`deolingo`: extending Answer Set Programming with Deontic Reasoning

Pedro Cabalar

University of A Coruña (Spain)
cabalar@udc.es

Ovidio M. Moar

University of A Coruña (Spain)
ovidio.manteiga@udc.es

Abstract

This work introduces `deolingo`, a system for deontic reasoning that extends the knowledge representation paradigm of Answer Set Programming (ASP) with deontic operators such as obligations, prohibitions, permissions and other derived connectives. The semantics of `deolingo` is based on *Deontic Equilibrium Logic with eXplicit negation* (DELX), a recently proposed formalism that was shown to provide a simple representational solution to many of the main deontic reasoning challenges in the literature, like the use of defeasible obligations, the distinction between explicit and implicit permissions, a proper treatment of dilemmas versus contrary-to-duty (CTD), or the formalisation of deontic and factual detachment, to name the most relevant ones. In the paper, we describe the basic syntax of `deolingo`, *deontic logic programs*, and explain how these programs are translated into regular ASP, using the ASP solver `clingo` as a backend. Apart from producing the models (answer sets) generated by `clingo` containing the derived obligations, `deolingo` can also provide explanations in terms of derivation trees using natural language statements. To this aim, `deolingo` has been extended with rule annotations that are interpreted using the ASP explanation tool `xclingo`. Finally, we include some running examples to illustrate the use of explainable deontic reasoning.

Keywords: Deontic Logic, Answer Set Programming, Equilibrium Logic, Normative Reasoning, Deontic Equilibrium Logic with eXplicit negation, Explainable Deontic Logic.

1 Introduction

One of the important needs in the field of Deontic Reasoning is the availability of automated reasoning tools that allow, not only to test the logical formalisation of common scenarios (or so-called "deontic paradoxes") from the literature in a systematic way on a computer, but also to incorporate normative reasoning to existing tools for practical knowledge representation and problem solving. In this work, we focus on adding deontic reasoning to the successful knowledge representation paradigm of *Answer Set Programming* (ASP) [19,17,3]. The uses and applications of ASP [8] have been continuously growing in the last decade and cover many diverse areas such as product configuration, work force and resource management, phone call routing, train scheduling, data integration, planning and robotics, music composition, video game scenario generation or spacecraft diagnosis, to name a few. Potentially, any of these scenarios could be subject to the incorporation of normative reasoning. To illustrate this idea, just consider any of the recent ASP applications to scheduling problems in digital health [7] (such as assignment of pre-surgical exams, periodic treatments, staff turns, room scheduling, etc) and how convenient would be the addition of a normative reasoning layer incorporating multiple regulations (health, data protection, risk management, etc), without losing the capabilities of ASP for combinatorial problem solving. Although there exist recent implementations of formalisms for normative reasoning that use ASP tools as a backend [15,13], they do not constitute an extension of ASP themselves, but use substantially different input languages: the policy language from [12] or the use of *Defeasible Deontic Logic* [14], respectively. Another interesting approach for automated normative reasoning is Kowalski's *Logical English* [16], that uses Prolog instead of ASP as a backend. However, none of these cases allows reusing current practical systems implemented in ASP to be extended with normative reasoning.

In this paper, we introduce a system, called `deolingo`, that incorporates deontic reasoning to the tool `clingo` [9], a popular and efficient solver implementing the successful knowledge representation paradigm of *Answer Set Programming* (ASP) [19,17,3]. The theoretical foundation for this extension relies on the recently introduced formalism of *Deontic Equilibrium Logic with eXplicit negation* (DELX) [4]. DELX is based on *Equilibrium Logic* [20] with explicit negation [1], a well-established, full logical characterisation of ASP that extends its semantics to any arbitrary propositional formula and further allows two types of negation: default and explicit. DELX further adds two operators, $\mathbf{O}\varphi$ and $\mathbf{F}\varphi$ (respectively read as "obligatory" and "forbidden"), whose semantics extends the three-valued setting already existing for explicit negation in Equilibrium Logic. One of the salient features of this formalism is that it deals with a weakened version of the **D** axiom of Standard Deontic Logic (SDL) [23], which can be stated as $\neg(\mathbf{O}p \wedge \mathbf{F}p)$, namely, the obligation and prohibition of a same fact cannot be simultaneously true. In DELX, this constraint is only imposed when no information about p or its negation is available (which would correspond to a dilemma) but consistency is restored if any of the two obligations is violated, that is, if either we do p violating the prohibition, or $\neg p$ violating the obligation. Cabalar et al. [4] illustrated how DELX can be used to solve many of the deontic reasoning challenges in the literature, like the use of defeasible obligations, the distinction between explicit and implicit permissions, a proper treatment of dilemmas versus contrary-to-duty (CTD), or the formalisation of deontic and factual detachment, to name the most relevant ones. Finally, Cabalar et al. also proved that any DELX theory can be reduced to a form called *deontic logic program* that has a direct translation to ASP, something we exploit now in the current paper for the `deolingo` implementation.

