A Sound and Complete Algorithm for Simple Conceptual Logic Programs*

Cristina Feier and Stijn Heymans

Knowledge-Based Systems Group, Institute of Information Systems
Vienna University of Technology
Favoritenstrasse 9-11, A-1040 Vienna, Austria
{feier,heymans}@kr.tuwien.ac.at

Abstract. Open Answer Set Programming (OASP) is a knowledge representation paradigm that allows for a tight integration of Logic Programming rules and Description Logic ontologies. Although several decidable fragments of OASP exist, no reasoning procedures for such expressive fragments were identified so far. We provide an algorithm that checks satisfiability in NEXPTIME for the fragment of EXPTIME-complete *simple conceptual logic programs*.

1 Introduction

Integrating Description Logics (DLs) with rules for the Semantic Web has received considerable attention over the past years with approaches such as *Description Logic Programs* [10], *DL-safe rules* [16], $\mathcal{DL}$+*log* [17], *dl-programs* [5], and Open Answer Set Programming (OASP) [13]. OASP combines attractive features from both the DL and the Logic Programming (LP) world: an open domain semantics from the DL side allows for stating generic knowledge, without mentioning actual constants, and a rule-based syntax from the LP side supports nonmonotonic reasoning via *negation as failure*.

Decidable fragments for OASP satisfiability checking were identified as syntactically restricted programs, that are still expressive enough for integrating rule- and ontology-based knowledge, see, e.g., *Conceptual Logic Programs* [12] or *g-hybrid knowledge bases* [11]. A shortcoming of those decidable fragments of OASP is the lack of effective reasoning procedures. In this paper, we take a first step in mending this by providing a sound and complete algorithm for satisfiability checking in a particular fragment of Conceptual Logic Programs.

The major contributions of the paper can be summarized as follows:

– We identify a fragment of Conceptual Logic Programs (CoLPs), called *simple CoLPs*, that disallow for inverse predicates and inequality compared to CoLPs, but are expressive enough to simulate the DL $\mathcal{SH}$. We show that

* This work is partially supported by the Austrian Science Fund (FWF) under the projects *Distributed Open Answer Set Programming (FWF P20305)* and *Reasoning in Hybrid Knowledge Bases (FWF P20840)*.

satisfiability checking w.r.t. simple CoLPs is EXPTIME-complete (i.e., it has the same complexity as CoLPs).
- We define a nondeterministic algorithm for deciding satisfiability, inspired by tableaux-based methods from DLs, that constructs a finite representation of an open answer set. We show that this algorithm is terminating, sound, complete, and runs in NEXPTIME.

The algorithm is non-trivial from two perspectives: both the minimal model semantics of OASP, compared to the model semantics of DLs, as well as the open domain assumption, compared to the closed domain assumption of ASP, pose specific challenges in constructing a finite representation that corresponds to an open answer set. Detailed proofs and an extended example can be found in [6].

2 Preliminaries

We recall the open answer set semantics from [13]. *Constants* $a, b, c, \ldots$, *variables* $x, y, \ldots$, *terms* $s, t, \ldots$, and *atoms* $p(t_1, \ldots, t_n)$ are defined as usual. A *literal* is an atom $p(t_1, \ldots, t_n)$ or a *naf-atom* $not\ p(t_1, \ldots, t_n)$. For a set α of literals or (possibly negated) predicates, $\alpha^+ = \{l \mid l \in \alpha, l \text{ an atom or a predicate}\}$ and $\alpha^- = \{l \mid not\ l \in \alpha, l \text{ an atom or a predicate}\}$. For a set X of atoms, $not\ X = \{not\ l \mid l \in X\}$. For a set of (possibly negated) predicates α, we will often write $\alpha(x)$ for $\{a(x) \mid a \in \alpha\}$ and $\alpha(x, y)$ for $\{a(x, y) \mid a \in \alpha\}$.

A *program* is a countable set of rules $\alpha \leftarrow \beta$, where α and β are finite sets of literals. The set α is the *head* of the rule and represents a disjunction, while β is called the *body* and represents a conjunction. If $\alpha = \emptyset$, the rule is called a *constraint*. *Free rules* are rules $q(x_1, \ldots, x_n) \vee not\ q(x_1, \ldots, x_n) \leftarrow$ for variables $x_1, \ldots, x_n$; they enable a choice for the inclusion of atoms. We call a predicate q *free* in a program if there is a free rule $q(x_1, \ldots, x_n) \vee not\ q(x_1, \ldots, x_n) \leftarrow$ in the program. Atoms, literals, rules, and programs that do not contain variables are *ground*. For a rule or a program X, let $cts(X)$ be the constants in X, $vars(X)$ its variables, and $preds(X)$ its predicates with $upreds(X)$ the unary and $bpreds(X)$ the binary predicates. A *universe* U for a program 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 possible constant in U. Let $\mathcal{B}_P$ ($\mathcal{L}_P$) be the set of atoms (literals) that can be formed from a ground program P.

An *interpretation* I of a ground P is any 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 $I \not\models p(t_1, \ldots, t_n)$. For a set of ground literals X, $I \models X$ if $I \models l$ for every $l \in X$. 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 ground program P 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* [7] 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.

In the following, a program is assumed to be a finite set of rules; infinite programs only appear as byproducts of grounding a finite program 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 and a $(x_1, \ldots, x_n) \in U^n$ such that $p(x_1, \ldots, x_n) \in M$.

