I N F S Y S

R E S E A R C H

R E P O R T

INSTITUT FÜR INFORMATIONSSYSTEME

ARBEITSBEREICH WISSENSBASIERTE SYSTEME

A SOUND AND COMPLETE ALGORITHM FOR SIMPLE CONCEPTUAL LOGIC PROGRAMS

Cristina Feier Stijn Heymans

INFSYS RESEARCH REPORT 184-08-10.

OCTOBER 2008

Institut für Informationssysteme
AB Wissensbasierte Systeme
Technische Universität Wien
Favoritenstrassße 9-11
A-1040 Wien, Austria
Tel: +43-1-58801-18405
Fax: +43-1-58801-18493
sek@kr.tuwien.ac.at
www.kr.tuwien.ac.at

TU WIEN

INFSYS RESEARCH REPORT

INFSYS RESEARCH REPORT 184-08-10., OCTOBER 2008

A SOUND AND COMPLETE ALGORITHM FOR SIMPLE CONCEPTUAL LOGIC PROGRAMS

Cristina Feier[1] and Stijn Heymans[2]

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]Institute of Information Systems, Knowledge-Based Systems Group, Vienna University of Technology, Favoritenstraße 9-11, A-1040 Vienna, Austria. E-mail: feier@kr.tuwien.ac.at.

[2]Institute of Information Systems, Knowledge-Based Systems Group, Vienna University of Technology, Favoritenstraße 9-11, A-1040 Vienna, Austria. E-mail: heymans@kr.tuwien.ac.at.

Acknowledgements: 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)*.

This Report is an extended version of a paper that appeared in the proceedings of the ICLP08 Workshop on Applications of Logic Programming on the (Semantic) Web and Web Services (ALPSWS 2008), December 2008.

Contents

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* [9], *DL-safe rules* [16], $\mathcal{DL}$+*log* [17], *dl-programs* [5], and Open Answer Set Programming (OASP) [12]. 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 mention of actual constants, and a rule-based syntax from the LP side supports nonmonotonic reasoning via *negation as failure*.

Decidable fragments for OASP satisfiability checking, like *Conceptual Logic Programs* [11] or *g-hybrid knowledge bases* [10], were identified by syntactically restricting OASP. These fragments are still expressive enough for integrating rule- and ontology-based knowledge. However there are no effective reasoning procedures for any of these decidable fragments of OASP so far. 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, inequality, and have some restrictions concerning the dependencies between different predicate symbols which appear in rules, compared to CoLPs, but are expressive enough to simulate the DL $\mathcal{ALCH}$. We show that 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.

2 Preliminaries

We recall the open answer set semantics from [12]. *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 negated 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 binary 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* [6] 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 $x \in \mathbb{N}_0^*$, i.e., a finite sequence of natural numbers (excluding 0), 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 $\forall x \neq y \in P : |x| \neq |y|$. A branch B in a tree T is a maximal path (there is no path which contains it) which contains the root of T.

A *labeled tree* is a pair (T, t) where T is a tree and $t : T \to \Sigma$ is a labeling function; sometimes we will identify the tree (T, t) with t. We denote the *subtree* of T at x by $T[x]$, i.e., $T[x] = \{y \in T \mid x \leq y\}$. For labeled trees $t : T \to \Sigma$, the subtree of t at $x \in T$ is $t[x] : T[x] \to \Sigma$ such that $t[x](y) = t(y)$ for $y \in T[x]$. For a tree $t : T \to \Sigma$, a tree $s : S \to \Sigma$, and a symbol $a \in \Sigma$, we denote with $t \cdot_a s$, the tree t with the subtrees starting with the first node on every path with label a (in case such a node exists) replaced by s. Consider the trees s and t depicted in Figure 1. The tree resulted by the application of $t \cdot_a s$ is depicted in Figure 2.

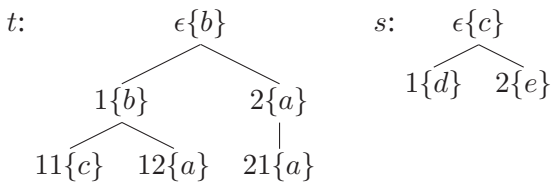

Figure 1: Two labeled trees: t and s

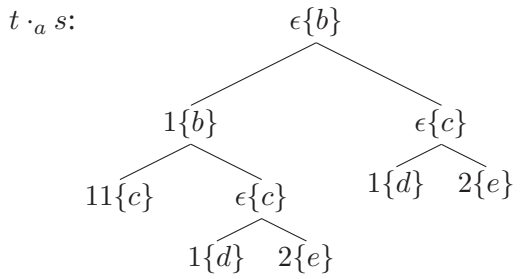

Figure 2: The new tree: $t \cdot_a s$

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, where 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 $t : U \rightarrow 2^{preds(P)}$ 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 $y = x \cdot i \wedge f \in t(y)$

The *label* $\mathcal{L}(z)$ of a node $z \in U$ is $\mathcal{L}(z) = \{q \mid q \in t(z), q \in upreds(P)\}$. We call such a (U, M) a *tree model* for p w.r.t. P. Note that binary predicates are maintained in the labels of nodes: a binary predicate f in the label of $x \cdot i$ indicates a connection $f(x, x \cdot i)$.

3 Simple Conceptual Logic Programs

In [11], 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, and by restricting the dependencies between predicate symbols which appear in the program. The resulting fragment is called in *Simple Conceptual Logic Programs*.

Definition 3.1 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 $\gamma^+ \neq \emptyset$.

Furthermore, let $D(P)$ be the *marked predicate dependency graph* of a program P as defined above, where $D(P)$ has as vertices the predicates from P and as arcs tuples (p, q), where there is either a rule (1) or a rule (2) with a head predicate p and a positive body predicate q; we call an arc (p, q) marked if q is a predicate in δ_m or δ for rules (1), respectively rules (2). P is a simple CoLP iff its marked predicate dependency graph $D(P)$ does not contain any cycle with a marked edge.

