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.

[38] Ronald Fagin, Jeffrey D. Ullman, and Moshe Y. Vardi. On the semantics of updates in databases. In *Proceedings of the Second ACM SIGACT SIGMOD Symposium on Principles of Database Systems (PODS'83)*, pages 352–365, 1983.

[39] C. Feier and S. Heymans. Hybrid Reasoning with Forest Logic Programs. In *Proc. of 6th European Semantic Web Conference*, volume 5554, pages 338–352. Springer, 2009.

[40] Cristina Feier, Hassan Aït-Kaci, Jürgen Angele, Jos de Bruijn, Hugues Citeau, Thomas Eiter, Adil El Ghali, Volha Kerhet, Eva Kiss, Roman Korf, Thomas Krekeler, Thomas Krennwallner, Stijn Heymans, Alessandro Mosca (FUB), Martín Rezk, and Guohui Xiao. D3.3 - Complexity and optimization of combinations of rules and ontologies. Technical report, ONTORULE IST-2009-231875 Project, 2010.

[41] Cristina Feier and Stijn Heymans. An optimization for reasoning with forest logic programs. In *Proc. of Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP)*. CoRR, 2010.

[42] Cristina Feier and Stijn Heymans. Reasoning with forest logic programs and f-hybrid knowledge bases. *TPLP*, 2011. to appear.

[43] Paul Fodor and Michael Kifer. Transaction logic with defaults and argumentation theories. In *ICLP (Technical Communications)*, pages 162–174, 2011.

[44] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and M. Schneider. Potassco: The Potsdam answer set solving collection. *AI Communications*, 24(2):105–124, 2011.

[45] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Proc. 5th International Conference on Logic Programming (ICLP'88)*, pages 1070–1080. MIT Press, 1988.

[46] Birte Glimm. Using sparql with rdfs and owl entailment. In *Reasoning Web 2011*, volume 6848 of *Lecture Notes in Computer Science*, pages 137–201, 2011.

[47] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. Lubm: A benchmark for owl knowledge base systems. *J. Web Sem.*, 3(2-3):158–182, 2005.

[48] 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.

[49] S. Heymans, D. Van Nieuwenborgh, and D. Vermeir. Open Answer Set Programming for the Semantic Web. *J. of Applied Logic*, 5(1):144–169, 2007.

[50] 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.

[51] 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.

[52] Stijn Heymans, Thomas Eiter, and Guohui Xiao. Tractable reasoning with dl-programs over datalog-rewritable description logics. In *European Conference on Artificial Intelligence*, pages 35–40, 2010.

[53] Matthew Horridge and Sean Bechhofer. The owl api: A java api for owl ontologies. *Semantic Web*, 2(1):11–21, 2011.

[54] Ian Horrocks. Ontologies and the semantic web. *Commun. ACM*, 51:58–67, December 2008.

[55] George E. Hughes and M. J. Cresswell. *A Companion to Modal Logic*. Methuen, London (United Kingdom), 1984.

[56] Ullrich Hustadt, Boris Motik, and Ulrike Sattler. Reasoning in description logics by a reduction to disjunctive datalog. *J. Autom. Reason.*, 39:351–384, October 2007.

[57] U. Sattler I. Horrocks and S. Tobies. Practical Reasoning for Expressive Description Logics. In *Proc. 6th Int. Conf. on Logic for Programming and Automated Reasoning (LPAR'99)*, volume LNAI 1705, pages 161–180. Springer Verlag, 1999.

[58] IBM JRules. http://www.ibm.com/software/integration/business-rule-management/jrules-family/.

[59] Yevgeny Kazakov. Consequence-driven reasoning for horn SHIQ ontologies. In Craig Boutilier, editor, *IJCAI*, pages 2040–2045, 2009.

