A Reasoner for Simple Conceptual Logic Programs[⋆]

Stijn Heymans, Cristina Feier, and Thomas Eiter

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

Abstract. Open Answer Set Programming (OASP) can be seen as a framework to represent tightly integrated combined knowledge bases of ontologies and rules that are not necessarily DL-safe. The framework makes the open-domain assumption and has a rule-based syntax supporting negation under a stable model semantics. Although decidability of different fragments of OASP has been identified, reasoning and effective algorithms remained largely unexplored. In this paper, we describe an algorithm for satisfiability checking of the fragment of *simple Conceptual Logic Programs* and provide a BProlog implementation. To the best of our knowledge, this is the first implementation of a (fragment) of a framework that can tightly integrate ontologies and non-DL-safe rules under an expressive nonmonotonic semantics.

1 Introduction

Integrating Description Logics (DLs) with rules for the Semantic Web has received considerable attention over the past years with tightly-coupled approaches such as *Description Logic Programs* [7, 13][1], *DL-safe rules* [14], *r-hybrid knowledge bases* [16], $\mathcal{DL}$+*log* [15], and *Description Logic Rules* [12], as well as loosely-coupled approaches such as *dl-programs* [4]. In [8], we proposed a tightly-coupled approach to combine knowledge bases using a similar semantics as Rosati's r-hybrid knowledge bases. However, instead of syntactically restricting the rule set to DL-safe[2] rules, we required the rule set to fall into a decidable fragment of *Open Answer Set Programming (OASP)* [11].

OASP is a language that 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,

[⋆] A preliminary version of this work, without the implementation, was presented at the 3rd Int. Workshop on Applications of Logic Programming to the (Semantic) Web and Web Services (ALPSWS 2008) for a limited audience.

[1] Note that even though the approaches of [7] and [13] carry the same name, they are different.

[2] A rule is DL-safe if each variable appears positively in a non-DL atom, where a non-DL atom is an atom that is not formed using a DL concept as predicate.

and a rule-based syntax from the LP side supports nonmonotonic reasoning via *negation as failure*. Decidability of several fragments of OASP was identified by syntactically restricting the shape of logic programs, while carefully safeguarding expressiveness, e.g., *Conceptual Logic Programs (CoLPs)* [9] and *Forest Logic Programs (FoLPs)* [10].

Decidability of combined knowledge bases $KB = \langle \Phi, P \rangle$, where Φ is a theory in a DL $\mathcal{DL}$ and P is a program in a decidable fragment $\mathcal{L}'$ of OASP, can then be shown whenever $\mathcal{DL}$ is reducible to $\mathcal{L}'$. For example, if Φ is a $\mathcal{SHIQ}$ DL theory and P is a CoLP, KB can be translated to a CoLP P' such that satisfiability is preserved. Relying on such OASP fragments, rules can then deduce results about anonymous individuals, in contrast with (weakly) DL-safe rules where deductions are only taking into account the instances in the knowledge base. OASP is thus a suitable alternative integrative formalism for both ontologies and rules. To make it a suitable implementation vehicle though, the lack of effective reasoning procedures for decidable fragments of OASP has to be overcome.

In this paper, we describe a terminating, sound and complete algorithm for satisfiability checking in a fragment of Conceptual Logic Programs, and report on a prototype implementation.

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

- We identify a fragment of Conceptual Logic Programs (CoLPs), called *simple CoLPs*, that are expressive enough to simulate the DL $\mathcal{ALCH}$.
- We define an 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 nondeterministic exponential time.
- We provide a prototypical implementation in BProlog [2]. Note that, to date, this provides the basis for the first implementation of a non-trivial tightly-coupled approach that supports a minimal model semantics and is not depending on (a variant of) DL-safeness.

Detailed proofs and discussion of related work can be found in [5].

2 Preliminaries

We recall the open answer set semantics from [11]. *Constants* $a, b, c, \ldots$, *variables* $X, Y, \ldots$, *terms* $s, t, \ldots$, and *atoms* $p(\boldsymbol{t})$ are defined as usual. A *literal* is an atom $p(\boldsymbol{t})$ or a *naf-atom* *not* $p(\boldsymbol{t})$. For a set α of literals or (possibly negated) predicates, $\alpha^+ = \{l \mid l \in \alpha, l \text{ an atom or a predicate}\}$ and $\alpha^- = \{l \mid \textit{not } l \in \alpha, l \text{ an atom or a predicate}\}$. For a set X of atoms, *not* $X = \{\textit{not } l \mid l \in X\}$. 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 $q(\boldsymbol{X}) \vee \textit{not } q(\boldsymbol{X}) \leftarrow$ for variables $\boldsymbol{X}$; they enable a choice for the inclusion of atoms. We call a predicate q *free* in a program if there is a free rule $q(\boldsymbol{X}) \vee not\ q(\boldsymbol{X}) \leftarrow$ in the program. Atoms, literals, rules, and programs that do not contain variables are *ground*. For a rule or a program χ, let $\mathcal{C}(\chi)$ be the constants in χ, $\mathcal{V}(\chi)$ its variables, and $\mathcal{P}(\chi)$ its predicates with $\mathcal{P}^1(\chi)$ the unary and $\mathcal{P}^2(\chi)$ the binary predicates. A *universe* U for a program P is a non-empty countable set $U \supseteq \mathcal{C}(P)$. We denote by P_U the ground program obtained from P by substituting every variable in P by every possible element 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 $\mathcal{B}_P$ and it is an *answer set* of P if the usual definition holds, see, e.g., [6].