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 literals involving x and y. The restriction concerning the marked dependency graph can be translated in the following terms: there is no path from a $p(x)$ to a $p(y)$ in the literal dependency graph of P_U, where p is a unary predicate from P, U is an arbitrary universe, and x and y are two distinct elements from U.

Simple CoLPs can simulate constraints $\leftarrow \beta(x), \left(\gamma_m(x, y_m), \delta_m(y_m)\right)_{1 \leq m \leq k}$, where for all 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 [11], simple CoLPs have the tree model property as well.

Proposition 3.2 *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 [11], checking satisfiability of simple CoLPs is in EXPTIME.

In [11], 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 $\mathcal{ALCH}$ concepts (i.e., $\mathcal{SHIQ}$ without inverse roles and quantified number restrictions) w.r.t. a $\mathcal{ALCH}$ 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) or transitive roles (due to the fact that they do not allow for positive literals in the successor part of a rule). As satisfiability checking of $\mathcal{ALC}$ concepts w.r.t. an $\mathcal{ALC}$ TBox (note that $\mathcal{ALC}$ is a fragment of $\mathcal{ALCH}$) is EXPTIME-complete ([1, Chapter 3]), we have EXPTIME-hardness for simple CoLPs as well.

Proposition 3.3 *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 $\pm p$ (in a given context) is $\mp p$, that is, $\mp p = not\ p$ if $\pm p = p$ and $\mp p = p$ if $\pm p = not\ p$.

For a unary rule r of the form (1), we define $degree(r) = |\{m \mid \gamma_m \neq \emptyset\}|$. 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 $varset_r : \{x, y_1, \ldots, y_k, (x, y_1), \ldots, (x, y_k)\} \rightarrow 2^{\{0,\ldots,|\beta|\}}$ 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 4.1 [completion structure] A *completion structure for a simple CoLP* P is a tuple $\langle T, G, \mathrm{CT}, \mathrm{ST}, \mathrm{RL}, \mathrm{SG}, \mathrm{NJ_U}, \mathrm{NJ_B}\rangle$. T is a tree which together with the labeling functions CT, ST, RL, SG, $\mathrm{NJ_U}$, and $\mathrm{NJ_B}$, is used to represent/construct a tentative tree model (the nodes of the tree are elements of the universe w.r.t. which the model is constructed). $G = \langle V, E\rangle$ 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 used to keep track of dependencies between elements of the constructed model, V being the model itself). Below the signature and the role for each labeling function is given:

- The *content* function $\mathrm{CT} : T \cup A_T \rightarrow 2^{preds(P) \cup not\ (preds(P))}$ 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 $\mathrm{CT}(x) \in upreds(P) \cup not(upreds(P))$ if $x \in T$, and $\mathrm{CT}(x) \in bpreds(P) \cup not(bpreds(P))$ if $x \in A_T$. Every positive appearance of a predicate symbol p in the content of some node/arc x of T indicates that $p(x)$ is part of the tentative model represented by T.

- The *status* function $\mathrm{ST} : \{(x, \pm q) \mid \pm q \in \mathrm{CT}(x), x \in T \cup A_T\} \rightarrow \{exp, unexp\}$ 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. As it will be indicated later, the completion structure is evolved such that the presence of any (possibly negated) predicate symbol in the content of some node/arc is justified, so it is necessary to keep track which predicate symbols have already been justified in every node/arc of T.

- The *rule* function $\mathrm{RL} : \{(x, q) \mid x \in T \cup A_T, q \in \mathrm{CT}(x)\} \rightarrow P$ associates with every node/edge x of T and every positive predicate $q \in \mathrm{CT}(x)$ a rule which has q as a head predicate: $\mathrm{RL}(x, q) \in P_q$.

- The *segment* function $\mathrm{SG} : \{(x, q, r) \mid x \in T, not\ q \in \mathrm{CT}(x), r \in P_q\} \rightarrow \mathbb{N}$ indicates which part of r justifies having $not\ q$ in $\mathrm{CT}(x)$.

- The *negative justification for unary predicates* function $\mathrm{NJ_U} : \{(x, q, r) \mid x \in T, not\ q \in \mathrm{CT}(x), r \in P_q\} \rightarrow 2^{\mathbb{N} \times T}$ 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 $\mathrm{CT}(x)$ in a node $z \in T$, or edge $(x, z) \in A_T$.

- The *negative justification for binary predicates* function $\mathrm{NJ_B} : \{(x, q, r) \mid x \in A_T, not\ q \in \mathrm{CT}(x), r \in P_q\} \rightarrow \mathbb{N}$ gives the index of the literal from r that is used to justify $not\ q \in \mathrm{CT}(x)$.

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 $\mathrm{CT}(\varepsilon) = \{p\}$, $\mathrm{ST}(\varepsilon, p) = unexp$, and the other labeling functions are undefined for every input.

We clarify the definition of a completion structure by means of an example. Consider the simple CoLP P:

$$
\begin{array}{lrcl}
r_1: & restore(X) & \leftarrow & crash(X), y(X, Y), backSucc(Y) \\
r_2: & backSucc(X) & \leftarrow & not\ crash(X), y(X, Y), not\ backFail(Y) \\
r_3: & backFail(X) & \leftarrow & not\ backSucc(X) \\
r_4: & yesterday(X, Y) \vee not\ yesterday(X, Y) & \leftarrow & \\
r_5: & crash(X) \vee not\ crash(X) & \leftarrow &
\end{array}
$$

Note that while there is a marked arc in $D(P)$, $(restore, backSucc)$, there is no cycle which contains it, so P is indeed a simple CoLP. An *initial completion structure* for checking the satisfiability of the unary predicate $restore$ w.r.t. P is depicted in Figure 3.

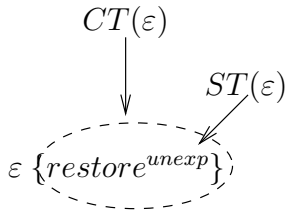

Figure 3: Initial completion structure

Intuitively, we created the root of our tree model which contains the predicate whose satisfiability is tested, $restore$ and set the state for this predicate w.r.t. the current node to be unexpanded.