We introduce some notations for trees as in [19]. For an $x \in \mathbb{N}_0^*$ [1] we denote the concatenation of a number $c \in \mathbb{N}_0$ to x as $x \cdot c$, or, abbreviated, as xc. Formally, a *(finite) tree* T is a (finite) subset of $\mathbb{N}_0^*$ such that if $x \cdot c \in T$ for $x \in \mathbb{N}_0^*$ and $c \in \mathbb{N}_0$, then $x \in T$. Elements of T are called *nodes* and the empty word ε is the *root* of T. For a node $x \in T$ we call $succ_T(x) = \{x \cdot c \in T \mid c \in \mathbb{N}_0\}$, *successors* of x. The *arity* of a tree is the maximum amount of successors any node has in the tree. The set $A_T = \{(x, y) \mid x, y \in T, \exists c \in \mathbb{N}_0 : y = x \cdot c\}$ denotes the set of edges of a tree T. We define a partial order $\leq$ on a tree T such that for $x, y \in T$, $x \leq y$ iff x is a prefix of y. As usual, $x < y$ if $x \leq y$ and $y \not\leq x$. A (finite) *path* P in a tree T is a prefix-closed subset of T such that $\forall x \neq y \in P : |x| \neq |y|$. We call $path_T(x, y)$ a finite path in T with x the smallest element of the path w.r.t. the order relation $<$ and y the greatest element. The *length* of a finite path is the number of elements of the path. Infinite paths have no greatest element w.r.t. $<$. A branch B in a tree T is a maximal path (there is no path which contains it) which contains the root of T.

For programs containing only unary and binary predicates it makes sense to define a *tree model property*: for a program P containing only unary and binary predicates, if a unary predicate $p \in preds(P)$ is satisfiable w.r.t. P then p is tree satisfiable w.r.t. P. A predicate p is *tree satisfiable* w.r.t. P if there exists

- an open answer set (U, M) of P such that U is a tree of bounded arity, and
- a labeling function $t : U \to 2^{preds(P)}$ such that
  - $p \in t(\varepsilon)$ and $t(\varepsilon)$ does not contain binary predicates, and
  - $z \cdot i \in U$, $i > 0$, iff there is some $f(z, z \cdot i) \in M$, and
  - for $y \in U$, $q \in upreds(P)$, $f \in bpreds(P)$,
    - $q(y) \in M$ iff $q \in t(y)$, and
    - $f(x, y) \in M$ iff $y = x \cdot i \wedge f \in t(y)$.

We call such a (U, M) a *tree model* for p w.r.t. P.

3 Simple Conceptual Logic Programs

In [12], we defined *Conceptual Logic Programs (CoLPs)*, a syntactical fragment of logic programs for which satisfiability checking under the open answer set semantics is decidable. We restrict this fragment by disallowing the occurrence of inequalities and inverse predicates, resulting in *simple conceptual logic programs.*

[1] By $\mathbb{N}_0$ we denote the set of natural numbers excluding 0, and by $\mathbb{N}_0^*$ the set of finite sequences over $\mathbb{N}_0$.

Definition 1. *A* simple conceptual logic program (simple CoLP) *is a program with only unary and binary predicates, without constants, and such that any rule is a* free rule*, a* unary rule

$$a(x) \leftarrow \beta(x), \big(\gamma_m(x, y_m), \delta_m(y_m)\big)_{1 \leq m \leq k} \tag{1}$$

where for all m*,* $\gamma_m^+ \neq \emptyset$*, or a* binary rule

$$f(x, y) \leftarrow \beta(x), \gamma(x, y), \delta(y) \tag{2}$$

with $\gamma^+ \neq \emptyset$*.*

Intuitively, the free rules allow for a free introduction of atoms (in a first-order way) in answer sets, unary rules consist of a root atom $a(x)$ that is motivated by a syntactically tree-shaped body, and binary rules motivate a $f(x, y)$ for a x and its 'successor' y by a body that only considers atoms involving x and y.

Simple CoLPs can simulate constraints $\leftarrow \beta(x), \big(\gamma_m(x, y_m), \delta_m(y_m)\big)_{1 \leq m \leq k}$, where $\forall m : \gamma_m^+ \neq \emptyset$, i.e., constraints have a body that has the same form as a body of a unary rule. Indeed, such constraints $\leftarrow$ *body* can be replaced by simple CoLP rules of the form $constr(x) \leftarrow not\ constr(x), body$, for a new predicate $constr$.

As simple CoLPs are CoLPs and the latter have the tree model property [12], simple CoLPs have the tree model property as well.

Proposition 1. *Simple CoLPs have the tree model property.*

For CoLPs this tree model property was important to ensure that a tree automaton [19] could be constructed that accepts tree models in order to show decidability. The presented algorithm for simple CoLPs relies as well heavily on this tree model property.

As satisfiability checking of CoLPs is EXPTIME-complete [12], checking satisfiability of simple CoLPs is in EXPTIME.

In [12], it was shown that CoLPs are expressive enough to simulate satisfiability checking w.r.t to $\mathcal{SHIQ}$ knowledge bases, where $\mathcal{SHIQ}$ is the Description Logic (DL) extending $\mathcal{ALC}$ with transitive roles ($\mathcal{S}$), support for role hierarchies ($\mathcal{H}$), inverse roles ($\mathcal{I}$), and qualified number restrictions ($\mathcal{Q}$). For an overview of DLs, we refer the reader to [1].

Using a restriction of this simulation, one can show that satisfiability checking of $\mathcal{SH}$ concepts (i.e., $\mathcal{SHIQ}$ without inverse roles and quantified number restrictions) w.r.t. a $\mathcal{SH}$ TBox can be reduced to satisfiability checking of a unary predicate w.r.t. a simple CoLP. Intuitively, simple CoLPs cannot handle inverse roles (as they do not allow for inverse predicates) neither can they handle number restrictions (as they do not allow for inequality). As satisfiability checking of $\mathcal{ALC}$ concepts w.r.t. an $\mathcal{ALC}$ TBox (note that $\mathcal{ALC}$ is a fragment of $\mathcal{SH}$) is EXPTIME-complete ([1, Chapter 3]), we have EXPTIME-hardness for simple CoLPs as well.