[60] C. Maria Keet, Ronell Alberts, Aurona Gerber, and Gibson Chimamiwa. Enhancing web portals with ontology-based data access: The case study of south africa's accessibility portal for people with disabilities. In Catherine Dolbear, Alan Ruttenberg, and Ulrike Sattler, editors, *OWLED*, volume 432 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[61] Roman Kontchakov, Carsten Lutz, David Toman, Frank Wolter, and Michael Zakharyaschev. The combined approach to ontology-based data access. In Toby Walsh, editor, *IJCAI*, pages 2656–2661. IJCAI/AAAI, 2011.

[62] Robert Kowalski and Fariba Sadri. Integrating logic programming and production systems in abductive logic programming agents. In *Proceedings of the 3rd International Conference on Web Reasoning and Rule Systems*, RR '09, pages 1–23. Springer-Verlag, 2009.

[63] Markus Krötzsch, Sebastian Rudolph, and Pascal Hitzler. On the complexity of horn description logics. In Bernardo Cuenca Grau, Pascal Hitzler, Conor Shankey, and Evan Wallace, editors, *OWLED*, volume 216 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2006.

[64] Markus Krötzsch, Sebastian Rudolph, and Pascal Hitzler. Complexity boundaries for horn description logics. In *AAAI*, pages 452–457, 2007.

[65] Markus Krötzsch, Sebastian Rudolph, and Pascal Hitzler. Complexity of horn description logics. Technical report, 2007.

[66] Markus Krötzsch, Sebastian Rudolph, and Peter H. Schmitt. On the semantic relationship between datalog and description logics. In *Proceedings of the Fourth international conference on Web reasoning and rule systems*, RR'10, pages 88–102, Berlin, Heidelberg, 2010. Springer-Verlag.

[67] Georg Lausen, Bertram Ludaescher, and Wolfgang May. On active deductive databases: The statelog approach. In *In Transactions and Change in Logic Databases*, pages 69–106. Springer-Verlag, 1998.

[68] Maurizio Lenzerini and Domenico Fabio Savo. On the evolution of the instance level of *DL-Lite* knowledge bases. In *Proceedings of the Twentyfourth International Workshop on Description Logic (DL 2011)*, volume 745 of *CEUR Electronic Workshop Proceedings,* `http://ceur-ws.org/`, 2011.

[69] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The dlv system for knowledge representation and reasoning. *ACM Trans. Comput. Log.*, 7(3):499–562, 2006.

[70] Hector J. Levesque. Foundations of a functional approach to knowledge representation. *Artificial Intelligence*, 23:155–212, 1984.

[71] Hector J. Levesque and Gerhard Lakemeyer. *The Logic of Knowledge Bases*. MIT Press, 2001.

[72] Alon Y. Levy and Marie-Christine Rousset. Combining Horn rules and description logics in CARIN. *Artificial Intelligence*, 104(1–2):165–209, 1998.

[73] H. Liu, C. Lutz, M. Milicic, and F. Wolter. Updating description logic ABoxes. In *Proc. 10th International Conference on Principles of Knowledge Representation and Reasoning (KR'06)*, pages 46–56, 2006.

[74] Hongkai Liu, Carsten Lutz, Maja Milicic, and Frank Wolter. Foundations of instance level updates in expressive description logics. *Artif. Intell.*, 175(18):2170–2197, 2011.

[75] Carsten Lutz. Inverse roles make conjunctive queries hard. In *Proceedings of the Twentieth International Workshop on Description Logic (DL 2007)*, volume 250 of *CEUR Electronic Workshop Proceedings,* `http://ceur-ws.org/`, pages 100–111, 2007.

[76] Carsten Lutz. The complexity of conjunctive query answering in expressive description logics. In *Proceedings of the Fourth International Joint Conference on Automated Reasoning (IJCAR 2008)*, volume 5195 of *LNAI*, pages 179–193. Springer, 2008.

[77] Li Ma, Yang Yang, Zhaoming Qiu, Guo Tong Xie, Yue Pan, and Shengping Liu. Towards a complete owl ontology benchmark. In York Sure and John Domingue, editors, *ESWC*, volume 4011 of *Lecture Notes in Computer Science*, pages 125–139. Springer, 2006.