At this stage the graph G consists of the single node $restore(\varepsilon)$ and naturally, no arcs.

In the following, we will show how to expand an initial completion structure to prove the satisfiability of a unary predicate p w.r.t. a simple CoLP P, how to determine when no more expansion is needed (*blocking*), and under what circumstances a *clash* occurs. In particular, *expansion rules* will evolve a completion structure starting with an initial completion structure for checking satisfiability of p w.r.t. P to a complete clash-free structure that corresponds to a finite representation of an open answer set in case p is satisfiable w.r.t. P. *Applicability rules* 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 (either as making part of the model, in case the literal is an atom, or as not making part of the model, in case the literal is a negated atom). 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 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 $\textsc{st}(z, \pm p) = unexp$,
- 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 $update(l, \pm p, z)$; more general, $update(l, \beta, z)$ for a set of (possibly negated) predicates β, denotes $\forall \pm a \in \beta, update(l, \pm a, z)$.

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

4.1.1 (i) Expand unary positive.

For a unary positive predicate (non-free) $p \in \text{CT}(x)$ such that $\text{ST}(x, p) = unexp$,

- nondeterministically choose a rule $r \in P_p$ of the form (1) that will motivate this predicate: set $\text{RL}(x, p) = r$,
- for the β in the body of this r, $update(p(x), \beta, x)$,
- for each $m, 1 \leq m \leq k$, nondeterministically choose a $y \in succ_T(x)$ or let $y = x \cdot s$, where $s \in \mathbb{N}_0^*$ s.t. $x \cdot s \notin succ_T(x)$ already. In the latter case, add y as a new successor of x in T: $T = T \cup \{y\}$. Next, $update(p(x), \gamma_m, (x, y))$ and $update(p(x), \delta_m, y)$.
- set $\text{ST}(x, p) = exp$.

In our example, the initial completion structure contains an unexpanded unary predicate, $restore$. The result of applying the rule above to this predicate is depicted in Figure 4.

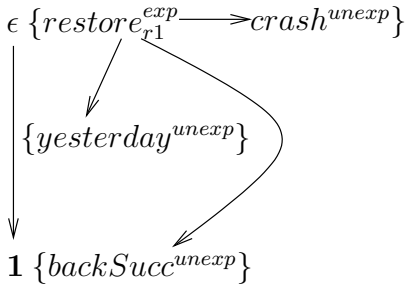

Figure 4: Expansion of a unary positive predicate symbol

The figure indicates that rule r_1, the only rule which defines $restore$, has been used to expand the predicate. The local part of the rule, $crash$, has been injected in the content of the local node, $\text{CT}(roo)$ and a new successor 1 was created in which the part of the rule corresponded to y was injected. The transition between the root and its successor is done by the binary predicate $yesterday$ which is injected in the content of the arc $(\varepsilon, 1)$. Also the new injected predicates together with their corresponding tree node form literals which are new vertices in G. An arc is created from $restore(\varepsilon)$ to each of these new literals.

4.1.2 (ii) Choose a unary predicate.

There is an $x \in T$ for which none of $\pm a \in \text{CT}(x)$ can be expanded with rules (i) and (iii), and for all $(x, y) \in A_T$, none of $\pm f \in \text{CT}(x, y)$ can be expanded with rules (iv-v) (we decided to place this rule before the other rules mentioned here, for the sake of the example continuity), and there is a $p \in upreds(P)$ such

that $p \notin \text{CT}(x)$ and $not\ p \notin \text{CT}(x)$. Then, add p to $\text{CT}(x)$ with $\text{ST}(x, p) = unexp$ or add $not\ p$ to $\text{CT}(x)$ with $\text{ST}(x, not\ p) = unexp$.

This rule says that in case there is a node x for which all the predicate symbols in its content and in the contents of its outgoing arcs were expanded and there are still unary predicate symbols which do not appear in the content of the current node, one has to pick such a unary predicate symbol p and to inject either p or $notp$ in $\text{CT}x$. This is needed for consistency reasons: it is not enough to find a justification for the predicate we want to prove that is satisfiable, but one has to show also that this justification makes part from an actual model, which is done by actually constructing such a model. Consider the completion structure described in Figure 4. We observe that the conditions described in this rule are fulfilled for the root of the tree ε. We pick $backSucc$ as a new unary predicate symbol which does not appear in $\text{CT}(\varepsilon)$ and inject $not\ backSucc$ in $\text{CT}(roo)$. This process is described in Figure 5.

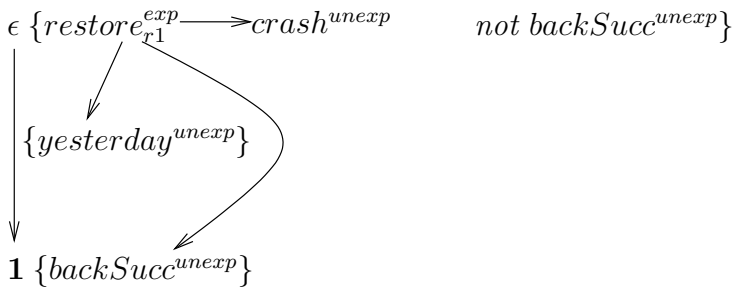

Figure 5: Choose a unary predicate

4.1.3 (iii) Expand unary negative.

For a unary negative predicate (non-free) $not\ p \in \text{CT}(x)$ and either

1. $\text{ST}(x, not\ p) = unexp$, then for every rule $r \in P_p$ of the form (1) nondeterministically choose a segment $m, 0 \leq m \leq k$: $\text{SG}(x, p, r) = m$.

   - If $m = 0$, choose a $\pm a \in \beta$, and $update(not\ p(x), \mp a, x)$, $\text{NJ}_\text{U}(x, p, r) = \{(i_r(\pm a(X)), x)\}$.
   - If $m > 0$, for every $y \in succ_T(x)$, (†) choose a $\pm a_y \in \gamma_m \cup \delta_m$, and set $\text{NJ}_\text{U}(x, p, r) = \{(i_r(\pm a_y(X, Y_m)), y) \mid \pm a_y \in \gamma_m\} \cup \{(i_r(\pm a_y(Y_m)), y) \mid \pm a_y \in \delta_m\}$. Next, $update(not\ p(x), \mp a_y, (x, y))$ if $\pm a_y \in \gamma_m$, and
     $update(not\ p(x), \mp a_y, y)$ if $\pm a_y \in \delta_m$.