In the following, programs are assumed to be finite; 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. For example, an open answer set of the rules $p \leftarrow not\ q(X)$, $p \leftarrow not\ p$, and $q(a) \leftarrow$ is $(\{a, b\}, \{q(a), p\})$. Note that if the universe U is constrained to $\{a\}$, the set of constants appearing in the rules, the set of rules is inconsistent, i.e., there is no answer set. An n-ary predicate p in P is *satisfiable*, if there is an open answer set (U, M) of P and a $\boldsymbol{x} \in U^n$ such that $p(\boldsymbol{x}) \in M$.

We define *trees* as tuples $T = (N_T, A_T)$, with N_T the set of *nodes* of the T, and A_T the set of edges of T. We denote the root of T with ε. For a node $x \in N_T$, we denote with $succ_T(x)$ the *successors* of x. The *arity* of T is the maximum number of successors any node has in T. For $x, y \in N_T$, $x \leq y$ iff x is an ancestor of y (possibly $x = y$). As usual, $x < y$ if $x \leq y$ and $y \not\leq x$.

3 Simple Conceptual Logic Programs

In [9], *Conceptual Logic Programs (CoLPs)*, were defined as 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.

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 either (i) a* free rule*, (ii) 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 (iii) a binary rule

$$f(X, Y) \leftarrow \beta(X), \gamma(X, Y), \delta(Y) \tag{2}$$

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

Furthermore, for such a set of rules P, let $D(P)$ be the marked predicate dependency graph*: $D(P)$ has as nodes the predicates from P and as edges tuples*

(p, q) if there is either a rule (1) or a rule (2) with a head predicate p and a positive body predicate q. An edge (p, q) is marked, if q is a predicate in some δ_m for rules (1), respectively δ for rules (2). In order for P to be a simple CoLP, $D(P)$ must not contain any cycle that has 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 $f(X, Y)$ for x and its 'successor' Y by a body that only considers literals involving Y and Y. The restriction on $D(P)$ ensures that there is no path from some $p(x)$ to some $p(y)$ in the positive atom dependency graph of P_U, where $p \in \mathcal{P}^1(P)$, and x, y are from an arbitrary universe U. Indeed, observe that any marked cycle in $D(P)$ contains a unary predicate and thus corresponds to a path from some $p(x)$ to some $p(y)$ in the atom dependency graph of P_U. Consider the program P:

$$\begin{aligned} r_1 :\quad & a(X) \leftarrow b(X), f(X, Y), \mathit{not}\ a(Y) \\ r_2 :\quad & b(X) \leftarrow a(X) \\ r_3 :\ & f(X, Y) \leftarrow g(X, Y), b(Y) \end{aligned}$$

The marked dependency graph is depicted in Figure 1.

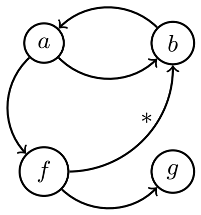


Fig. 1. Marked dependency graph of a CoLP P

There is a single marked edge in $D(P)$, viz. the edge (f, b). Although the cycle (a, b, a) does not contain any marked edge, the cycle (a, f, b, a) contains the marked edge (f, b) and thus P is not a simple CoLP. If we leave r_3 out, the remaining program is a simple *CoLP*. Intuitively, simple CoLPs allow for local recursion (through X) and non-local negative recursion (through Y, see r_1), but not for non-local positive recursion.

As satisfiability checking of CoLPs is EXPTIME-complete [9], checking satisfiability of simple CoLPs is in EXPTIME. Moreover, using a similar simulation of DLs as in [9], one can show that satisfiability checking of $\mathcal{ALCH}$[3] concepts w.r.t. a $\mathcal{ALCH}$ TBox can be reduced to satisfiability checking of a unary predicate w.r.t. a simple CoLP. Thus, satisfiability checking of unary predicates w.r.t. simple CoLPs is EXPTIME-complete.

[3] For a definition of $\mathcal{ALCH}$, we refer the reader to [3].

4 Illustration of the Algorithm

Before formally defining the algorithm for satisfiability checking of a unary predicate p w.r.t a simple CoLP P, we show an example run of the algorithm.