Since `deolingo` extends ASP, it inherits all its reasoning and problem solving capabilities and can be immediately used for various analytical tasks [2] in normative reasoning such as checking *compliance* of a situation with respect to a set of norms, *consistency* inter or intra-regulations, *entailment* of some particular obligation from a set of norms, or *normative* Artificial Intelligence (AI), that is, providing other AI systems with deontic reasoning capabilities. Additionally, we have also exploited the ASP system `xclingo` [5], that enriches `clingo` with the generation of explanations, to allow constructing natural language explanations in the form of proof trees that allow justifying the obligations and prohibitions derived in each model.

The rest of the paper is structured as follows. The next section contains some preliminaries, including a brief description of DELX, plus some basic concepts of ASP and the `clingo` solver. Section 3 describes the implemented tool, including the input language, its use and some implementation details. Section 3.5 further introduces the explainability features and presents an example. Finally, Section 4 concludes the paper.

2 Background

2.1 Deontic Equilibrium Logic with Explicit Negation (DELX)

As happens with Equilibrium Logic, DELX is defined in two steps: we first describe a monotonic logic (an three-valued extension of an intermediate logic) and then define its *equilibrium models* as a model selection criterion, obtaining a non-monotonic formalism in that way. The mononotic basis of DELX, called *Deontic Here-and-There with eXplicit negation* (DHTX) is defined as follows. A *formula* φ of DHTX follows the grammar:

$$\varphi ::= p \in At \mid \bot \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \rightarrow \varphi \mid \ \neg\varphi \mid \mathbf{O}\varphi \mid \mathbf{F}\varphi$$

We define the derived operators $\varphi \leftrightarrow \psi \stackrel{\text{def}}{=} (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$, $not\ \varphi \stackrel{\text{def}}{=} (\varphi \rightarrow \bot)$ and the constant $\top$ as $not\ \bot$. Intuitively, $not\ \varphi$ stands for the default negation of φ (there is no evidence about φ) whereas $\neg\varphi$ represents its stronger, explicit negation (φ can be proved to be false). We sometimes write $\psi \rightarrow \varphi$ reversed as $\varphi \leftarrow \psi$ (to represent rules in logic programs). We also define the derived deontic operators shown in Figure 1. $\mathbf{P}\varphi$ stands for the explicit permission for φ whereas $\mathbf{P^d}\varphi$ is its default version. The superindices above stand for: **d**=*default*, **f**=*fulfilled*, **v**=*violated*, **nv**=*non-violated*, **nf**=*non-fulfilled* and **u**=*undetermined*. We define the same variants $\mathbf{F}^x$ in terms of $\mathbf{O}^x$ for all $x \in \{\mathbf{d}, \mathbf{f}, \mathbf{v}, \mathbf{nv}, \mathbf{nf}, \mathbf{u}\}$. For more details on these operators, see [4].

A theory Γ is defined as a set of formulas. Given a signature or set of atoms At, an *extended atom* is any of the expressions p, $\mathbf{O}p$ or $\mathbf{F}p$ for any $p \in At$. We define an *explicit literal* as any extended atom a or its explicit negation $\neg a$. A set of explicit literals is said to be *consistent* if it contains no pair $\{a, \neg a\}$. A

$$
\begin{array}{rlrl}
\mathbf{O^v}\,\varphi & \overset{\text{def}}{=} \mathbf{O}\varphi \wedge \neg\varphi & \mathbf{O^f}\,\varphi & \overset{\text{def}}{=} \mathbf{O}\varphi \wedge \varphi \\
\mathbf{O^{nf}}\,\varphi & \overset{\text{def}}{=} \mathbf{O}\varphi \wedge \mathit{not}\ \varphi & \mathbf{O^{nv}}\,\varphi & \overset{\text{def}}{=} \mathbf{O}\varphi \wedge \mathit{not}\ \neg\varphi \\
\mathbf{O^u}\,\varphi & \overset{\text{def}}{=} \mathbf{O}\varphi \wedge \mathit{not}\ (\varphi \vee \neg\varphi) & \mathbf{P}\varphi & \overset{\text{def}}{=} \neg\mathbf{F}\varphi \\
\mathbf{O^d}\,\varphi & \overset{\text{def}}{=} (\mathit{not}\ \mathbf{P}\neg\varphi \rightarrow \mathbf{O}\varphi) & \mathbf{P^d}\,\varphi & \overset{\text{def}}{=} (\mathit{not}\ \mathbf{F}\varphi \rightarrow \mathbf{P}\varphi) \\
\mathbf{O}(\varphi \mid \psi) & \overset{\text{def}}{=} (\psi \vee \mathbf{O^{nv}}\,\psi \rightarrow \mathbf{O}\varphi) & \mathbf{F}^x\,\varphi & \overset{\text{def}}{=} \mathbf{O}^x\,\neg\varphi
\end{array}
$$

Fig. 1. Derived deontic operators.

default literal is any explicit literal L or its default negation *not* L. We define a *rule* as an implication of the form

$$H_1 \vee \cdots \vee H_n \leftarrow B_1 \wedge \cdots \wedge B_m \tag{1}$$

with $n, m \geq 0$ where the H_i and B_j are *default literals.* The disjunction in the consequent is called the rule *head* and the conjunction in the antecedent receives the name of rule *body.* A *deontic logic program* is a set of rules.