Proposition 2. *Satisfiability checking w.r.t. simple CoLPs is* EXPTIME*-complete.*

4 An Algorithm for Simple Conceptual Logic Programs

In this section, we define a sound, complete, and terminating algorithm for satisfiability checking w.r.t. simple CoLPs.

For every non-free predicate q and a simple CoLP P, let P_q be the rules of P that have q as a head predicate. 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). The negation of $\pm p$ is $\mp p$, that is, $\mp p = \mathit{not}\ p$ if $\pm p = p$ and $\mp p = p$ if $\pm p = \mathit{not}\ p$.

For a unary rule r of the form (1), we define $\mathit{degree}(r) = |\{m \mid \gamma_m \neq \emptyset\}|$. For every non-free rule $r : \alpha \leftarrow \beta \in P$, we assume that there exists an injective function $i_r : \beta \rightarrow \{0, \ldots, |\beta|\}$ which defines a total order over the literals in β and an inverse function $l_r : \{0, \ldots, |\beta|\} \rightarrow \beta$ which returns the literal with the given index in β. For a rule r which has body variables $x, y_1, \ldots, y_k$ we introduce a function $\mathit{varset}_r : \{x, y_1, \ldots, y_k, (x, y_1), \ldots, (x, y_k)\} \rightarrow 2^{\{0,\ldots,|\beta|\}}$ which for every variable or pair of variables which appears in at least one literal in a rule returns the set of indices of the literals formed with the corresponding variable(s).

The basic data structure for our algorithm is a *completion structure*.

Definition 2 (completion structure). *A completion structure for a simple CoLP P is a tuple $\langle T, G, \textsc{ct}, \textsc{st}, \textsc{rl}, \textsc{sg}, \textsc{nj}_{\mathrm{U}}, \textsc{nj}_{\mathrm{B}}\rangle$, where T is a tree which together with the labeling functions* CT, ST, RL, SG, $\textsc{nj}_{\mathrm{U}}$, *and* $\textsc{nj}_{\mathrm{B}}$, *represents a tentative tree model and $G = \langle V, E\rangle$ is a directed graph with nodes $V \subseteq \mathcal{B}_{P_T}$ and edges $E \subseteq \mathcal{B}_{P_T} \times \mathcal{B}_{P_T}$ which is keeps track of dependencies between elements of the constructed model. The labeling functions are defined as following:*

- *The* content *function* $\textsc{ct} : T \cup A_T \rightarrow 2^{\mathit{preds}(P) \cup \mathit{not}\ (\mathit{preds}(P))}$ *maps a node of the tree to a set of (possibly negated) unary predicates and an edge of the tree to a set of (possibly negated) binary predicates such that* $\textsc{ct}(x) \subseteq \mathit{upreds}(P) \cup \mathit{not}(\mathit{upreds}(P))$ *if* $x \in T$, *and* $\textsc{ct}(x) \subseteq \mathit{bpreds}(P) \cup \mathit{not}(\mathit{bpreds}(P))$ *if* $x \in A_T$.
- *The* status *function* $\textsc{st} : \{(x, \pm q) \mid \pm q \in \textsc{ct}(x), x \in T \cup A_T\} \rightarrow \{\mathit{exp}, \mathit{unexp}\}$ *attaches to every (possibly negated) predicate which appears in the content of a node/edge x a status value which indicates whether the predicate has already been expanded in that node/edge.*
- *The* rule *function* $\textsc{rl} : \{(x, q) \mid x \in T \cup A_T, q \in \textsc{ct}(x)\} \rightarrow P$ *associates with every node/edge x of T and every positive predicate $q \in \textsc{ct}(x)$ a rule which has q as a head predicate:* $\textsc{rl}(x, q) \in P_q$.
- *The* segment *function* $\textsc{sg} : \{(x, q, r) \mid x \in T, \mathit{not}\ q \in \textsc{ct}(x), r \in P_q\} \rightarrow \mathbb{N}$ *indicates which part of r justifies having not q in* $\textsc{ct}(x)$.
- *The* negative justification for unary predicates *function* $\textsc{nj}_{\mathrm{U}} : \{(x, q, r) \mid x \in T, \mathit{not}\ q \in \textsc{ct}(x), r \in P_q\} \rightarrow 2^{\mathbb{N} \times T}$ *indicates by means of tuples $(n, z) \in \mathbb{N} \times T$ which literal $l_r(n)$ from r is used to justify not q in* $\textsc{ct}(x)$ *in a node $z \in T$, or edge $(x, z) \in A_T$.*
- *The* negative justification for binary predicates *function* $\textsc{nj}_{\mathrm{B}} : \{(x, q, r) \mid x \in A_T, \mathit{not}\ q \in \textsc{ct}(x), r \in P_q\} \rightarrow \mathbb{N}$ *gives the index of the literal from r that is used to justify not $q \in \textsc{ct}(x)$.*

An initial completion structure *for checking the satisfiability of a unary predicate p w.r.t. a simple CoLP P is a completion structure with $T = \{\varepsilon\}$, $V = \{p(\varepsilon)\}$, $E = \emptyset$, and $\textsc{ct}(\varepsilon) = \{p\}$, $\textsc{st}(\varepsilon, p) = unexp$, and the other labeling functions undefined for every input.*

We clarify the definition of a completion structure by means of an example. Take the program P :

$$\begin{array}{ll} r_1 : f(x,y) \vee not\ f(x,y) \leftarrow & \\ r_2 : & a(x) \leftarrow f(x,y_1), a(y_1), f(x,y_2) \\ r_3 : & b(x) \leftarrow not\ a(x) \end{array}$$

A possible completion structure for this program P is as follows. Take a tree $T = \{\varepsilon, \varepsilon 1\}$, i.e., a tree with root ε and successor $\varepsilon 1$, and take $\textsc{ct}(\varepsilon) = \{b, not\ a\}$, $\textsc{ct}(\varepsilon, \varepsilon 1) = \{f\}$, and $\textsc{ct}(\varepsilon 1) = \{not\ a, b\}$. Intuitively, we lay out the structure of our tree model.