   After every rule has been processed set $\text{ST}(x, not\ p) = exp$.

2. $\text{ST}(x, not\ p) = exp$ and for some $r \in P_p$, $\text{SG}(x, p, r) \neq 0$, and $\text{NJ}_\text{U}(x, p, r) = S$ with $|S| < |succ_T(x)|$, 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 $\text{SG}(x, p, r) = m \neq 0$ and for every $y \in succ_T(x)$ which has not been yet considered previously, repeat the operations in (†) as above.

In general, justifying a negative unary literal $not\ q \in \mathrm{CT}(x)$ (or in other words, the absence of $q(x)$ in the constructed model) 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 $\pm a \in \mathrm{CT}(x)$) 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).

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 $\mathrm{NJ_U}(x, q, r)$ 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.

The expansion of such a unary negative literal might be done in several steps: in case the literal is not locally justified for some rule which defines it, new successors might be introduced for the current node after the first expansion of the negative literal, and then, the literal has to be justified in these new successors as well. This case is treated in the second part of the expansion rule.

Consider the completion structure described in Figure 5. A possible way to expand $not\ backSucc(\varepsilon)$ is depicted in Figure 6. In this case the absence of $backSucc(\varepsilon)$ from the answer set is justified locally w.r.t. the only rule which defines $backSucc$, which is r_2. This means that $\mathrm{SG}(\varepsilon, backSucc, r_2) = 0$ and $\mathrm{NJ_U}(\varepsilon, backSucc, r_2) = \{1, \varepsilon\}$ (there is only one literal in the local part of r_2: $crash$).

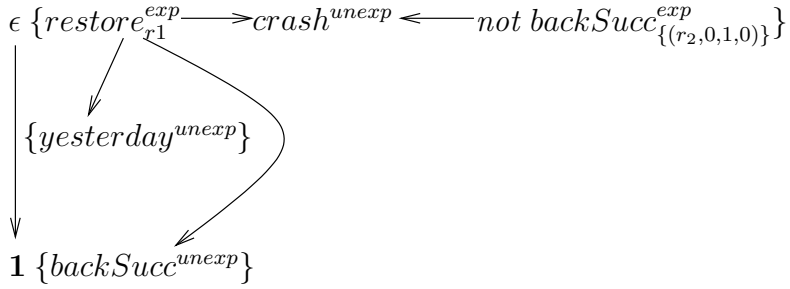

Figure 6: Expansion of a unary negative predicate symbol

We also present an example which demonstrates in an intuitive way the need to refute the same segment in all successors of a node in case the refutation of a unary negative literal in the current node w.r.t. a certain rule is not done locally. Consider a program P which contains the rule $r_1 : a(X) \leftarrow f(X, Y), b(Y), g(X, Z), b(Z)$ and a completion structure for P. Assume the current node is x, x already has three successors $x \cdot 1$, $x \cdot 2$ and $x \cdot 3$, and the predicate to be expanded (justified) is $not\ a$ (Figure 4.1.3).

The only rule which has to be considered (its body has to be refuted) is r_1: this cannot be done locally as the rule has no local part. Thus, the body of the rule has to be refuted in every successor. Figure 8 depicts a situation where the body of the rule has been refuted in a correct way: literals containing the variable Y, thus making part from the first segment of the rule, have been chosen to be refuted in every successor. There is no way to ground r_1 such that all of its body literals are satisfied by choosing as values for Y and Z one of x_1, x_2, or x_3.

On the other hand, Figure 9 depicts a situation where the body of the rule has been refuted in an incorrect way: the literals chosen to be refuted make part from different segments of the rule. In the absence of other

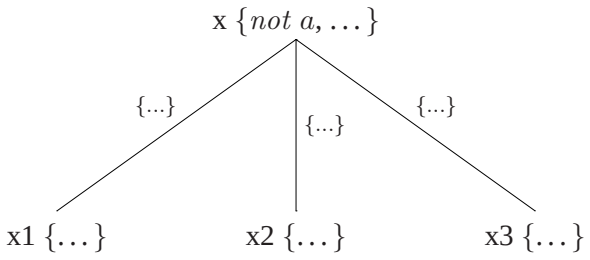

Figure 7: Expanding unary negative: example 2

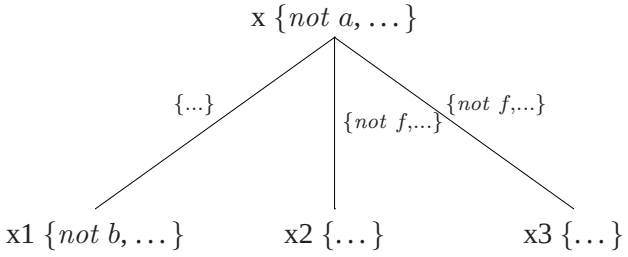

Figure 8: Expanding unary negative: OK

constraints, a situation like the one described in Figure 10 can subsequently appear, in which $a(x)$ is actually justified as the body of the grounded rule $a(x) \leftarrow f(x, x_3), b(x_3), g(x, x_2), b(x_2)$ is satisfied in the current model.