The semantics of DHTX is as follows. A *Deontic interpretation* T for a signature At is a consistent set of explicit literals satisfying: $\{\mathbf{O}p, \mathbf{F}p\} \subseteq T$ implies $\{p, \neg p\} \cap T \neq \emptyset$, for any $p \in At$. This condition corresponds to the weakening of the **D** axiom from SDL we commented before: obligation $\mathbf{O}p$ and prohibition $\mathbf{F}p$ can only coexist when there is either evidence about p or about $\neg p$ and so, one of the two obligations has been violated.

A *Deontic HT-interpretation* is a pair $\langle H, T\rangle$ of sets of explicit literals s.t. T is a deontic interpretation and $H \subseteq T$. $\langle H, T\rangle$ is *total* when $H = T$. We define the set of *deontic worlds* as $\{r, o, f\}$ respectively standing for *real*, *obligation* and *forbidden.* The complement of a world $w \in \{r, o, f\}$ is defined as $\overline{r} \overset{\text{def}}{=} r$, $\overline{o} \overset{\text{def}}{=} f$ and $\overline{f} \overset{\text{def}}{=} o$. DHTX constitutes a three-valued logic whose satisfaction relation has a dual falsification relation, both described in the following definition.

Definition 2.1 (Satisfaction/falsification) *$M = \langle H, T\rangle$ satisfies* (resp. *falsifies*) *a formula φ at a deontic world $w \in \{r, o, f\}$, written $M, w \models \varphi$ ($M, w =\!\!| \ \varphi$), if the conditions from Fig. 2 hold.*

As usual, when some interpretation $\langle H, T\rangle$ satisfies all the formulas in a theory Γ we write $\langle H, T\rangle \models \Gamma$ and call $\langle H, T\rangle$ a *model* of Γ.

Definition 2.2 ((Deontic) Equilibrium model)
A total DHTX-interpretation $\langle T, T\rangle$ is a (deontic) *equilibrium model* of a theory Γ if $\langle T, T\rangle \models \Gamma$ and there is no $H \subset T$ such that $\langle H, T\rangle \models \Gamma$.

DELX is the logic induced by equilibrium models defined in this way and subsumes standard Equilibrium Logic, and so ASP, for the syntactic fragment without deontic operators. An important result from [4] is the following:

Theorem 2.3 (Theorem 4 from [4]) *Any deontic theory can be reduced to a deontic logic program with the same DHTX models.*

The proof of this theorem provides the transformations required for the

φ	$M, w \models \varphi$ **when**	$M, w =\!\mid \varphi$ **when**
$\top$ ($\bot$)	always (never)	never (always)
$\alpha \wedge \beta$	$M, w \models \alpha$ and $M, w \models \beta$	$M, w =\!\mid \alpha$ or $M, w =\!\mid \beta$
$\alpha \vee \beta$	$M, w \models \alpha$ or $M, w \models \beta$	$M, w =\!\mid \alpha$ and $M, w =\!\mid \beta$
$\alpha \to \beta$	$M', w \not\models \alpha$ or $M', w \models \beta$ for $M' \in \{M, \langle T, T\rangle\}$	$\langle T, T\rangle, w \models \alpha$ and $M, w =\!\mid \beta$
$\neg\alpha$	$M, \overline{w} =\!\mid \alpha$	$M, \overline{w} \models \alpha$
p	$p \in H$ if $w = r$ $\mathbf{O}p \in H$ if $w = o$ $\neg\mathbf{F}p \in H$ if $w = f$	$\neg p \in H$ if $w = r$ $\neg\mathbf{O}p \in H$ if $w = o$ $\mathbf{F}p \in H$ if $w = f$
$\mathbf{O}\alpha$	$M, o \models \alpha$	$M, o =\!\mid \alpha$
$\mathbf{F}\alpha$	$M, f =\!\mid \alpha$	$M, f \models \alpha$

Fig. 2. Conditions for satisfaction and falsification in DHTX.

reduction[1]. In fact, some of these transformations are applied by `deolingo` to unfold the deontic constructs accepted in its input language. The interest of deontic logic programs is that they can be directly encoded in ASP by treating deontic atoms $\mathbf{O}p$ and $\mathbf{F}p$ as regular atoms and adding the constraint:

$$\bot \leftarrow \mathbf{O}p, \mathbf{F}p, \mathit{not}\ p, \mathit{not}\ \neg p \tag{2}$$

corresponding to the already mentioned weakening of the SDL axiom **D**.

2.2 Brief overview of ASP and `clingo`

Answer Set Programming (ASP) [17,19,3] is a declarative problem solving paradigm rooted in non-monotonic logic, logic programming with default negation, and the *stable models* semantics [11]. It provides a powerful framework for solving complex combinatorial search problems by representing knowledge in a logical form (programs consisting of a set of logical rules, facts, and constraints) and computing stable models (answer sets) that correspond to solutions. ASP solvers such as `clingo` [10] are used to compute these answer sets, effectively solving the encoded problem.