We take $\textsc{rl}(\varepsilon, b) = r_3$ indicating that r_3 is responsible for motivating the occurrence of b in ε, set $\textsc{st}(\varepsilon, b) = exp$, and keep the status undefined for all other nodes and edges in T.

In general, justifying a negative unary literal $not\ q \in \textsc{ct}(x)$ (or in other words, the absence of $q(x)$ in the corresponding open interpretation) implies that every rule which defines q has to be refuted (otherwise q would have to be present), thus at least one body literal from every rule in P_q has to be refuted. A certain rule $r \in P_q$ can either be locally refuted (via a literal which can be formed using x and some $\pm a \in \textsc{ct}(x)$) or it has to be refuted in every successor of x. In the latter case, if x has more than one successor, it can be shown that the same segment of the rule has to be refuted in all the successors, whereby a segment of a rule is one of $\{\beta, (\gamma_m \cup \delta_m)_{1 \leq m \leq k}\}$ for unary rules (1). In the example, in order to have $not\ a \in \textsc{ct}(\varepsilon)$, we need that for all successors y_1, y_2, either $f \in \textsc{ct}(\varepsilon, y_1), a \in \textsc{ct}(y_1)$ does not hold, or $f \in \textsc{ct}(\varepsilon, y_2))$ does not hold; as y_1 is not appearing in the second segment (and vice versa for y_2), either for all successors y, $f \in \textsc{ct}(\varepsilon, y), a \in \textsc{ct}(y)$ does not hold, or for all successors y, $f \in \textsc{ct}(\varepsilon, y)$ does not hold, such that in our case $\textsc{sg}(x, a, r_2) = 1$ (the segment $f(x, y_1), a(y_1)$): the function $\textsc{sg}$ picks up such a segment to be refuted, where segments are referred to by the numbers 0 for β, and m for $\gamma_m \cup \delta_m$, $1 \leq m \leq k$.

After picking a segment to refute a negative unary predicate, we need means to indicate which literal in the segment, per successor, can be used to justify this negative unary predicate. This can be per successor a different literal from the segment such that $\textsc{nj}_{\textsc{u}}(x, q, r)$ is a set of tuples (n, z) where z is the particular successor (or x itself in case the negative unary predicate can be justified locally) and n the position of the literal in the rule r. In the example, $\textsc{nj}_{\textsc{u}}(x, a, r_2) = \{(1, \varepsilon 1)\}$, i.e., the literal $a(y_1)$ as $not\ a \in \textsc{ct}(\varepsilon 1)$. Note that if $z = x$ the set $\textsc{nj}_{\textsc{u}}(x, q, r)$ would be a singleton set as no successors are needed to justify $not\ q$.

Rules that can deduce negated binary predicates are always *local* in the sense that to justify a $not\ q \in \textsc{ct}(x)$ for $x \in A_T$ one only needs to consider x.

In the following, we will show how to expand the initial completion structure in order to prove satisfiability of a predicate, how to determine when no more

expansion is needed (*blocking*), and under what circumstances a *clash* occurs. In particular, *expansion rules* will expand an initial completion structure to a complete clash-free structure that corresponds to a finite representation of an open answer set; *applicability rules* state the necessary conditions such that those expansion rules can be applied.

4.1 Expansion Rules

The expansion rules will need to update the completion structure whenever in the process of justifying a literal l in the current model a new literal $\pm p(z)$ has to be considered. This means that $\pm p$ has to be inserted in the content of z in case it is not already there and marked as unexpanded, and in case $\pm p(z)$ is an atom, it has to be ensured that it is a node in G and furthermore, in case l is also an atom, a new arc from l to $\pm p(z)$ should be created to capture the dependencies between the two elements of the model. More formally:

- if $\pm p \notin \textsc{ct}(z)$, then $\textsc{ct}(z) = \textsc{ct}(z) \cup \{\pm p\}$ and $\textsc{st}(z, \pm p) = \mathit{unexp}$,
- if $\pm p = p$ and $\pm p(z) \notin V$, then $V = V \cup \{\pm p(x)\}$,
- if $l \in \mathcal{B}_{P_T}$ and $\pm p = p$, then $E = E \cup \{(l, \pm p(z))\}$.

As a shorthand, we denote this sequence of operations as $\mathit{update}(l, \pm p, z)$; more general, $\mathit{update}(l, \beta, z)$ for a set of (possibly negated) predicates β, denotes $\forall \pm a \in \beta, \mathit{update}(l, \pm a, z)$.

In the following, let $x \in T$ and $(x, y) \in A_T$ be the node, respectively edge, under consideration.

(i) Expand unary positive. For a unary positive predicate (non-free) $p \in \textsc{ct}(x)$ such that $\textsc{st}(x, p) = \mathit{unexp}$,

- nondeterministically choose a rule $r \in P_p$ of the form (1) that will motivate this predicate: set $\textsc{rl}(x, p) = r$,
- for the β in the body of this r, $\mathit{update}(p(x), \beta, x)$,
- for each $\gamma_m, 1 \leq m \leq k$, from r, nondeterministically choose a $y \in \mathit{succ}_T(x)$ or let $y = x \cdot s$, where $s \in \mathbb{N}_0^*$ s.t. $x \cdot s \notin \mathit{succ}_T(x)$ already. In the latter case, add y as a new successor of x in T: $T = T \cup \{y\}$. Take a new constant $c \in C$ s.t. $\forall z \in T : c \notin t(z)$ and update the label t for the newly created node: $t(y) = c$[2]. Next, $\mathit{update}(p(x), \gamma_m, (x, y))$ and $\mathit{update}(p(x), \delta_m, y)$.
- set $\textsc{st}(x, p) = \mathit{exp}$.