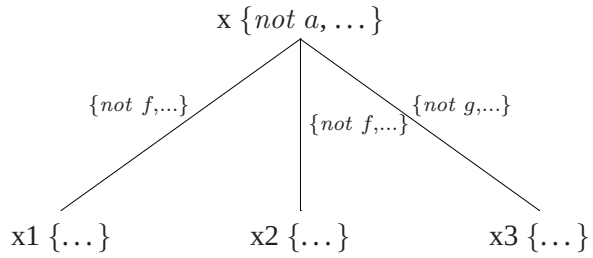

Figure 9: Expanding unary negative: NOT OK (1)

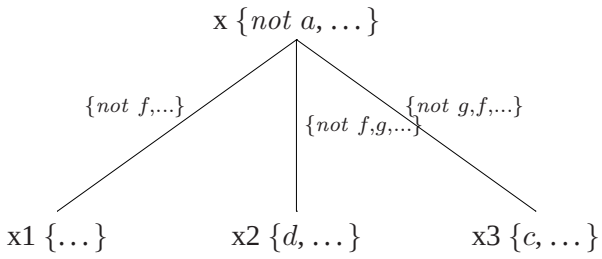

Figure 10: Expanding unary negative: NOT OK (2)

4.1.4 (iv) Expand binary positive.

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

4.1.5 (v) Expand binary negative.

For a binary negative predicate symbol (non-free) $not\ p$ in $\text{CT}(x, y)$ such that $\text{ST}((x, y), not\ p) = unexp$, nondeterministically choose for every rule $r \in P_p$ of the form (2) an s from $varset_r(X)$, $varset_r(X, Y)$ or $varset_r(Y)$ and let $\text{NJ}_\text{B}((x, y), p, r) = s$.

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

Finally, set $\text{ST}((x, y), not\ p) = exp$.

Note that a binary rule is always local in the sense that a binary literal $\pm f(x, y)$ can always be justified using component from x, y, and/or (x, y).

4.1.6 (vi) Choose a binary predicate.

There is an $x \in T$ for which none of $\pm a \in \text{CT}(x)$ can be expanded with rules (i-ii), and for all $(x, y) \in A_T$ none of $\pm f \in \text{CT}(x, y)$ can be expanded with rules (iii-iv), and there is a $(x, y) \in A_T$ and a $p \in bpreds(P)$ such that $p \notin \text{CT}(x, y) \wedge not\ p \notin \text{CT}(x, y)$. Then, add p to $\text{CT}(x, y)$ with $\text{ST}((x, y), p) = unexp$ or add $not\ p$ to $\text{CT}(x, y)$ with $\text{ST}((x, y), not\ p) = unexp$.

The intuition for this rule is similar with the intuition for expansion rule (ii).

4.2 Applicability Rules

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

4.2.1 (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-iii) ,
- 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-vi).

We impose that no expansions can be performed on a node from T until its predecessor is saturated.

4.2.2 (viii) Blocking

We call a node $x \in T$ *blocked* if

- its predecessor is saturated, and
- there is an ancestor y of x, $y < x$, such that $\text{CT}(x) \subset \text{CT}(y)$.

The rule says that if there is an ancestor node whose content includes the content of the current node, the current node can be blocked: intuitively, one can show that provided that the content of the ancestor is justified, the content of the current node can also be justified in a similar way (this is possible due to the fact that every positive literal formed with the ancestor node is justified in a finite number of steps as a consequence of the restriction on the marked dependency graph of a simple CoLP; for more details consult the soundness proof). We call (y, x) a *blocking pair* and say that y *blocks* x; we will also refer to x as a blocked node and to y as the blocking node for a blocking pair (y, x). We impose that no expansions (i-vi) can be performed on a blocked node from T.

4.2.3 (ix) Caching

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

- its predecessor is saturated,
- there is a node y which is not an ancestor of x, $y < x$, such that $\mathrm{CT}(x) \subset \mathrm{CT}(y)$.

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 x. We call (y, x) a *caching pair* and say that y *caches* x; we will also refer to x as a cached node and to y as the caching node for a caching pair (y, x).

4.3 Termination, Soundness, and Completion

We call a completion structure *contradictory*, if for some $x \in T$ and $a \in upreds(P)$, $\{a, not\ a\} \subseteq \mathrm{CT}(x)$ or for some $(x, y) \in A_T$ and $f \in bpreds(P)$, $\{f, not\ f\} \subseteq \mathrm{CT}(x, y)$. A *complete completion structure* for a simple CoLP P and a $p \in upreds(P)$, 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 $CS = \langle T, G, \mathrm{CT}, \mathrm{ST}, \mathrm{RL}, \mathrm{SG}, \mathrm{NJ_U}, \mathrm{NJ_B} \rangle$ is *clash-free* if (1) CS is not contradictory, (2) T does not contain cyclic nodes, and (3) G does not contain positive 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 4.2 (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 w.r.t. 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 path in the tree will eventually contain two nodes which fulfill the blocking condition, such that no expansion rules can be applied to successor nodes of the blocked 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 4.3 (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.

From a clash-free complete completion structure, we will construct an open interpretation, and show that this interpretation is an open answer set of P that satisfies p.

1. *Construction of open interpretation*. In order to construct the possibly infinite universe and the possibly infinite interpretation induced by a clash-free complete completion structure, we introduce some new notation. Let $blocked(T)$ be the set of blocking pairs from T and $cached(T)$ the set of caching pairs from T. Also, given a set of symbols Σ, define a labeling function t on T (and implicitly a labeled tree t): $t : T \rightarrow \Sigma$ which assigns to every node of T a symbol from Σ in such a way that no symbol is associated to more than one node. With T_{ext} (t_{ext}) we denote the extended tree which will constitute the actual universe for our constructed interpretation.

   In case $blocked(T) = \emptyset$ and $cached(T) = \emptyset$, $t_{ext} = t$. Otherwise, repeat the following an infinite number of times: for every $(x, y) \in blocked(T)$ do $t \cdot_{t(x)} t[y]$ (every blocked node in a blocking pair will be replaced with the subtree in t starting at the blocking node) and for every $(x, y) \in cached(T)$ do $t \cdot_{t(x)} t[y]$ (every node for which a previous justification can be used is replaced with the tree providing that justification). Note that a new node with the same label as the blocked node is created every time the transformation $t \cdot_{t(x)} t[y]$ is applied for a pair $(x, y) \in blocked(T)$, so at the next iteration this node will be subject to the transformation, and so on. The resulting labeled tree will be t_{ext}. Figure 11 depicts a complete clash-free completion which has a blocking pair and a caching pair (the dotted arrow indicates the connection between the cached node and the caching node, while the dashed arrow indicates the connection between the blocked node and the blocking node) together with the extended tree obtained as a result of application of the operations described above.