As in tableaux algorithms for Description Logics, the algorithm's basic data structure is a *labeled tree* where the labels are sets of positive and negative predicates. Indeed, the algorithm essentially tries to construct a tableau for the predicate and the program, thus representing an open answer set by a finite structure. Additionally – to take care of minimality – we keep track of the dependencies between atoms by means of a dependency graph.

Expansion rules expand the labeled tree in accordance with the simple CoLP, to construct a partial open answer set, based on the following principles.

1. The occurrence of a positive predicate p in a label has to be motivated by making the body of a rule with head predicate p true in the labeled tree. This principle is similar to the principle of *foundedness* in ordinary Answer Set Programming. We keep track of those dependencies in a positive atom dependency graph that must be acyclic (no atom can motivate itself).
2. The occurrence of a negative predicate *not* p has to be justified by showing that no body of a rule with head predicate p is true in the labeled tree. This ensures *satisfaction* of rules.

Applicability rules constrain the use of expansion rules:

– We can only expand nodes that have a *saturated* parent node, i.e., the parent node has to be fully expanded: it should contain either positive or negative information about all unary predicates in its label and about all binary predicates in its outgoing edges, and no expansion rules are applicable on that parent node.
– Similarly as in DL tableaux, we have a *blocking* rule, that takes care of stopping an expansion on a node x if the label of an ancestor y of x, $y < x$, subsumes the label of x. We thus avoid infinite expansions, and represent a possibly infinite open answer set by a finite structure.
– Additionally, the *caching rule* prohibits to expand a node if the label of a node somewhere else in the tree (thus, not necessarily an ancestor) subsumes the current label. They are not necessary to make the algorithm sound, complete, and terminating, but they make the completion tree smaller.

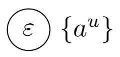
The algorithm succeeds (p is satisfiable w.r.t. to P), if a labeled tree T and a dependency graph G can be built such that the tree does not contain labels with contradiction (for example, *not* p and p in one label) and the dependency graph is not cyclic.

As an example we take the simple CoLP P and check satisfiability of $a \in \mathcal{P}^1(P)$:

$$\begin{aligned}
&r_1 : a(X) \leftarrow f(X, Y_1), b(Y_1), not\ f(X, Y_2), g(X, Y_2), b(Y_2)\\
&r_2 : b(X) \leftarrow f(X, Y), not\ c(Y)\\
&r_3 : c(X) \leftarrow not\ b(X)
\end{aligned}$$

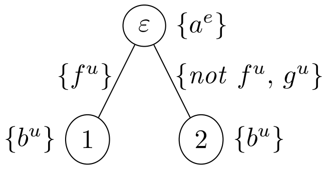
with f and g free.

The initially labeled tree consists of a single node ε with label a:



The dependency graph G contains the corresponding atom $a(\varepsilon)$. Note that we draw the tree with each predicate in the label superscripted with an indication of its expansion status – a^u means a is *u*nexpanded.

Recall that we want to construct a (partial) open answer set. As in ordinary ASP, an atom that is in an (open) answer set has to be *motivated* by a rule, i.e., there has to be a rule with head predicate a that has a true body. The *Expand Unary Positive* rule does exactly this (see (i) in Section 5.1): it selects a rule with head predicate a and expands the labeled tree according to the body of the rule. In the example, the only relevant rule is r_1. To make its body true, we extend the tree with 2 successors (corresponding to Y_1 and Y_2), such that both successors are labeled with b and its edges with f, and *not* f and g respectively:



The predicate a is now expanded; the root has two children, 1 and 2, with the unexpanded b in their label as well as unexpanded binary predicates on the outgoing edges.

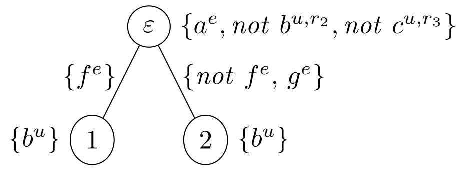
The root node is not saturated yet (see (vii) in Section 5.2), i.e., there are unary predicates in P of which we do not have a positive or negative occurrence in the label, neither have all choices been made for ε's outgoing edges and the binary predicates in P. Moreover, there are predicates in the successors of the root that are not expanded yet. Thus, we cannot start expanding the root's children.

Note that all binary positive predicates in the outgoing edges from ε are free such that they are trivially minimally motivated (we do not need to make any bodies true to motivate the presence of either f or g). Remains the negative predicate *not* f on the edge $(\varepsilon, 2)$. In order to justify the presence of *not* f, one has to negate the bodies of *all* binary rules that have f as head predicate. As there are no such rules, *not* f can be set to expanded and considered justified.

The root is still not saturated after the above operations (it does not contain all positive or negative versions of the unary predicates). In order to mend this, we *choose a unary predicate* (see (iii) in Section 5.1). The algorithm picks a predicate, b for example, and adds its negation to the content of ε. Note that the algorithm can pick either b or *not* b. Currently, we use the naive heuristics, though, that it is more likely that something is not in a node.