In ASP, *propositional programs* are built from atoms (elementary propositions) and rules of the form (1) where, this time, H_i and B_i can only be atoms p or their default negation $\mathit{not}\ p$. In standard ASP syntax, conjunctions in the rule body are represented as commas whereas, in code text files, the left implication $\leftarrow$ is written as '`:-`', and rules are ended by a dot. As an example, a possible propositional ASP program is shown in Listing 1. The first rule states that the light is on if the power is on and we cannot prove that the bulb is broken. The second rule contains a *choice*, that corresponds an abbreviation of the formula $\mathit{broken} \vee \mathit{not}\ \mathit{broken} \leftarrow \mathit{hit}$ and has a non-deterministic effect: if we

[1] For space reasons, we do not include them here but we refer the reader to [4].

hit the bulb, we generate two answer sets, one in which it is broken, and one in which it is not. The third line is a *constraint* that corresponds to a rule with an empty head, or $\bot$, and disregards any potential answer set where both *broken* and *protected* are true. The second rule is a *fact* (corresponding to a rule with empty body, or $\top$) directly stating that the power is on. Finally, the last two lines are *facts* (corresponding to a rules with empty body, or $\top$) directly stating that the power is on and that we hit the bulb. The program has two answer sets, $\{power\_on, hit, light\_on\}$ and $\{power\_on, hit, broken\}$.

```
light_on :- power_on, not broken.
{broken} :- hit.
:- broken, protected.
power_on.
hit.
```

Listing 1. An example of propositional ASP program.

As we explained before, explicit negation ($\neg$) distinguishes between lack of justification (*not* a) and justification for negation ($\neg a$). In program rules, this negation can only appear in front of atoms and, in code text files, it is normally written as `-a`. The use of strong negation is a modelling convenience, reducible to ordinary programs by treating the strong negated atoms as renamed atoms where they cannot appear together in any answer set.

The traditional semantics of ASP [12] based on stable models is defined in terms the so-called program *reduct* but, for the purposes of this paper, it suffices to say that the stable models of an ASP program just correspond to its equilibrium models as in Definition 2.2, since ASP programs constitute a (non-deontic) syntactic fragment of the language defined in Section 2.1.

Propositional programs, however, are not the most common use of ASP. Logic programs usually accept a limited version of a first-order language where atomic formulas are formed with predicates, the use of function symbols is forbidden or, at least, limited and all variables in a rule are universally quantified. Variables are represented with identifiers starting with an upper-case letter. For instance, a possible program with predicates and variables could look like Listing 2 where we have now three lamps $(1, 2, 3)$ connected to the same power line and we hit bulb number 2. Variables in a rule must satisfy a so-called *safety* condition: a variable is *safe* if it occurs in some predicate in the positive body of the rule. This explains why the rule line 4 includes predicate `lamp(X)` in the positive body (otherwise, `X` would not be safe) whereas lines 5 and 6 do not need to specify that `X` is a lamp.

Given a program with variables P, we define *ground*(P) as the ground program that results from replacing the variables by all their possible instantiations formed by elements in the domain. The answer sets of P then simply correspond to the answer sets of *ground*(P) understood as a propositional program. In the case of our example above, for instance, the ground program would correspond to replacing each one of the lines 4 to 6 by three copies replacing `X` respectively

```
lamp(1).
lamp(2).
lamp(3).
light_on(X) :- lamp(X), power_on, not broken(X).
{broken(X)} :- hit(X).
:- broken(X), protected(X).
power_on.
hit(2).
```

Listing 2. An example of ASP program with variables.

by `1`, `2` and `3`. The program has two answer sets, respectively differing in the atoms $\mathit{light\_on}(2)$ and $\mathit{broken}(2)$, whereas they share the rest of atoms $\{\mathit{lamp}(1), \mathit{lamp}(2), \mathit{lamp}(3), \mathit{power\_on}, \mathit{hit}(2), \mathit{light\_on}(1), \mathit{light\_on}(3)\}$.

Current tools for computing with answer set programs support several basic reasoning tasks, which include computing a single answer set, determining that none exist, computing a given number of answer sets, and computing all of them. ASP processing typically works in two stages. First, the predicate program is replaced with an equivalent propositional program by variable replacement (*grounding*). Second, that program is processed by a propositional ASP solver. Most of ASP processing systems make a clear distinction between the two stages and offer separate tools for each, others integrate them. A simple approach to grounding is to replace a program P with *ground(P)*, but generally this is not efficient and multiple optimisation techniques need to be applied to obtain a simpler ground program that has the same answer sets as the original P.