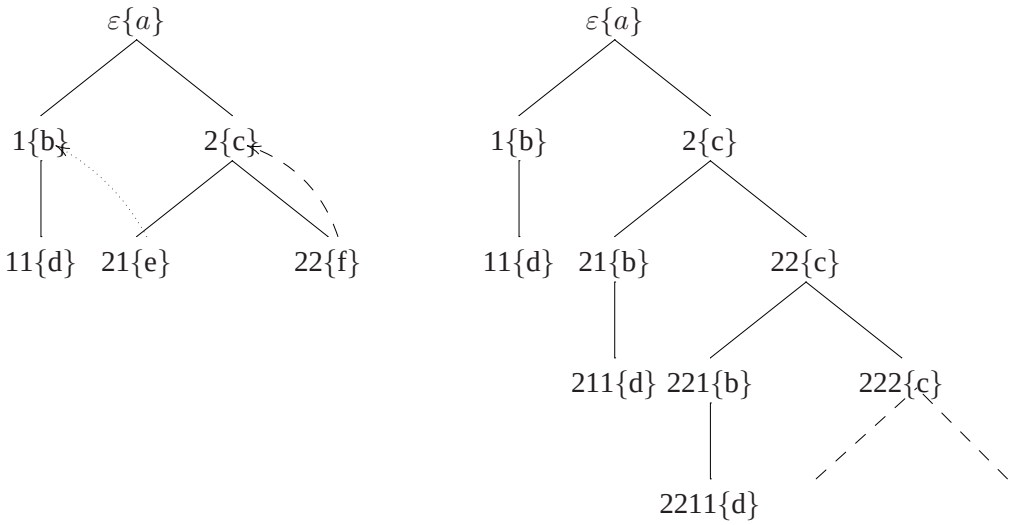

Figure 11: A complete clash-free completion structure with blocked and cached nodes and its corresponding extended tree

We observe that $\forall x \in T_{ext}, \exists! y \in T : t(x) = t(y)$, i.e., for every node in the constructed tree exactly one node in the original tree exists that has the same label; we denote such y for x as $\overline{x}$, and similarly a literal $\overline{l}$ denotes l with each argument $x \in T_{ext}$ replaced by its corresponding $\overline{x}$. Given the way T_{ext} was constructed (by concatenation of subtrees from T), either $x = \overline{x}$ or $x \in T'$, with $T' \subset T_{ext}$ being a version of a subtree of T in T_{ext}, $T[z]$, where z is a blocking node in T, and $\overline{x} \in T[z]$. Being a

version of each other, implies that T' and $T[z]$ have the same tree structure and values for the labeling functions. As $x \in T'$, $\overline{x} \in T[z]$, and $t(x) = t(\overline{x})$, one could say that x is the counterpart of $\overline{x}$ in T', so one can define the values of the labeling functions for x based on the values of the labeling functions of $\overline{x}$. Thus, for every $x \in T_{ext}$, we define:

- $\text{CT}(x) = \text{CT}(\overline{x})$,
- $\forall p \in \text{CT}(x) : \text{RL}(x, p) = \text{RL}(\overline{x}, p)$, and
- $\forall not\ p \in \text{CT}(x), r \in P_p : \text{SG}(x, p, r) = \text{SG}(\overline{x}, p, r)$.

Also note that for x and $\overline{x}$: $|succ(x)| = |succ(\overline{x})|$, and $x \cdot c \in succ(x)$ iff $\overline{x} \cdot c \in succ(\overline{x})$. Thus, we can define NJ_B and NJ_U for nodes in T_{ext}:

- $\forall (x, x \cdot c) \in A_{T_{ext}}, not\ p \in \text{CT}(x, x \cdot c) : \text{NJ}_\text{B}((x, x \cdot c), p, r) = \text{NJ}_\text{B}((\overline{x}, \overline{x} \cdot c), p, r)$, and
- $\forall not\ p \in \text{CT}(x), r \in P_p : \text{NJ}_\text{U}(x, p, r) = \{(x \cdot c, s) \mid (x, x \cdot c) \in A_{T_{ext}}, (\overline{x} \cdot c, s) \in \text{NJ}_\text{U}(\overline{x}, p, r)\}$.

By $G_{ext} = \langle V_{ext}, E_{ext} \rangle$ we denote the graph with nodes $V_{ext} = \{p(x) \mid x \in T_{ext} \cup A_{T_{ext}}, p \in \text{CT}(x)\}$, and edges $E_{ext} = \{(p(x), q(x)) \mid x \in T_{ext}, (p(\overline{x}), q(\overline{x})) \in E\} \cup \{(p(x), q(x \cdot c)) \mid x, x \cdot c \in T_{ext}, (p(\overline{x}), q(\overline{x \cdot c})) \in E\} \cup \{(p(x), q(x, x \cdot c)) \mid x, x \cdot c \in T_{ext}, (p(\overline{x}), q(\overline{x}, \overline{x \cdot c})) \in E\} \cup \{(p(x, x \cdot c), q(x, x \cdot c)) \mid x, x \cdot c \in T_{ext}, (p(\overline{x}, \overline{x \cdot c}), q(\overline{x}, \overline{x \cdot c})) \in E\} \cup \{(p(x, x \cdot c), q(x \cdot c)) \mid x, x \cdot c \in T_{ext}, (p(\overline{x}, \overline{x \cdot c}), q(\overline{x \cdot c})) \in E\} \cup \{(p(x, x \cdot c), q(x)) \mid x, x \cdot c \in T_{ext}, (p(\overline{x}, \overline{x \cdot c}), q(\overline{x})) \in E\}$.

Define the open interpretation (U, M) then as (T_{ext}, V_{ext}), i.e., the universe is the constructed tree T_{ext} and the interpretation corresponds to the nodes in V_{ext}.

2. *M is a model of P_U^M.* All free rules are trivially satisfied.