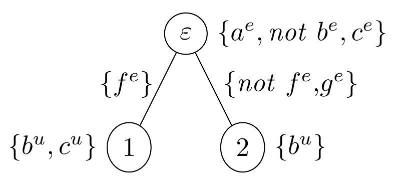
Additionally to its unexpanded superscript u, we keep track of all the rules with head b (in this case only r_2). Recall that we are trying to construct a partial open answer set with as a universe (part of) the tree we are constructing. In the example, the universe is currently $\varepsilon, 1$, and 2. Intuitively, the negative presence of b has to be justified by making sure all bodies, ground with this universe, of rules that have head predicate b are made false. Otherwise, there would be a rule that has a true body and thus forces us to introduce b instead of *not* b. In the example program, the body of r_2 is $f(X,Y), not\ c(Y)$ which becomes true for the current tree, if there is a successor of ε that connects with ε via f and where *not* c holds. Thus, in order to make sure that this body does not become true, we have to enforce that for each successor of ε either it is not connected via f or its label contains c. The example is simplistic: if the body would be $f(X,Y_1), c(Y_1), g(X,Y_2), d(Y_2)$ one would need to show for each 2 successors y_1, y_2 of ε (the node with which X is unified) that f is not present on the outgoing edge to y_1 or that c is not in the label of y_1 or that g is not present on the outgoing edge to y_2 or that d is not in the label of y_2.

Note that in order to justify a negative unary predicate, we need knowledge of all possible successors of a node. We only obtain this knowledge after all positive unary predicates that are or will be appearing in this node have been expanded (recall that positive unary predicates might introduce new successors, as did a). The algorithm thus *tries to complete the node first* with either negative or positive predicates, and only starts expanding negative predicates if all positive ones have been expanded. In the current tree, we are thus still missing a choice for c. By default, we again choose *not* c which has to be justified by r_3.

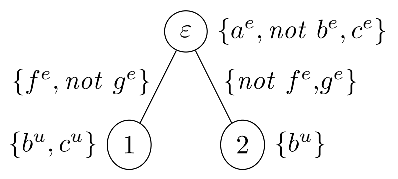


Now, all unary predicates are present in ε and all positive ones (a) are expanded, i.e., with the current label no more successors can be introduced, such that we can start expanding the negative predicates. We choose to justify *not* c in the root ε by making the body of r_3 false, i.e., b has to be in ε. Clearly, this would cause a contradiction, such that we backtrack on the choice for c and include c in ε. Now, there is a positive predicate c that is not yet expanded, such that before expanding *not* b^{u,r_2}, we have to expand c as c might introduce new successors that influence the justification of *not* b. Clearly, c can be motivated, using r_2, as *not* b is present in ε.

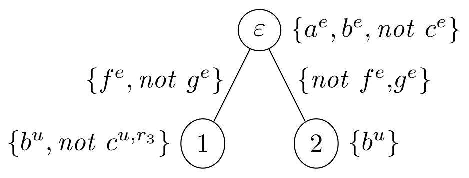
Thus, we can now justify *not* b^{u,r_2}, i.e., for each successor of ε we need that either f is not present in the outgoing edge or c has to be present in the label of that successor (see rule (ii) in Section 5.1). Thus, as *not* f is already in the label of the edge from ε to 2 and f is in the label of the edge from ε to 1 , we only have to add the unexpanded c to 1. We have the following tree:



To finally saturate ε one can see that either g or *not* g is missing in the edge $(\varepsilon, 1)$. The *Choose a Binary Predicate* rule (see (vi) in Section 5.1) adds *not* g.



Now, one can see that to expand c^u in node 1 one needs *not* b in 1, by r_3. However, b^u is already present. The algorithm backtracks, i.e., removes *not* g^e again, removes c^e again, up until the wrong choice of the *Choose a unary predicate* rule, to introduce b^u instead of *not* b^{u,r_2}.[4] Expanding this b^u using rule r_2 introduces *not* c^u in node 1, and choosing *not* c in ε and 1 and *not* g in the edge $(\varepsilon, 1)$, leads to


ε $\{a^e, b^e, not\ c^e\}$
$\{f^e, not\ g^e\}$
$\{not\ f^e, g^e\}$
$\{b^u, not\ c^{u,r_3}\}$ 1
2 $\{b^u\}$

Now that ε is saturated, one can consider node 1. One sees that the label of 1 is a subset of the label of ε (not taking into account the expansion status), and, intuitively, one can use the expansions used on ε to further expand 1 similarly. This technique is called *blocking* and is similar to the blocking used in DL tableaux methods: node 1 is blocked by ε.

Due to the condition on simple CoLPs that $D(P)$ does not contain cycles with marked edges, one can construct an open answer set by rolling out at 1 the subtree that resides at ε. Similarly, one can see that node 2 is blocked by ε as well.