We conclude this section recalling some basic features of `clingo`, especially those related to the `deolingo` implementation. `clingo` is a state-of-the-art ASP solver developed by the Potassco group [10], freely available at the Potassco web page. It integrates both a grounder (gringo) and a solver (clasp) into a single, efficient system, making it a popular choice for solving complex computational problems with a declarative approach. `clingo` can be easily installed as a Python package and run as a command-line application. It also provides a library with an extensible Application Programming Interface (API) that can be imported to build other solvers using it, as is the case of `deolingo`. Version `clingo` 5 introduced the possibility of extending the language with *theory atoms* [9], whose syntax can be flexibly defined using a grammar description (this is called a *theory specification*) and whose semantics can be programmed by the user. A theory atom has is denoted starting with the symbol `&` and followed by an atom name and a list of expressions between curly braces and separated by '`;`'. Alternatively, the theory atom may additionally contain some order or comparison symbol followed by another expression. To put an example, an atom of the form `&sum{2*x; -3*y}>=5` could be used to represent the linear inequation $2 * x - 3 * y \geq 5$, where x and y are external numerical variables whose evaluation is programmed externally. In fact, atoms like the one above are commonly used in extensions of `clingo` (such as `clingcon`, `clingo[DL]` and `clingo[LP]`) that deal with external computation of numerical constraints, interleaved with the ASP solving search process through routines called *propagators*.

In the case of deolingo, however, we do not need using propagators with an external solver, but we exploit instead the use of the *Abstract Syntax Tree* (AST) module inside clingo API that allows exploring the syntactic form of theory atoms. This allows the possibility to introduce syntactic transformations in a program before initiating the solving process or even to remove theory atoms in favour of standard (auxiliary) predicates that encode their meaning, which is the technique actually used by deolingo.

3 The deolingo system

The system deolingo consists of a command-line application and a Python library implemented on top of clingo. Furthermore, a web interface is provided in order to easily try, encode and solve their deontic theories without need of local installation. Both the input and the output formats are adapted to represent the necessary concepts from deontic logic. For the input, a special syntax is used to encode the deontic problems which is provided by clingo theory atoms. On the other hand, the normal output from clingo - with the true atoms in each answer set - is extended by grouping them in the three deontic worlds defined by DELX.

Additionally, deolingo also features explainability by means of xclingo. The deolingo input language is extended by including the xclingo annotations (a kind of formatted comments) that contain natural language descriptions to be shown in the explanations plus directives about what the user considers *relevant* information to be shown in each case. When working in *explainable mode*, deolingo output generates explanations for each (relevant) true atom in the answer set. Those explanations have the form of proof trees whose nodes are textual descriptions of the rules and atoms applied.

3.1 Input language

The input to the deolingo system is a set of text files containing clingo logic programs together with a clingo theory specification, which defines the grammar for the deolingo theory atoms. The complete theory specification can be seen in Appendix A. Without entering into detail about theory specifications in clingo (see [9]), in our case, it first defines a deontic_term (used inside theory atoms) as an expression that can be formed with the Boolean operators `-, &&, ||` that respectively stand for explicit negation $\neg$, conjunction $\wedge$ and disjunction $\vee$. Operator '`|`' is used for conditional obligations of the form $\mathbf{O}(\varphi \mid \psi)$ as defined in Figure 1. The list of the most relevant theory atoms is described in Figure 3, where they are grouped in three main categories. In the figure, we assume that p actually represents any deontic term (not just a proposition). The main atoms in Fig. 3(a), correspond to obligations or their explicit negations (permissions). As a remark, it is worth to note that clingo does not allow explicit negation of a theory atom, so the formula $\neg\mathbf{O}p$ cannot be represented as "`-&obligatory{p}`". This explains the need of a specific theory atom for that purpose we called `&omissible{p}`. The second group in Fig. 3(b) corresponds to violations and fulfillments: these atoms can be easily rephrased in terms of the main atoms, but are sometimes convenient to write more readable rule bodies. Finally, the third group has to do with default obligations. The first four theory atoms in that group have the form of conditionals and can only be used in rule heads in `deolingo`.

Deontic atom	DELX formula	Equivalent to
`&obligatory{p}`	$\mathbf{O}p$	$\mathbf{F}\neg p$, `&forbidden{-p}`
`&forbidden{p}`	$\mathbf{F}p$	$\mathbf{O}\neg p$, `&obligatory{p}`
`&permitted{p}`	$\mathbf{P}p$	$\neg\mathbf{O}\neg p$, `&omissible{-p}`
`&omissible{p}`	$\neg\mathbf{O}p$	$\mathbf{P}\neg p$
`&optional{p}`	$\neg(\mathbf{O}p \vee \mathbf{F}p)$	$\mathbf{P}p \wedge \mathbf{P}\neg p$

(a) Main deontic atoms

Deontic atom	DELX formula	Equivalent to
`&violated_obligation{p}`	$\mathbf{O}p \wedge \neg p$	
`&violated_prohibition{p}`	$\mathbf{F}p \wedge p$	$\mathbf{O}\neg p \wedge p$
`&fulfilled_obligation{p}`	$\mathbf{O}p \wedge p$	
`&fulfilled_prohibition{p}`	$\mathbf{F}p \wedge \neg p$	$\mathbf{O}\neg p \wedge \neg p$
`&non_violated_obligation{p}`	$\mathbf{O}p \wedge \mathit{not}\ \neg p$	
`&non_violated_prohibition{p}`	$\mathbf{F}p \wedge \mathit{not}\ p$	$\mathbf{O}\neg p \wedge \mathit{not}\ p$
`&non_fulfilled_obligation{p}`	$\mathbf{O}p \wedge \mathit{not}\ p$	
`&non_fulfilled_prohibition{p}`	$\mathbf{F}p \wedge \mathit{not}\ \neg p$	$\mathbf{O}\neg p \wedge \mathit{not}\ \neg p$
`&undetermined_obligation{p}`	$\mathbf{O}p \wedge \mathit{not}\ p \wedge \mathit{not}\ \neg p$	
`&undetermined_prohibition{p}`	$\mathbf{F}p \wedge \mathit{not}\ p \wedge \mathit{not}\ \neg p$	$\mathbf{O}\neg p \wedge \mathit{not}\ p \wedge \mathit{not}\ \neg p$