(ii) Expand unary negative. For a unary negative predicate (non-free) $\mathit{not}\ p \in \textsc{ct}(x)$ and either

[2] These constants keep track of the names of nodes in the tree, and will be useful in constructing the corresponding open answer set in the proofs of soundness and completeness; they have no role in the algorithm in itself

1. $\textsc{st}(x, \mathit{not}\ p) = \mathit{unexp}$, then for every rule $r \in P_p$ of the form (1) nondeterministically choose a segment $m, 0 \leq m \leq k$: $\textsc{sg}(x, p, r) = m$.
   - If $m = 0$, choose a $\pm a \in \beta$, and $\mathit{update}(\mathit{not}\ p(x), \mp a, x)$, $\textsc{nj}_{\text{U}}(x, p, r) = \{(i_r(\pm a(X)), x)\}$.
   - If $m > 0$, for every $y \in \mathit{succ}_T(x)$, (†) choose a $\pm a_y \in \gamma_m \cup \delta_m$, and set $\textsc{nj}_{\text{U}}(x, p, r) = \{(i_r(\pm a_y(X, Y_m)), y) \mid \pm a_y \in \gamma_m\} \cup \{(i_r(\pm a_y(Y_m)), y) \mid \pm a_y \in \delta_m\}$. Next, $\mathit{update}(\mathit{not}\ p(x), \mp a_y, (x, y))$ if $\pm a_y \in \gamma_m$, and $\mathit{update}(\mathit{not}\ p(x), \mp a_y, y)$ if $\pm a_y \in \delta_m$.

   After every rule has been processed set $\textsc{st}(x, \mathit{not}\ p) = \mathit{exp}$.
2. $\textsc{st}(x, \mathit{not}\ p) = \mathit{exp}$ and for some $r \in P_p$, $\textsc{sg}(x, p, r) \neq 0$, and $\textsc{nj}_{\text{U}}(x, p, r) = S$ with $|S| < |\mathit{succ}_T(x)|$, i.e., *not* p has already been expanded, but for some rule r it did not receive a local justification (at x), and meanwhile new successors of x have been introduced. Thus, one has to justify *not* p in the new successors as well.

   For every $r \in P_p$ of the form (1) such that $\textsc{sg}(x, p, r) = m \neq 0$ and for every $y \in \mathit{succ}_T(x)$ which has not been yet considered previously, repeat the operations in (†) as above.

(iii) Expand binary positive. For a binary positive predicate symbol (non-free) p in $\textsc{ct}(x, y)$ such that $\textsc{st}((x, y), p) = \mathit{unexp}$: nondeterministically choose a rule $r \in P_p$ of the form (2) that motivates p by setting $\textsc{rl}((x, y), p) = r$, and $\mathit{update}(p(x, y), \beta, x)$, $\mathit{update}(p(x, y), \gamma, (x, y))$, and $\mathit{update}(p(x, y), \delta, y)$. Finally, set $\textsc{st}((x, y), p) = \mathit{exp}$.

(iv) Expand binary negative. For a binary negative predicate symbol (non-free) *not* p in $\textsc{ct}(x, y)$ such that $\textsc{st}((x, y), \mathit{not}\ p) = \mathit{unexp}$, nondeterministically choose for every rule $r \in P_p$ of the form (2) an s from $\mathit{varset}_r(X)$, $\mathit{varset}_r(X, Y)$ or $\mathit{varset}_r(Y)$ and let $\textsc{nj}_{\text{B}}((x, y), p, r) = s$.

- If $s \in \mathit{varset}(X)$ and $\pm a(X) = l_r(s)$, $\mathit{update}(\mathit{not}\ p(x, y), \mp a, x)$,
- If $s \in \mathit{varset}(X, Y)$ and $\pm f(X, Y) = l_r(s)$, $\mathit{update}(\mathit{not}\ p(x, y), \mp f, (x, y))$,
- If $s \in \mathit{varset}(Y)$ and $\pm a(Y) = l_r(s)$, $\mathit{update}(\mathit{not}\ p(x, y), \mp a, y))$.

Finally, set $\textsc{st}((x, y), \mathit{not}\ p) = \mathit{exp}$.

(v) Choose a unary predicate. There is an $x \in T$ for which none of $\pm a \in \textsc{ct}(x)$ can be expanded with rules (i-ii), and for all $(x, y) \in A_T$, none of $\pm f \in \textsc{ct}(x, y)$ can be expanded with rules (iii-iv), and there is a $p \in \mathit{upreds}(P)$ such that $p \notin \textsc{ct}(x)$ and $\mathit{not}\ p \notin \textsc{ct}(x)$. Then, add p to $\textsc{ct}(x)$ with $\textsc{st}(x, p) = \mathit{unexp}$ or add *not* p to $\textsc{ct}(x)$ with $\textsc{st}(x, \mathit{not}\ p) = \mathit{unexp}$.

(vi) Choose a binary predicate. There is an $x \in T$ for which none of $\pm a \in \textsc{ct}(x)$ can be expanded with rules (i-ii), and for all $(x, y) \in A_T$ none of $\pm f \in \textsc{ct}(x, y)$ can be expanded with rules (iii-iv), and there is a $(x, y) \in A_T$ and a $p \in \mathit{bpreds}(P)$ such that $p \notin \textsc{ct}(x, y)$ and $\mathit{not}\ p \notin \textsc{ct}(x, y)$. Then, add p to $\textsc{ct}(x, y)$ with $\textsc{st}((x, y), p) = \mathit{unexp}$ or add *not* p to $\textsc{ct}(x, y)$ with $\textsc{st}((x, y), \mathit{not}\ p) = \mathit{unexp}$.