The constructed dependency graph G is

$$\{(a(\varepsilon), f(\varepsilon, 1)), (a(\varepsilon), b(1)), (a(\varepsilon), f(\varepsilon, 2)), (a(\varepsilon), b(2)), (b(\varepsilon), f(\varepsilon, 1))\}$$

i.e., the graph that keeps track of the positive dependencies. We thus have constructed a labeled tree where no rules are applicable anymore, that does not

[4] Note that if we would have picked *not* c before *not* b above, we could have avoided this backtracking. It is future work to investigate how to optimize backtracking.

contain contradictions, and where the dependency graph is acyclic. The algorithm concludes that a is satisfiable w.r.t. P. This is indeed correct.

If we replace in P the rule r_1 with

$$\begin{aligned} r'_1 : a(X) \leftarrow\ & a(X), f(X, Y_1), b(Y_1), \\ & \mathit{not}\ f(X, Y_2), g(X, Y_2), b(Y_2) \end{aligned}$$

we end up with the same labeled tree; however, the dependency graph G' is $G \cup \{(a(\varepsilon), a(\varepsilon))\}$ and thus cyclic. One can check that a is indeed not satisfiable w.r.t. P, as no open answer set exists that contains some $a(x)$.

Note that we make extensive use of the *choose* rules and the concept of *saturation* to complete the labels of nodes such that they contain either the negative or positive versions of all unary predicates in the program (and similarly for binary predicates in the edges). Assume we would not do such a completion, i.e., we would drive the expansion of the nodes purely on what is entailed by the predicate p to satisfy and would not choose predicates that we apparently do not need to satisfy p.

Such a driven computation would not guarantee the global satisfaction of the program. Instead, it explores a partial solution pattern that would be able to satisfy a predicate. To make sure that this partial solution pattern can be extended to an open answer set, we have to complete it repeatedly. For example, if in the example program, we would add a rule $d(X) \leftarrow \mathit{not}\ d(X)$ the program would not have any open answer sets. However, an expansion that starts with a and does not make choices for the "non-relevant" predicate d would wrongly succeed. Thus, we need to make a choice for d or *not* d in every node. Once we do, one would not be able to construct a labeled tree without contradictions.

In tableaux methods for DLs, one does not have this problem. The TBox (the program in our setting) is usually internalized and the satisfiability of the resulting concept is checked. There is no need to make extensive nondeterministic choices for each concept name in this concept expression.

5 Algorithm

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 refer to the same symbol (either p or *not* p). The negation of $\pm p$ (in a given context) is $\mp p$, that is, $\mp p = \mathit{not}\ p$ if $\pm p = p$ and $\mp p = p$ if $\pm p = \mathit{not}\ p$.

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

Definition 2. *A completion structure for a simple CoLP P is a tuple $\langle T, G,$ $\textsc{ct}, \textsc{st}\rangle$ where $T = (N_T, A_T)$ is a tree, $G = \langle V, A\rangle$ is a directed graph with nodes $V \subseteq \mathcal{B}_{P_{N_T}}$ and edges $A \subseteq V \times V$, and $\textsc{ct} : N_T \cup A_T \to 2^{\mathcal{P}(P) \cup \mathit{not}(\mathcal{P}(P))}$ and $\textsc{st} : \{(x, \pm q) \mid \pm q \in \textsc{ct}(x), x \in A_T\} \cup \{(x, q) \mid q \in \textsc{ct}(x), x \in N_T\} \cup$*

$\{(x, not\ q, r) \mid not\ q \in \textsc{ct}(x), x \in N_T, r \in P_q\} \rightarrow \{exp, unexp\}$ are labeling functions.

The tree T together with the labeling functions is used to represent/construct a tentative tree-shaped open answer set, where N_T represents the tentative universe. $G = \langle V, A \rangle$ is a directed graph which helps to keep track of dependencies between atoms in the constructed model, where V represents the tentative model (such a structure enables checking of the minimality requirement: no atom should depend on itself). The role of the labeling functions is as follows:

- The *content* function $\textsc{ct}$ 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 $\textsc{ct}(x) \subseteq \mathcal{P}^1(P) \cup not(\mathcal{P}^1(P))$ if $x \in N_T$, and $\textsc{ct}(x) \subseteq \mathcal{P}^2(P) \cup not(\mathcal{P}^2(P))$ if $x \in A_T$. The presence of a predicate symbol p (resp. negated predicate symbol *not* p) in the content of some node/edge x of T indicates that $p(x)$ is part (resp. not part) of the tentative model represented by T.
- The *status* function $\textsc{st}$ attaches to every (possibly negated) predicate which appears in the content of an edge x and every positive predicate in the content of a node x a status value which indicates whether the predicate has already been expanded in that node/edge. As indicated in Section 4, for negative predicates in nodes, we additionally keep track of the rule that justifies the negative occurrence.