(b) Violation/fulfillment deontic atoms

Deontic atom	DELX formula	Equivalent to
`&default_obligation{p}`	$\mathbf{O^d}\,p$	$\mathit{not}\ \mathbf{P}\neg p \to \mathbf{O}p$
`&default_prohibition{p}`	$\mathbf{F^d}\,p$	$\mathit{not}\ \mathbf{P}p \to \mathbf{F}p$
`&omissible_by_default{p}`	$\mathit{not}\ \mathbf{O}p \to \mathbf{P}\neg p$	$\mathit{not}\ \mathbf{O}p \to \neg\mathbf{O}p$
`&permitted_by_default{p}`	$\mathit{not}\ \mathbf{F}p \to \mathbf{P}p$	$\mathit{not}\ \mathbf{O}\neg p \to \neg\mathbf{O}\neg p$
`&omissible_implicitly{p}`	$\mathit{not}\ \mathbf{O}p$	`not &obligatory{p}`
`&permitted_implicitly{p}`	$\mathit{not}\ \mathbf{F}p$	`not &forbidden{p}`

(c) Implicit/default deontic atoms

Fig. 3. Explanation of `deolingo` theory atoms.

As an example of an input program, take the file included in Listing 3 describing the well-known Prakken and Sergot's *Cottage Fence* scenario [21]. Line 2 states that fences are forbidden by default, namely, unless there exists evidence of an explicit permission. This line is an abbreviation of the also admissible rule:

```
&forbidden{fence} :- not &permitted{fence}.
```

Line 3 is a CTD stating that if the prohibition of is violated and a fence is placed, then the fence must be painted in white. Again, the rule can be seen as an abbreviation, this time of:

`&obligatory{white_fence} :- &forbidden{fence}, fence.`

Line 4 grants a permission for a fence if the cottage is by the sea. Lines 5 and 6 state that a white fence counts as a fence (both in the factual world and inside obligations). Finally, lines 7 and 8 state that the cottage is by the sea and that we put a fence.

```
1 % File cottage.lp: Prakken & Sergot cottage scenario
2 &default_prohibition{fence}.
3 &obligatory{white_fence} :- &violated_prohibition{fence}.
4 &permitted{fence} :- sea.
5 fence :- white_fence.
6 &obligatory{fence} :- &obligatory{white_fence}.
7 sea.
8 fence.
```

Listing 3. An enconding of the Cottage Fence Scenario in deolingo.

As we explained above, theory atoms can be applied to deontic terms, something that provides some (limited) use of complex formulas and combination with other usual ASP constructs (use of ASP variables, pooling, conditional literals, etc). For instance, we can use conjunction in obligations in a rule body as in `&obligatory{p; q(X); a(Y):b(Y)}` being unfolded as the conjunction of `&obligatory{p}; &obligatory{q(X)}; &obligatory{a(Y)}:b(Y).` Similarly, we can use disjunction in obligations in the rule head with an analogous effect. The rules containing deontic theory atoms are processed by deolingo and translated into regular atoms in ASP, using for that purpose their formalisation in DELX. deolingo reduces a deontic program with formulas to a deontic program in normal form before solving it, but not all the complex formulas are supported. Some of these equivalences were not implemented, either due to syntax limitations of clingo, or due to potential performance overhead, or also to follow the established programming conventions. One example of syntax limitation which prevented the implementation of nested formulas (like obligations of obligations), is that clingo syntax does not accept nested theory atoms. Nevertheless, the equivalent normalised formulas can be used to represent the same deontic logic semantics, so in practice, the limitations are just syntactical, but they do not limit the expressive power. For a complete list of complex formulas accepted by deolingo see [18].

3.2 Running deolingo

deolingo can be used as a command-line interface application (CLI) and it supports all the clingo command-line options since it extends the clingo Application class. deolingo can be easily installed[2] in any computer by using

[2] More details on installation at https://github.com/ovidiomanteiga/deolingo.

the Python package manager command `pip install deolingo`. To solve the deontic logic program written in file `cottage.lp`, use the following command: `$ deolingo cottage.lp`. By default, `deolingo` output the first answer set, as `clingo`. To print only a certain number of models, the `clingo` syntax for the command line invocation is supported: `$ deolingo 2 cottage.lp`, where 0 has the special meaning of *all models*. Refer to the GitHub repository to see all the options to run `deolingo`.