   Take a ground unary rule: $r' : a(x) \leftarrow \beta^+(x), \bigcup_{1 \le m \le k} {\gamma_m}^+(x, y_m), \bigcup_{1 \le m \le k} {\delta_m}^+(y_m)$ originating from $r : a(X) \leftarrow \beta(X), \bigcup_{1 \le m <\le k} \gamma_m(X, Y_m), \bigcup_{1 \le m \le k} \delta_m(Y_m)$ with $\beta^-(x) \nsubseteq M$ and for all $1 \le m \le k$, ${\gamma_m}^-(x, y_m) \nsubseteq M$ and ${\delta_m}^-(y_m) \nsubseteq M$. Assume $M \models \beta^+(x) \cup \bigcup_{1 \le m \le k} {\gamma_m}^+(x, y_m) \cup \bigcup_{1 \le m \le k} {\delta_m}^+(y_m)$ (together with the assumptions about the negative part of the rule, this amounts to $M \models \beta(x) \cup \bigcup_{1 \le m <\le k} \gamma_m(x, y_m) \cup \bigcup_{1 \le m \le k} \delta_m(y_m)$) and $a(x) \notin M$ (the rule is not satisfied). Then $not\ a \in \text{CT}(x)$, x is saturated, $\text{ST}(x, not\ a) = exp$, and no expansions rules can be further applied to $not\ a$. This implies that for every rule $r \in rules_P(a)$, one of the following holds:

   - $not\ a$ is locally justified in r: $\text{SG}(x, not a, r) = 0$ and $\text{NJ}_\text{U}(x, not\ a, r) = \{(s, x)\}$, where $s \in varset_r(X)$. If $\pm b(X) = lit_r(s)$, $\mp b$ was injected in the content of x in the process of expansion by a call to $update(U, G, not\ a(x), \mp b, x)$ (see expansion rule (iii))[1], which amounts to $\mp b(x) \in M$. This contradicts with $M \models \beta(X)$, as $\pm b(X) \in \beta$.
   - $not\ a$ has been justified in all successors of x: $\text{SG}(x, not\ a, r) = t, 0 < t \le k$ and $\text{NJ}_\text{U}(x, not\ a, r) = S$ and $|S| = |succ(x, T_{ext})|$. It's easy to see that $y_t \in succ(x, T_{ext})$, as $\exists f(x, y_t) \in \gamma(x, y_t)^+$ and $M \models \gamma^+(x, y_t)$. Now, given that $y_t \in succ(x, T_{ext})$ and considering the expansion rule (ii), there must be a tuple $(s, y_t) \in S$ s.t. $s \in varset_r(Y_t)$ or $s \in varset_r(X, Y_t)$. Let $\pm b(Y_t) = lit_r(s)$, if $s \in varset_r(Y_t)$ and $\pm b(X, Y_t) = lit_r(s)$ If $s \in varset_r(X, Y_t)$. Depending on the case $\mp b(y_t)$ or $\mp b(x, y_t)$ was injected in $\text{CT}(x)$ or $\text{CT}(x, y_t)$ (again, according

[1] actually, in case $x \in T_{ext}$ and $x \notin T$, $\text{CT}(x)$ was not generated per se by application of expansion rules, but was defined as a replica of the content of $\overline{(x)}$. But one could see this process of replication as a virtual application of the expansion rules for all the nodes in the extended tree T_{ext}

to the expansion rule (iii)), that is $\mp b(y_t) \in V_{ext}$ or $\mp b(x, y_t) \in V_{ext}$. This contradicts with $M \models \gamma_t(x, y_t)$ as $\pm b(x, y_t) \in \gamma_t(x, y_t)$ or with $M \models \delta_t(y_t)$ as $\pm b(y_t) \in \gamma_t(y_t)$.

The proof for the satisfiability of binary rules is similar.

3. *M is a minimal model of P_U^M.*

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 rules (i) and (iii). 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 set $\{l_1, l_2, \ldots\}$ with $(l_i \in M)_{1 \leq i}$ and $(j, l_i \notin M')_{1 \leq i}$. We observe that $(l_i, l_{i+1})_{1 \leq i} \in E_{ext}$ by construction of E_{ext} (and ultimately by expansion rules (i) or (iii)), so our assumption leads to the existence of a cycle or of a positive path of infinite length in G_{ext}. There cannot be any cycle in G_{ext} as the completion structure we deal with is clash-free, so there must be a positive path of infinite length in G_{ext}. However the following claim contradicts our assumption:

Claim 4.4 There is no positive path of infinite length in G_{ext}.

Assume there is such a positive path of infinite length. Then it has the form $(p_i(x_i))_{1 \leq i}$, where $(p_i \in upreds(P) \cup bpreds(P))_{1 \leq i}$. As there is a finite number of predicate symbols in P, it means that there are two indexes $j, k \in \mathbb{N}^*$, $j \neq k$, such that $p_j = p_k = p$ and $x_j \neq x_k$ (otherwise there will be a cycle in G_{ext}). In other words there is a path from $p(x_j)$ to $p(x_k)$ in G_{ext}. This contradicts with the restriction on simple CoLPs concerning their marked predicate dependency graph, so our initial assumption was false.

The claim is proved, so there is no model $M' \subset M$ of $Q = P_U^M$. Thus, M is minimal.

□

Proposition 4.5 (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.

If p is satisfiable w.r.t. P then p is tree satisfiable w.r.t. P (Proposition 3.2), such that there must be a tree model (U, M) for p w.r.t. P.

We 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. The proof is inspired by completeness proofs in Description Logics for tableaux, for example in [13].

Let $\langle T, G, \text{CT}, \text{ST}, \text{RL}, \text{SG}, \text{NJ}_\text{U}, \text{NJ}_\text{B} \rangle$ be an initial completion structure for p w.r.t. P.

We inductively define a function $\pi : T \cup A_T \rightarrow U$ that relates nodes/edges in the completion structure to nodes in the tree model where $\pi(x, y) = \pi(y)$[2], and satisfying the following properties:

$$\ddagger \begin{cases} \{q \mid q \in \text{CT}(z)\} \subseteq \mathcal{L}(\pi(z)), \text{ for all } z \in T \cup A_T \\ \{q \mid not\ q \in \text{CT}(z)\} \cap \mathcal{L}(\pi(z)) = \emptyset, \text{ for all } z \in T \cup A_T \end{cases}$$

[2]Recall that for a tree model $f(x, y)$, f is stored in the label of y.