The algorithm starts with defining an *initial completion structure* which basically captures the constraint that p, the predicate checked to be satisfiable is in the content of some node x, or in other words $p(x)$ is in the open answer set for some individual x.

Definition 3. *An* initial completion structure *given a unary predicate p and a simple CoLP P is a completion structure $\langle T, G, \textsc{ct}, \textsc{st} \rangle$ with $T = (N_T, A_T)$, $N_T = \{\varepsilon\}$, $A_T = \emptyset$, $G = \langle V, A \rangle$, $V = \{p(\varepsilon)\}$, $A = \emptyset$, $\textsc{ct}(\varepsilon) = \{p\}$, and $\textsc{st}(\varepsilon, p) = unexp$.*

Next, we show how to evolve by means of *expansion rules* an initial completion structure of p and P to an expanded 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 to to apply these expansion rules. Note that when multiple expansion rules can be applied, one is chosen non-deterministically.

5.1 Expansion Rules

The expansion rules update the completion structure by making explicit what is needed for justifying the presence or absence of a certain atom in the partial model represented by the current completion. We first define a recurring operation in the expansion rules which describes the necessary updates in the completion structure whenever justifying a literal l in the current model imposes the presence of a new literal $\pm p(z)$ in the model. In such a case $\pm p$ is inserted in the content of z if it is not already there and marked as unexpanded, and in case $\pm p(z)$ is an atom, it should be a node in G. Moreover, if l is also an atom, a new edge from l to $\pm p(z)$ should be created to indicate the dependency of l on $\pm p(z)$ in the model. 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 $V = V \cup \{\pm p(z)\}$,
- if $l \in \mathcal{B}_{P_{N_T}}$ and $\pm p = p$, then $A = A \cup \{(l, \pm p(z))\}$.

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, for a completion structure $\langle T, G, \textsc{ct}, \textsc{st}\rangle$, let $x \in N_T$ and $(x, y) \in A_T$ be the node, resp. edge, under consideration.

(i) Expand unary positive. For a unary positive (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). The rule will motivate the presence of $p(x)$ in the tentative open answer set. To this end we continue by enforcing the body of this rule to be true in the constructed completion structure.
- for the β in the body of r, $\mathit{update}(p(x), \beta, x)$,
- nondeterministically pick up (or define when needed) k successors for x, $(y_m)_{1 \leq m \leq k}$, such that for every $1 \leq m \leq k$: $y_m \in \mathit{succ}_T(x)$ or y_m is a new successor of x and T is updated: $N_T = N_T \cup \{y_m\}$, $A_T = A_T \cup \{x, y_m\}$,
- for every successor y_m of x, $1 \leq m \leq k$: $\mathit{update}(p(x), \gamma_m, (x, y_m))$ and $\mathit{update}(p(x), \delta_m, y_m)$,
- set $\textsc{st}(x, p) = \mathit{exp}$.

(ii) Expand unary negative. Justifying a negative unary predicate $\mathit{not}\ p \in \textsc{ct}(x)$ (the absence of $p(x)$ in the constructed model) means refuting the body of every ground rule which defines $p(x)$ (a body that is true in the constructed model would otherwise enforce the presence of $p(x)$, a contradiction with the fact that $\mathit{not}\ p \in \textsc{ct}(x)$). Formally, for a unary negative $\mathit{not}\ p \in \textsc{ct}(x)$ and a rule $r \in P_p$ of the form (1) and $\textsc{st}(x, \mathit{not}\ p, r) = \mathit{unexp}$ do one of:

- choose some $\pm q \in \beta$, $\mathit{update}(\mathit{not}\ p(x), \mp q, x)$, and set $\textsc{st}(x, \mathit{not}\ p, r) = \mathit{exp}$, or
- if for all $p \in \mathcal{P}^1(P)$, $p \in \textsc{ct}(x)$ or $\mathit{not}\ p \in \textsc{ct}(x)$, and for all $p \in \textsc{ct}(x)$, $\textsc{st}(x, p) = \mathit{exp}$, then for all $y_{i_1}, \ldots, y_{i_k}$ such that $(1 \leq i_j \leq n)_{1 \leq j \leq k}$, where $\mathit{succ}_T(x) = \{y_1, \ldots y_n\}$, do one of the following:
  - for some m, $1 \leq m \leq k$, pick up a binary (possibly negated) predicate symbol $\pm f$ from γ_m and $\mathit{update}(\mathit{not}\ p(x), \mp f, (x, y_{i_m}))$, or
  - for some m, $1 \leq m \leq k$, pick up a unary negated predicate symbol $\mathit{not}\ q$ from δ_m and $\mathit{update}(\mathit{not}\ p(x), q, y_{i_m})$.

Set $\text{ST}(x, \textit{not } p, r) = \textit{exp}$.