The default `deolingo` output shows all the true atoms in each deontic answer set grouped by the three *worlds*, as described in DELX: **facts, obligations, and prohibitions**. In the factual worlds representing facts, in the obligations world representing obligation or its absence (omission), and, in the prohibitions world, representing prohibition or its absence (permission). The `deolingo` output for the file in Listing 3 is shown in Listing 4, where, in this case, the explicit permission disables the default prohibition of a fence.

```
Answer: 1
FACTS: fence, sea
OBLIGATIONS:
PROHIBITIONS: &permitted{fence}
SATISFIABLE
```

Listing 4. Output for deonitc program in Listing 3.

In order to run a deontic logic program using the `deolingo` web editor[3] the input program must be coded in the `deolingo` editor. That can be achieved either by simply typing it in the editor, or by loading it from a local file, or by selecting one example from the drop-down selector with all the examples. Once that is ready, simply select the desired run options (a subset of the ones available in the CLI app) and click the *Solve* button. The program is solved in the backend and, when finished, the deontic answer sets are displayed under the `deolingo` editor.

3.3 Translation to regular ASP

`deolingo` generates a translated program from the original program with theory atoms into another logic program which has prefixed symbolic atoms instead of theory atoms, so that it can be directly grounded and solved by `clingo`. This translated program can be printed by using the command-line argument `--translate` for development and debug purposes. The translated program contains comments with the original source lines of each translated expression. Besides, all the necessary deontic rules are added after the translated sentences from the source program, so that the meaning of the program is preserved. Here is an example of translation of the source program:

```
&obligatory{a} :- b.
```

[3] `deolingo` web editor `https://deolingo.azurewebsites.net/editor`.

To the translated program:

```
% Source line: 1
deolingo_obligatory(a) :- b.
% Deontic weak axiom D for DELX
:- deolingo_obligatory(X), deolingo_forbidden(X), not
    deolingo_holds(X), not deolingo_holds(-X).
% ALL THE DEONTIC RULES HERE ... %
```

The translation contains the transformed first line of the source program; then the weak version of the deontic axiom **D** corresponding to (2); then all the deontic operators and equivalence rules; and finally the truth reification rules for each deontic atom (each atom that is either obligatory or forbidden). For instance, the rules defining the obligation violation operator would look like:

```
% Obligation violation
deolingo_violated_obligation(X) :- deolingo_obligatory(X),
    deolingo_holds(-X).
deolingo_obligatory(X) :- deolingo_violated_obligation(X).
deolingo_holds(-X) :- deolingo_violated_obligation(X).
```

These rules define the operator in both directions of the implication, being the first rule the one that defines that if something is obligatory and false, it means the obligation to do it was violated. The other two rules specify that it must be derived that something is obligatory and false if it is proved that there is a violated obligation to do it.

Also, `deolingo` adds the rules that define the equivalence between deontic atoms. For example, the following rules define the equivalence between obligations and prohibitions:

```
% Obligation/prohibition equivalence
deolingo_obligatory(X) :- deolingo_forbidden(-X).
deolingo_forbidden(X)  :- deolingo_obligatory(-X).
-deolingo_obligatory(X) :- -deolingo_forbidden(-X).
-deolingo_forbidden(X)  :- -deolingo_obligatory(-X).
```

3.4 Implementation details

`deolingo` consists of two main software components developed in Python and bundled as package: the `deolingo` Python library and the Python CLI application. `deolingo` extends the `Application` class from `clingo` API, which handles some input and output, and provides template methods that are useful to run the `deolingo` application. `deolingo` also extends the `clingo Control` class, providing the same interface as the former, while it reuses some existing functionality from it. In the case of `xclingo`, the integration is done with the `xclingo` library by extending the `XClingoControl` class.

`deolingo` is designed to respect the `clingo` interfaces so that other solvers can be build on top, by extending the uniform interface in a similar manner as `deolingo` does with `clingo`. They can extend both the `DeolingoApplication` and the `DeolingoControl` classes to provide the same interface, process the deontic part of their input `clingo` programs and then apply their transformations during the process of parsing, transforming the abstract syntax tree, grounding, solving and printing the output answer sets.

The integration of `deolingo` with `clingo` follows these steps:

1. `deolingo` first uses the `clingo` application to load the input programs and parse them into `clingo` abstract syntax trees (ASTs).
2. Then, it uses custom `clingo` transformers to transform the ASTs such that the complex formulas are broken into the *deontic atom normal form* [4] where we only allow **O** and **F** applied to atoms. After that, deontic theory atoms are transformed into symbolic atoms.
3. `deolingo` adds the weak deontic axiom (2), the deontic equivalence rules and the deontic operator rules.
4. Then, it grounds the translated program by calling the `ground()` method of the `clingo` Control object.
5. `deolingo` calls the `solve()` method of the `clingo` Control object to find solutions.
6. Finally, it transforms the output from `clingo` by grouping the atoms in the three deontic worlds defined by DELX and prints the output.