4.2 Applicability Rules

For a simple CoLP P, a universe U for P, a graph $G = \langle V, E\rangle$ with nodes $V \in \mathcal{B}_{P_U}$ and $E \in \mathcal{B}_{P_U} \times \mathcal{B}_{P_U}$, and a set of constants $C \subseteq U$ we denote by $G(C)$ the graph obtained from G by considering only those nodes $V(C) \subseteq V$ which have an element from C as a first argument (remember that nodes are unary or binary literals) and the edges $E(C)$ which already existed between the nodes from $V(C)$ in the initial graph. Formally, $V(C) = \{\pm p(x) \mid \pm p(x) \in V \wedge x \in C\} \cup \{\pm p(x, y) \mid \pm p(x, y) \in V \wedge x \in C\}$ and $E(C) = E \cap (V(C) \times V(C))$. When U is a tree and C is a path in U, $C = path_U(x, y)$, $G(C)$ will contain those nodes from G which have as arguments nodes from C or outgoing arcs in U from nodes in C.

A second set of rules is not updating the completion structure under consideration, but restricts the use of the expansion rules:

(vii) Saturation We will call a node $x \in T$ *saturated* if

- for all $p \in upreds(P)$ we have $p \in \text{CT}(x)$ or *not* $p \in \text{CT}(x)$ and none of $\pm a \in \text{CT}(x)$ can be expanded according to the rules (i-ii) or (v),
- for all $(x, y) \in A_T$ and $p \in bpreds(P)$, $p \in \text{CT}(x, y)$ or *not* $p \in \text{CT}(x, y)$ and none of $\pm f \in \text{CT}(x, y)$ can be expanded according to the rules (iii-iv) or (vi).

We impose that no expansions (i-vi) can be performed on a node from T until its predecessor is saturated.

(viii) Blocking We call a node $x \in T$ *blocked* if

- its predecessor is saturated,
- there are two ancestors y, z such that $y < z < x$, and $\text{CT}(z) = \text{CT}(y)$, and
- $G_{y,z} = \langle V(path_T(y, z)), E(path_T(y, z)) \cup \{(a(z), a(y)) \mid a \in \text{CT}(z)\}\rangle$ is acyclic.

Note that ancestors y, z are saturated as well, by rule (vii).

Intuitively, if there is a pair of ancestor nodes that have equal content, and if by adding connections from atoms formed using the lower node in the pair to atoms formed using the higher node in the pair and the same predicate, no cycles are created in a subgraph of G which has as nodes all nodes which have as arguments nodes from the path or outgoing arcs from these nodes (the rest of G is not relevant in this context), the current node can be blocked: one can show that provided that the content of the higher node in the pair is justified, the content of the lower node in the pair can be justified also without further expansions. We call (y, z) a *blocking pair* and say that y *blocks* z; if no confusion is possible with the blocked node x, we will usually also refer to z as a blocked node and to y as the blocking node for a blocking pair (y, z). We impose that no expansions (i-vi) can be performed on a blocked node from T.

(ix) Cyclic We call a node $x \in T$ *cyclic* if

- its predecessor is saturated,
- there are two ancestors y, z such that $y < z < x$, and $\text{CT}(z) = \text{CT}(y)$, and
- $G_{y,z}$ contains a cycle.

The intuition is similar as with blocking, however, instead of being able to reuse the justification of the higher node for the lower node, the presence of a cycle in $G_{y,z}$ is indicating that we would create an infinitely positive path in G to motivate the higher node, when reusing the justification, which is, due to the minimal model semantics, not allowed. We call (y, z) a *cyclic* pair, and, when no confusion can arise, we designate z as a cyclic node as well. We impose that no expansions (i-vi) can be performed on any node in T if it contains a cyclic node.

(x) Caching We call a node $x \in T$ *cached* if

- its predecessor is saturated,
- there are two nodes y, z such that $z < x$ and $z \not\leq y$ and $y \not\leq z$ (i.e., z is not an ancestor of y nor is y an ancestor of z), and $\text{CT}(z) = \text{CT}(y)$; we call the pair (y, z) a *caching pair* and we say that y *caches* z; usually, if no confusion is possible, we will also refer to z as a cached node and to y as the caching node. Furthermore, there exists no caching pair (u, y).

We impose that no expansions can be performed on a cached node from T. Intuitively, x is not further expanded, as one can reuse the (cached) justification for y when dealing with z. The condition that there is no caching pair (u, y) ensures that we do not replace the justification of z with that one of a node y which at its turn is reusing the justification of some other node u. In particular, this precludes a situation like y caches z and z caches y.

4.3 Termination, Soundness, and Completion

We call a completion structure *contradictory*, if for some $x \in T$ and $a \in \mathit{upreds}(P)$, $\{a, \mathit{not}\ a\} \subseteq \text{CT}(x)$ or for some $(x, y) \in A_T$ and $f \in \mathit{bpreds}(P)$, $\{f, \mathit{not}\ f\} \subseteq \text{CT}(x, y)$. A *complete completion structure* for a simple CoLP P and a $p \in \mathit{upreds}(P)$, is a completion structure that results from applying the expansion rules to the initial completion structure for p and P, taking into account the applicability rules, such that no expansion rules can be further applied. Furthermore, a complete completion structure $CS = \langle T, G, \text{CT}, \text{ST}, \text{RL}, \text{SG}, \text{NJ}_\text{U}, \text{NJ}_\text{B}\rangle$ is *clash-free* if (1) CS is not contradictory, (2) t does not contain cyclic nodes, and (3) G does not contain cycles.