[78] Boris Motik. Description Logics and Disjunctive Datalog—More Than just a Fleeting Resemblance? In Holger Schlingloff, editor, *Proc. of the 4th Workshop on Methods for Modalities (M4M-4)*, volume 194 of *Informatik-Berichte der Humboldt-Universität zu Berlin*, pages 246–265, Berlin, Germany, December 1–2 2005.

[79] Boris Motik and Riccardo Rosati. Reconciling Description Logics and Rules. *Journal of the ACM*, 57(5):1–62, 2010.

[80] Magdalena Ortiz. *Query Answering in Expressive Description Logics: Techniques and Complexity Results*. PhD thesis, Vienna University of Technology, Austria, 2010.

[81] Magdalena Ortiz, Sebastian Rudolph, and Mantas Simkus. Worst-case optimal reasoning for the horn-dl fragments of owl 1 and 2. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Twelfth International Conference (KR-10)*, pages 269–279. AAAI Press, May 2010.

[82] D. Pearce and G. Wagner. Logic programming with strong negation. In *Proceedings of the international workshop on Extensions of logic programming*, pages 311–326, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[83] Héctor Pérez-Urbina, Boris Motik, and Ian Horrocks. A comparison of query rewriting techniques for DL-Lite. In *In Proc. of the Int. Workshop on Description Logics (DL2009)*, Oxford, UK., July 2009.

[84] Antonella Poggi, Domenico Lembo, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. Linking data to ontologies. *Journal on Data Semantics*, X:133–173, 2008.

[85] Louiqa Raschid. A semantics for a class of stratified production system programs. *J. Log. Program.*, 21(1):31–57, 1994.

[86] Alan L. Rector and Jeremy Rogers. Ontological and practical issues in using a description logic to represent medical concept systems: Experience from galen. In *Reasoning Web*, volume 4126 of *Lecture Notes in Computer Science*, pages 197–231, 2006.

[87] Raymond Reiter. What should a database know? *Journal of Logic Programming*, 14:127–153, 1990.

[88] Martín Rezk and Michael Kifer. Formalizing production systems with rule-based ontolgies, 2011. Available from `http://www.inf.unibz.it/~mrezk/techreportTRPS.pdf`.

[89] Martín Rezk and Michael Kifer. Reasoning with actions in transaction logic. In *The Fifth International Conference on Web Reasoning and Rule Systems RR'11*, pages 201–216. Springer, 2011.

[90] Martín Rezk and Werner Nutt. Combining production systems and ontologies. In *The Fifth International Conference on Web Reasoning and Rule Systems RR'11*, pages 287–293. Springer, 2011.

[91] Martn Rezk and Werner Nutt. Combining production systems and ontologies. In *Proceedings of the Fifth International Conference on Web Reasoning and Rule Systems (RR 2011)*, pages 287–293, 2011.

[92] Riccardo Rosati. The limits of querying ontologies. In *Proceedings of the Eleventh International Conference on Database Theory (ICDT 2007)*, volume 4353 of *LNCS*, pages 164–178. Springer, 2007.

[93] Peter Rosina. D4.3 - Audi R&D Business Orchestration System WP 4 'Case Study: CAx Integration'. Technical report, ONTORULE IST-2009-231875 Project, 2010.

[94] Evren Sirin and Bijan Parsia. Sparql-dl: Sparql query for owl-dl. In *Proc. OWLED 2007*, 2007.

[95] Trung Kien Tran. Query answering in the description logic Horn-SHIQ. Master's thesis, Vienna Univerty of Technology, 2011.

[96] A. Van Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

[97] 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.

[98] M. Šimkus and T. Eiter. $\mathbb{FDNC}$: Decidable Non-monotonic Disjunctive Logic Programs with Function Symbols. In *Proc. 14th Int. Conf. on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2007)*, 2007.