One can see that once the body of a ground version of a unary rule $r \in P_p$, for which the head term X is substituted with the current node x, is locally refuted, the bodies of all ground versions of this rule, for which X is substituted with x, are locally refuted, too. For the other refutation case, all possible groundings of a rule have to be considered and this is not possible until all successors of x are known. This is the case when all positive predicates in the content of the current node have been expanded and no positive predicate will be further inserted in $\text{CT}(x)$. If this condition is met, an iteration over all possible groundings of the rule r is triggered. For every possible grounding, one of the body literals from the non-local part of the rule (γs or δs) has to refuted.
(iii) Choose a unary predicate. If for all $q \in \text{CT}(x)$, $\text{ST}(x, q) = \textit{exp}$, and for all $(x, y) \in A_T$, and for all $\pm f \in \text{CT}(x, y)$, $\text{ST}((x, y), \pm f) = \textit{exp}$, and there is a $p \in \mathcal{P}^1(()P)$ such that $p \notin \text{CT}(x)$, and $\textit{not } p \notin \text{CT}(x)$, then either add p to $\text{CT}(x)$ with $\text{ST}(x, p) = \textit{unexp}$, or add *not* p to $\text{CT}(x)$ with $\text{ST}(x, \textit{not } p, r) = \textit{unexp}$, for every rule $r \in P_p$.

In other words, if there is a node x for which all positive predicates in its content and all predicates in the contents of its outgoing edges have been expanded, but there are still unary predicates p which do not appear in $\text{CT}(x)$, one has to pick such a p and inject either p or *not* p in $\text{CT}(x)$. This is needed for consistency: it does not suffice to find a justification for the predicate to satisfy, but one also has to show that this justification is part of an actual open answer set, which is done by effectively constructing it (cf. end of section 4). We do not impose that all negative predicate symbols are expanded as that would constrain all the ensuing literals to be locally refuted.

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

5.2 Applicability Rules

The applicability rules restrict the use of the expansion rules.
(vii) Saturation. A node $x \in N_T$ is *saturated*, if for all $p \in \mathcal{P}^1(P)$, $p \in \text{CT}(x)$ or *not* $p \in \text{CT}(x)$, and no $\pm q \in \text{CT}(x)$ can be expanded with rules (i-iii), and for all $(x, y) \in A_T$ and $f \in \mathcal{P}^2(()P)$, $f \in \text{CT}(x, y)$ or *not* $f \in \text{CT}(x, y)$, and no $\pm f \in \text{CT}(x, y)$ can be expanded with (iv-vi). No expansions should be performed on a node from T until its predecessor is saturated.
(viii) Blocking. A node $x \in N_T$ is *blocked*, if its predecessor is saturated and there is an ancestor y of x, $y < x$, s.t. $\text{CT}(x) \subseteq \text{CT}(y)$.

No expansions can be performed on a blocked node. Intuitively, if there is an ancestor y of x whose content includes the content of x one can reuse the justification for y when dealing with x.
(ix) Caching. A node $x \in N_T$ is *cached*, if its predecessor is saturated and there is a non-cached node $y \in N_T$ such that $y \nleq x$, $x \nleq y$, and $\text{CT}(x) \subseteq \text{CT}(y)$.

No expansions can be performed on a cached node. Intuitively, x is not further expanded, as one can reuse the (cached) justification for y when dealing with x.

5.3 Termination, Soundness, and Completeness

A completion structure is *contradictory* if either (i) for some $x \in N_T$ and $a \in \mathcal{P}^1(P)$, $\{a, not\ a\} \subseteq \textsc{ct}(x)$ or (ii) for some $(x, y) \in A_T$ and $f \in \mathcal{P}^2(P)$, $\{f, not\ f\} \subseteq \textsc{ct}(x, y)$. An *expanded completion structure* for a simple CoLP P and $p \in \mathcal{P}^1(P)$, 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, s.t. no expansion rules can be further applied. An expanded completion structure $CS = \langle T, G, \textsc{ct}, \textsc{st}\rangle$ is *clash-free* if: (1) CS is not contradictory, (2) G does not contain cycles.

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

Theorem 1. *Let P be simple CoLP and $p \in \mathcal{P}^1(P)$. Then, (1) one can construct a finite expanded 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, and (2) there exists a clash-free expanded completion structure for p w.r.t. P iff p is satisfiable w.r.t. P.*

The OASP-R system implements the above algorithm in BProlog [2]. The source code for the program together with some example input programs is available at http://www.kr.tuwien.ac.at/staff/heymans/priv/oasp-r/.