We show that an initial completion structure for a unary predicate p and a simple CoLP P can always be expanded to a complete completion structure (*termination*), that, if p is satisfiable w.r.t. P, there is a clash-free complete completion structure (*soundness*), and, finally, that, if there is a clash-free complete completion structure, p is satisfiable w.r.t. P (*completeness*).

Proposition 3 (termination). *Let P be a simple CoLP and $p \in upreds(P)$. Then, one can construct a finite complete completion structure by a finite number of applications of the expansion rules to the initial completion structure for p and P, taking into account the applicability rules.*

Proof Sketch. Assume one cannot construct a complete completion structure by a finite number of applications of the expansion rules, taking into account the applicability rules. Clearly, if one has a finite completion structure that is not complete, a finite application of expansion rules would complete it unless successors are introduced. However, one cannot introduce infinitely many successors: every infinite path in the tree will eventually contain two saturated nodes with equal content and thus either a blocked or a cyclic pair, such that no expansion rules can be applied to successor nodes of the blocked or cyclic node in the pair. Furthermore, the arity of the tree in the completion structure is bound by the predicates in P and the degrees of the rules. □

Proposition 4 (soundness). *Let P be a simple CoLP and $p \in upreds(P)$. If there exists a clash-free complete completion structure for p w.r.t. P, then p is satisfiable w.r.t. P.*

Proof Sketch. From a complete clash-free completion structure for p and P we can construct an open answer set of P that satisfies p by unfolding the completion structure. Intuitively, blocking pairs represent a state where the open answer set contains some infinitely repeating pattern that consists of a finite motivation for the literals in the blocking pair: the definition of a blocking pair is such that when we replace the motivation for the blocked node (i.e., the subtree below this node) by the subtree that motivates the blocking node in the pair, no infinite positive path arises. As the subtree of the blocked node is a subtree of the subtree of the blocking node, we need to repeat such a replacement infinitely. Furthermore, cached nodes represent the situation that a motivation for a node is being repeated elsewhere, such that also cached pairs will be removed by a substitution of subtrees. One can show that such a construction results in a tree model for the program. □

Proposition 5 (completeness). *Let P be a simple CoLP and $p \in upreds(P)$. If p is satisfiable w.r.t. P, then there exists a clash-free complete completion structure for p w.r.t. P.*

Proof Sketch. If p is satisfiable w.r.t. P then p is tree satisfiable w.r.t. P (Proposition 1), such that there must be a tree model (U, M) for p w.r.t. P.

One can construct a clash-free complete completion structure for p w.r.t. P, by guiding the nondeterministic application of the expansion rules by (U, M) and taking into account the constraints imposed by the saturation, blocking, caching, and clash rules.

It is worth noting that the naive application of the rules according to (U, M) does not work: the tree model might contain cyclic patterns that would result in cyclic nodes in the completion structure. However, such patterns cannot occur

infinitely (this would contradict the minimality of an open answer set), such that we can choose those expansion rules that bypass the cyclicity and immediately choose the finite motivation for a certain node. □

4.4 Complexity Results

Let $CS = \langle T, G, \text{CT}, \text{ST}, \text{RL}, \text{SG}, \text{NJ}_\text{U}, \text{NJ}_\text{B}\rangle$ be a completion structure and CS' the completion structure constructed from CS by removing from T all subtrees with roots y where (x, y) is some blocked, cyclic, or caching pair. The size of each of these subtrees is at most $k + 1$, where k is bound by the amount n of unary predicates q in P and the degrees of the rules P_q. Moreover, there are at most mk such subtrees, where m is the amount of nodes in CS'.

Assume CS' has more than 2^n nodes, then there must be two nodes $x \neq y$ such that $\text{CT}(x) = \text{CT}(y)$. If $x < y$ or $y < x$, either (x, y) or (y, x) is a blocked or cyclic pair, which contradicts the construction of CS'. If $x \not< y$ and $y \not< x$, (x, y) or (y, x) is a caching pair, again a contradiction. Thus, CS' contains at most 2^n nodes, so $m \leq 2^n$. Since CS' resulted from CS by removing at most mk subtrees of maximal size $k+1$ each, the amount of nodes in CS is $m+m(k+1) \leq (k+2)2^n$, i.e., exponential in the size of P, such that the algorithm has to visit a number of nodes that is exponential in the size of P.

The graph G has as well a number of nodes that is exponential in the size of P. Since checking for cycles in a directed graph can be done in linear time, the algorithm runs in NEXPTIME, a nondeterministic level higher than the worst-case complexity characterization (Proposition 2).

Note that such an increase in complexity is expected. For example, although satisfiability checking in $\mathcal{SHIQ}$ is EXPTIME-complete, practical algorithms run in 2-NEXPTIME [18]. Thanks to caching, however, we only have an increase to NEXPTIME.

5 Related Work

Description Logic Programs [10] represent the common subset of OWL-DL ontologies and Horn logic programs (programs without negation as failure or disjunction). As such, reasoning can be reduced to normal LP reasoning.

In [16], a clever translation of $\mathcal{SHIQ}(\mathbf{D})$ ($\mathcal{SHIQ}$ with data types) combined with *DL-safe rules* (a rule is DL-safe if each variable in the rule appears in a non-DL-atom, where a DL-atom is an atom with the predicate corresponding to a DL-concept or DL-role) to disjunctive Datalog is provided. The translation relies on a translation to clauses and subsequently applying techniques from basic superposition theory.