[99] W3C, 2010. Available from `http://www.w3.org/TR/rif-prd/`.

[100] Marianne Winslett. *Updating logical databases*. Cambridge University Press, New York, NY, USA, 1990.

[101] Guohui Xiao and Thomas Eiter. Inline evaluation of hybrid knowledge bases - phd description. In Sebastian Rudolph and Claudio Gutierrez, editors, *RR*, volume 6902 of *Lecture Notes in Computer Science*, pages 300–305. Springer, 2011.

[102] Carlo Zaniolo. A unified semantics for active and deductive databases. In *Workshop on Rules In Database Systems (RIDS-93)*, pages 271–287. Springer Verlag, 1993.

Glossary

ABox Synonym of Assertion Box, 3, 84, 113

Answer Set Programming Answer Set Programming (ASP) is a form of declarative programming oriented towards difficult search problems. It is based on the stable model (answer set) semantics of logic programming. In ASP, search problems are reduced to computing stable models, and answer set solvers are used to perform search., 35

Closed-World-Assumption In the Closed World Assumption one considers all facts about a universe of discourse to be known., 129

Conjunctive query The conjunctive queries are the fragment of first-order logic given by the set of formulae that can be constructed from atomic formulae using conjunction and existential quantification, but not using disjunction, negation, or universal quantification., 1

Datalog Datalog is a query and rule language for deductive databases that syntactically is a subset of Prolog., 3

Decidability A decision problem, i.e., a yes-or-no question with a potentially infinite input domain, is decidable if there is an algorithm that computes the correct answer for every finite input within a finite amount of time., 35

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., 82, 107

DL-Programs DL-Programs is a loosely coupled approach of integration of Ontology and Rules., 3

DReW DReW (Datalog ReWriter) 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, 3

F-hybrid Knowledge Bases *f-hybrid* knowledge bases is a tightly-coupling formalism for integrating rules and ontologies that combines knowledge bases expressed in the Description Logic SHOQ with Forest Logic Programs., 35

Forest Logic Programs Forest Logic Programs (FoLPs) is a decidable subset of OASP which has the forest-model property., 35

Herbrand Universe The Herbrand universe of a first order language is the set of all ground terms. If the language has no constants, then the language is extended by adding an arbitrary new constant., 35

Open Answet Set Programming Open Answer Set Programming (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., 35

Open-World-Assumption In the Open World Assuption one considers that not all facts are known., 81

Production Rule Systems A production system (or production rule system) is a computer program typically used to provide some form of artificial intelligence, which consists primarily of a set of rules about behavior. These rules, termed productions, are a basic representation found useful in automated planning, expert systems and action selection. A production system provides the mechanism necessary to execute productions in order to achieve some goal for the system., 80, 107

RIF PRD RIF PRD is the RIF dialect intended for XML serialization of production rule languages, 80

Satisfiability checking for FoLPs Satisfiability checking in the context of reasoning with Forest Logic Programs is a reasoning task which consists in checking that a unary predicate is satisfiable, i.e. there exists an open answer set which contains a fact pertaining to that predicate., 36

Stable Model Semantics The concept of a stable model, or answer set, is used to define a declarative semantics for logic programs with negation as failure. This is one of several standard approaches to the meaning of negation in logic programming, along with program completion and the well-founded semantics. The stable model semantics is the basis of answer set programming., 35

Transaction Logic Transaction logic is an extension of predicate logic with both declarative and procedural semantics that describe state changes in logic programming over dynamic databases., 2

Transaction Logic with Partially Defined Actions Transaction Logic with Partially Defined Actions is an extension of Transaction Logic where a theory consists of serial-Horn rules, partial action definitions, and certain statements about states and actions, called *premises*., 2

Universe of discourse The universe of discourse refers to the set of all things under consideration in the context; in a FOL context, it is the set of things covered by universal quantification., 1