The implementation is a straightforward translation of the algorithm into BProlog, using BProlog's backtracking mechanism to take care of the nondeterministic choices in our algorithm. We chose a Prolog engine for its fast prototype capabilities and BProlog in particular for it being one of the fastest performing Prolog engines currently available.[5]

6 Complexity Results

Let $CS = \langle T, G, \textsc{ct}, \textsc{st}\rangle$ be a completion structure and let CS' be the completion structure from CS by removing from N_T all blocked and cached nodes y. There are at most $k \times l$ such nodes, where k is bound by $|\mathcal{P}^1(P)|$ and the number of non-empty γ_m (resp. γ) of rules of the form (1) (resp. (2)) and l is the number of nodes in CS'. If CS' has more than 2^n nodes, then there must be two nodes $x \neq y$ such that $\textsc{ct}(x) = \textsc{ct}(y)$; if $x < y$ or $y < x$, either x or y is blocked, which contradicts the construction of CS'. If $x \not< y$ and $y \not< x$, x or y is cached, again a contradiction. Thus, CS' contains at most 2^n nodes, so $l \leq 2^n$. Since CS' resulted from CS by removing at most $k \times l$ nodes, the number of nodes in CS is at most $(k + 1)2^n$, and the algorithm has to visit a number of nodes that is exponential in the size of P. At each visit, executing

[5] http://probp.com/performance.htm

an expansion rule or checking an applicability rule can be done in exponential time. 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, we obtain the following result: the algorithm runs in nondeterministic exponential time, a nondeterministic variant of the worst-case complexity characterization. Note that such an increase in complexity is expected. For example, although satisfiability checking in $\mathcal{SHIQ}$ is EXPTIME-complete, practical algorithms run in double nondeterministic exponential time [17].

6.1 Experimental Evaluation

We investigated the performance of our BProlog implementation on some example programs: a set of rules describing *family* relations and a set of rules describing a *game* environment.[6]

The *family* program contains 64 rules and 88 predicates; the *game* program contains 265 rules and 544 predicates. A run of 1000 satisfiability checks results in an average of 0.131 seconds for the *family* program and 15.919 seconds for the *game* program, where each satisfiability check resulted in a positive answer. Time spent goes significantly up when more rules/predicates are present. This is not surprising as the number of nondeterministic choices increases with the rules/predicates present. In case predicates are not satisfiable, the location of the rules that cause the inconsistency is vital. If the inconsistency arises within the rules high up in the program, satisfiability checking stays under 0.2 seconds for both example programs; if the inconsistency arises within rules low in the program, our reasoner does not return within 300 seconds. This difference in behavior depending on the location of the inconsistency is due to the BProlog backtracking mechanism and the order in which it solves goals.

Note that adding more rules can actually lead to better results in OASP-R. For example, using the rules from *game+* which extends *game* by adding rules in the beginning, one gets an average of 13.686 seconds per satisfiability check, i.e., 2 seconds better than without those extra rules.

7 Outlook

We intend to investigate several optimizations of the algorithm originating from both the DL tableaux as well as ASP reasoning algorithms. For example, dependency-directed backtracking will allow to backtrack on the choices that caused an inconsistency instead of backtracking on the last choice the BProlog engine made. Similar to DL tableaux, we will investigate whether we can internalize a program to a form that reduces the amount of nondeterminism in the algorithm. A Java implementation will allow us to more flexibly implement optimization strategies.

[6] Experiments were done on a QuadCore Intel(R) Xeon(R) CPU E5450 at 3GHz under Linux (openSUSE 11.0 (X86-64)). All example programs can be found at http://www.kr.tuwien.ac.at/staff/heymans/priv/oasp-r/ and originated from ontologies that accompanied the RacerPro DL reasoner [1].

References

1. RacerPro 1.9.0. Racer Systems GmbH & Co. KG. http://www.racer-systems.com/index.phtml.
2. BProlog 7.1. Afany software. http://www.probp.com/.
3. F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications.* Cambridge University Press, 2003.
4. 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.
5. 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.
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. 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 Conference (WWW)*, pages 48–57. ACM, 2003.
8. 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.
9. 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.
10. S. Heymans, D. Van Nieuwenborgh, and D. Vermeir. Open answer set programming for the semantic web. *Journal of Applied Logic*, 5(1):144–169, 2007.
11. 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.
12. 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.
13. T. Lukasiewicz. A novel combination of answer set programming with description logics for the semantic web. In *Proc. of ESWC 2007*, pages 348–398, 2007.
14. B. Motik, U. Sattler, and R. Studer. Query answering for OWL-DL with rules. *Journal of Web Semantics*, 3(1):41–60, July 2005.
15. R. Rosati. DL+log: Tight integration of description logics and disjunctive datalog. In *Proc. KR*, pages 68–78, 2006.
16. Riccardo Rosati. On the decidability and complexity of integrating ontologies and rules. *Journal of Web Semantics*, 3(1):61–73, 2005.
17. S. Tobies. *Complexity Results and Practical Algorithms for Logics in Knowledge Representation.* PhD thesis, 2001.

This work is partially supported by the Austrian Science Fund (FWF) under the projects P20305 and P20840, and by the European Commission under the project OntoRule (IST-2009-231875). We would like to thank Uwe Keller for his valuable comments.