Reasoning in $\mathcal{DL}$+*log* [17] does not use a translation to other approaches, but defines a specific algorithm based on a partial grounding of the program and a test for containment of conjunctive queries over the DL knowledge bases. Note that [17] has a *standard names assumption* as well as a *unique names assumption* - all interpretations are over some fixed, countably infinite domain,

different constants are interpreted as different elements in that domain, and constants are in one-to-one correspondence with that domain.

dl-programs [5] have a more loosely coupled take on integrating DL knowledge bases and logic programs by allowing the program to query the DL knowledge base while as well having the possibility to send (controlled) input to the DL knowledge base. Reasoning is done via a stable model computation of the logic program, interwoven with queries that are oracles to the DL part.

Description Logic Rules[14] are defined as decidable fragments of SWRL. The rules have a tree-like structure similar to the structure of simple CoLPs rules. Depending on the underlying DL, one can distinguish between $\mathcal{SROIQ}$ rules (these do not actually extend $\mathcal{SROIQ}$, they are just syntactic sugar on top of the language), $\mathcal{EL}^{++}$ rules, DLP rules, and ELP rules [15]. The latter can be seen as an extension of both $\mathcal{EL}^{++}$ rules and DLP rules, hence their name.

The algorithm presented in Section 4 can be seen as a procedure that constructs a tableau (as is common in most DL reasoning procedures), representing the possibly infinite open answer set by a finite structure. There are several DL-based approaches which adopt a minimal-style semantics. Among this are autoepistemic[4], default[2] and circumscriptive extensions of DL[3][9]. The first two extensions are restricted to reasoning with explicitly named individuals only, while [9] allows for defeats to be based on the existence of unknown individuals. A tableau-based method for reasoning with the DL $\mathcal{ALCO}$ in the circumscriptive case has been introduced in [8]. A special preference clash condition is introduced there to distinguish between minimal and non-minimal models which is based on constructing a new classical DL knowledge base and checking its satisfiability. It would be interesting to explore the connections between our algorithm and the algorithm described there, in particular between our graph-cycle based clash condition and the preference clash condition.

6 Conclusions and Outlook

We identified a decidable class of programs, simple CoLPs, and provided a non-deterministic algorithm for checking satisfiability under the open answer set semantics that runs in NEXPTIME.

The presented algorithm is the first step in reasoning under an open answer set semantics. We intend to extend the algorithm such that it can handle the inverse predicates and inequalities of CoLPs, as well as constants. The latter would enable combined reasoning with the DL $\mathcal{SHOIQ}$ (closely related to OWL-DL) and expressive rules.

References

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

2. F. Baader and B. Hollunder. Embedding defaults into terminological representation systems. *J. of Automated Reasoning*, 14(2):149–180, 1995.
3. P. Bonatti, C. Lutz, and F. Wolter. Expressive non-monotonic description logics based on circumscription. In *Proc. of 10th Int. Conf. on Principles of Knowledge Repr. and Reasoning (KR'06)*, pages 400–410, 2006.
4. F. M. Donini, D. Nardia, and R.Rosati. Description logics of minimal knowledge and negation as failure. *ACM Transactions on Comput. Logic*, 3(2):177–225, 2002.
5. T. Eiter, G. Ianni, T. Lukasiewicz, R. Schindlauer, and H. Tompits. Combining answer set programming with description logics for the semantic web. *Artificial Intelligence*, 172(12-13):1495–1539, 2008.
6. C. Feier and S. Heymans. A sound and complete algorithm for simple conceptual logic programs. Technical Report INFSYS RESEARCH REPORT 184-08-10, KBS Group, Technical University Vienna, Austria, October 2008. http://www.kr.tuwien.ac.at/staff/heymans/priv/projects/fwf-doasp/alpsws2008-tr.pdf.
7. M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *Proc. of ICLP'88*, pages 1070–1080, Cambridge, Massachusetts, 1988.
8. S. Grimm and P. Hitzler. Reasoning in circumscriptive $\mathcal{ALCO}$. Technical report, FZI at University of Karlsruhe, Germany, September 2007.
9. S. Grimm and P. Hitzler. Defeasible inference with circumscriptive OWL ontologies. In *Workshop on Advancing Reasoning on the Web: Scalability and Commonsense*, 2008.
10. B. N. Grosof, I. Horrocks, R. Volz, and S. Decker. Description logic programs: combining logic programs with description logic. In *Proc. of the World Wide Web Conf.*, pages 48–57. ACM, 2003.
11. S. Heymans, J. de Bruijn, L. Predoiu, C. Feier, and D. Van Nieuwenborgh. Guarded hybrid knowledge bases. *Theory and Practice of Logic Programming*, 8(3):411–429, 2008.
12. 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, June 2006.
13. S. Heymans, D. Van Nieuwenborgh, and D. Vermeir. Open answer set programming with guarded programs. *ACM Transactions on Computational Logic (TOCL)*, 9(4), October 2008.
14. M. Krötzsch, S. Rudolph, and P. Hitzler. Description logic rules. In *Proc. 18th European Conf. on Artificial Intelligence(ECAI-08)*, pages 80–84. IOS Press, 2008.
15. M. Krötzsch, S. Rudolph, and P. Hitzler. ELP: Tractable rules for OWL 2. In *Proc. 7th Int. Semantic Web Conf. (ISWC-08)*, 2008.
16. B. Motik, U. Sattler, and R. Studer. Query answering for OWL-DL with rules. *Journal of Web Semantics: Science, Services and Agents on the World Wide Web*, 3(1):41–60, July 2005.
17. R. Rosati. DL+log: Tight integration of description logics and disjunctive datalog. In *Proc. of the Int. Conf. on Principles of Knowledge Representation and Reasoning (KR)*, pages 68–78, 2006.
18. S. Tobies. *Complexity Results and Practical Algorithms for Logics in Knowledge Representation.* PhD thesis, LuFG Theoretical Computer Science, RWTH-Aachen, Germany, 2001.
19. 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-Verlag, 1998.