Intuitively, the positive content of a node/edge in the completion structure is contained in the label of the corresponding tree model node, and the negative content of a node/edge in the completion structure cannot occur in the label of the corresponding tree model node.

Claim 4.6 Let CS be a completion structure and π a function that satisfies (‡). If an expansion rule is applicable to CS then the rule can be applied such that the resulting completion structure CS' and an extension π' of π still satisfies (‡).

We start by setting $\pi(\varepsilon) = \varepsilon$ (the root of the structure corresponds to the root of the tree model) and $\mathrm{CT}(\varepsilon) = \{p\}$. It is clear that (‡) is satisfied. By induction let CS be a completion structure and π a function that satisfies (‡). We consider the expansion rules and the applicability rules:

1. *Expand unary positive.* As $q \in \mathrm{CT}(x)$, we have, by the induction hypothesis, that $q \in \mathcal{L}(\pi(x))$. Since M is a minimal model there is an $r \in P_q$ of the form (1) and a ground version $r' : q(\pi(x)) \leftarrow body \in (P_q)^M_U$ such that $M \models body$. Set $\mathrm{RL}(q, x) = r$ and $update(q(x), \beta, x)$. Next, for each $\gamma_k(x, z_k), 1 \leq m \leq k$ from r'

   - If $z_k = \pi(x \cdot s)$ for some $x \cdot s$ already in T, take $y_k = x \cdot s$ or if $z_k = x \cdot s$ and z_k is not yet the image of π of some node in T, then add $x \cdot s$ as a new successor of x in T: $T = T \cup \{x \cdot s\}$ and set $\pi(x \cdot s) = x \cdot s$.
   - $update(q(x), \gamma_m, (x, y_k))$,
   - $update(q(x), \delta_m, y_k)$,

   In other words we removed the nondeterminism from the *expand unary rule*, by choosing the rule r and the successors corresponding to the open answer set (U, M). One can verify that (‡) still holds for π.

2. One can deal with the rules (ii-vi) in a similar way, making the nondeterministic choices in accordance with (U, M).

3. *Saturation.* No expansion rule can be performed on a node from T until its predecessor is saturated. This rule is independent of the particular open answer set which guides the construction, so it is applied as usually.

4. *Blocking*. Consider a node $x \in T$ which is currently expanded. If there is a node $y \in T$ such that $y < x$, $\mathrm{CT}(x) \subseteq \mathrm{CT}(y)$, we can stop the expansion as y and x form a blocking pair. Naturally, (‡) still holds for π as we have not modified the content of nodes, but just removed some unexpanded nodes. Note that at this point we no longer use the guidance of (U, M): (U, M) might justify in a different way the predicates grounded in x, but that is not important; we are guaranteed by the soundness result that we have a proof by blocking.

5. *Caching*. Consider a node $x \in T$ which is currently expanded. If there is a saturated node $y \in T$ such that $x \not< y$ and $y \not< x$ and $\mathrm{CT}(x) \subseteq \mathrm{CT}(y)$, we can stop the expansion of x as x and y form a caching pair. Again, (‡) still holds for π as we have not modified the content of nodes, but just removed some unexpanded nodes. And like in the case of blocking, we no longer use the guidance of (U, M): (U, M) might justify in a different way the predicates grounded in x, but that is not important; we are guaranteed by the soundness result that we have a proof for x by reusing the proof for y.

Provided that the process of applying the rules above terminates, starting from a tree-shaped open answer set (U, M) which satisfies p w.r.t. P, we have constructed a complete completion structure for p w.r.t. P which is not contradictory (this can easily be seen from (‡) and the fact that (U, M) is an open answer set). The obtained completion is also cycle-free as in the original open answer set no atom is justified circularly. Even if the open answer set (U, M) is infinite, eventually on every infinite path of the corresponding tree, there must be two nodes with equal content; these nodes will generate a blocking situation in the expansion process described above, so the process does terminate. So, the construction does terminate and its result is a complete clash-free completion structure.

□

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 nodes y where (x, y) is some blocked, or caching pair. There are at most mk such nodes, where k is bound by the amount n of unary predicates q in P and the degrees of the rules P_q and 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 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 nodes, the maximum amount of nodes in CS is $(k + 1)2^n$, 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 3.3).

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 [9] 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][8]. The first two extensions are restricted to reasoning with explicitly named individuals only, while [8] 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 [7]. 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.

6 Conclusions and Outlook

We identified a decidable class of programs, simple CoLPs, and provided a nondeterministic 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 whole fragment of CoLPs, as well as the presence of 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] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *Proc. of ICLP'88*, pages 1070–1080, Cambridge, Massachusetts, 1988.

[7] S. Grimm and P. Hitzler. Reasoning in circumscriptive $\mathcal{ALCO}$. Technical report, FZI at University of Karlsruhe, Germany, September 2007.

[8] S. Grimm and P. Hitzler. Defeasible inference with circumscriptive OWL ontologies. In *Workshop on Advancing Reasoning on the Web: Scalability and Commonsense*, 2008.

[9] 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.

[10] S. Heymans, J. de Bruijn, L. Predoiu, C. Feier, and D. V. Nieuwenborgh. Guarded hybrid knowledge bases. *Theory and Practice of Logic Programming*, 8(3):411–429, 2008.

[11] S. Heymans, D. V. 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.

[12] S. Heymans, D. V. Nieuwenborgh, and D. Vermeir. Open answer set programming with guarded programs. *ACM Transactions on Computational Logic (TOCL)*, 9(4), October 2008.

[13] U. S. I. Horrocks and S. Tobies. Practical reasoning for expressive description logics. In *Proceedings 6th International Conference on Logic for Programming and Automated Reasoning (LPAR'99)*, volume LNAI 1705, pages 161–180. Springer Verlag, 1999.

[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.