3.5 Explainability with `xclingo`

We conclude this section describing the `deolingo` *explainable mode* which accepts annotations inherited from the ASP explanation tool `xclingo`. These annotations have the form of formatted comments (lines starting with '`%!`') that can be ignored by `clingo` but are used by `xclingo` to decide which information must be displayed in the explanations. The main types of annotations are: `trace_rule`, that includes some text describing that a rule has been fired; `trace`, that allows describing the meaning of some atom in natural language; and `show_trace` that tells `xclingo` the atoms we want to explain, among all those ones included in the current answer set. Listing 5 shows an annotated version of Listing 3. If we run the command `$ deolingo cottage.lp --explain` on this version, we obtain the following output:

```
*
|__The cottage is by the sea
*
|__There is a fence
*
|__There may be a fence, if the cottage is by the sea; A
    fence is permitted
| |__The cottage is by the sea
```

that tells us that the permission "A fence is permitted" is derived *because* "The cottage is by the sea" and the rule justifying that derivation was "There may be a fence, if the cottage is by the sea".

When reaching deontic contradictions without factual information related, in the normal operation of `deolingo`, the system outputs `UNSATISFIABLE`,

```
%!trace_rule {"There must be no fence, unless a permission
    is granted"}
&default_prohibition{fence}.
%!trace_rule {"It must be a white fence, if the prohibition
    of a fence is violated"}
&obligatory{white_fence} :- &violated_prohibition{fence}.
%!trace_rule {"There may be a fence, if the cottage is by
    the sea"}
&permitted{fence} :- sea.
%!trace_rule {"A white fence is a fence"}
fence :- white_fence.
%!trace_rule {"It is obligatory a fence, if it is obligatory
     a white fence"}
&obligatory{fence} :- &obligatory{white_fence}.
%!show_trace sea.
sea.
%!show_trace fence.
fence.
%!show_trace &permitted{X}.
%!trace {"There is a fence"} fence.
%!trace {"The cottage is by the sea"} sea.
%!trace {"A fence is permitted"} &permitted{fence}.
```

Listing 5. An annotated version of Listing 3.

meaning there are no answer sets that can satisfy the deontic logic program. When working with a small sample program, it is relatively easy to spot the contradictory deontic statements, but in a big normative set of rules, that task can be much more difficult.

deolingo provides a command-line option --weak which can help finding the root cause of these contradictions. When running in that mode, deolingo adds the deontic weak axiom (2) as a weak constraint to the program, instead of a normal constraint. A symbolic atom capturing the inconsistent deontic atom is added to the answer set in case of contradiction and, additionally, that atom is traced in the xclingo explanation.

```
deolingo_inconsistency(X) :- not deolingo_inconsistency(-X),
     obligatory(X), forbidden(X), not holds(X), not holds(-X
    )}.
:~ deolingo_inconsistency(X). [1@1, X]
#show deolingo_inconsistency/1.
%!trace {"INCONSISTENCY: % is obligatory and forbidden
    without factual information!", X} deolingo_inconsistency
    (X).
%!show_trace deolingo_inconsistency(X).
```

Listing 6: Deontic weak axiom **D** as a weak constraint.

4 Conclusions and Future Work

We have presented a deontic reasoning tool, deolingo, that extends ASP with deontic operators allowing normative reasoning. The tool constitutes a logic-programming based implementation of the recent logical formalism of *Deontic Equilibrium Logic with eXplicit negation* (DELX) [4]. `deolingo` provides multiple advantages. On the one hand, it constitutes a proper extension of the ASP solver `clingo`, immediately allowing the incorporation of deontic reasoning on already existing practical applications of ASP. On the other hand, it provides all the representational advantages of DELX for solving deontic reasoning challenges[4] as discussed in [4]. A third important advantage is `deolingo`'s explainability that allows providing a derivation proof, expressed in natural language, justifying all the derived obligations, permissions, violations and fulfillments. This feature is crucial for modern AI systems, especially in those cases related to regulations.

The main topic for immediate future work is providing a temporal extension of `deolingo`, based on the recent extension of DELX to the temporal case [22] and potentially using the temporal ASP tool `telingo` [6] as a backend.

References

[1] Aguado, F., P. Cabalar, J. Fandinno, D. Pearce, G. Pérez and C. Vidal, *Revisiting explicit negation in answer set programming*, Theory and Practice of Logic Programming **19** (2019), pp. 908–924.

[2] Benzmüller, C., X. Parent and L. van der Torre, *A deontic logic reasoning infrastructure*, Computer Science and Communications, University of Luxembourg, Luxembourg (2018).

[3] Brewka, G., T. Eiter and M. Truszczyński, *Answer set programming at a glance*, Communications of the ACM **54** (2011), pp. 92–103.

[4] Cabalar, P., A. Ciabattoni and L. van der Torre, *Deontic equilibrium logic with explicit negation*, in: *Logics in Artificial Intelligence: 18th European Conference, JELIA 2023, Dresden, Germany, September 20–22, 2023, Proceedings* (2023), p. 498–514.

[5] Cabalar, P., J. Fandinno and B. Muñiz, *A system for explainable answer set programming*, Electronic Proceedings in Theoretical Computer Science **325** (2020), p. 124–136.

[6] Cabalar, P., R. Kaminski, P. Morkisch and T. Schaub, *telingo = asp + time*, in: M. Balduccini, Y. Lierler and S. Woltran, editors, *Logic Programming and Nonmonotonic Reasoning* (2019), pp. 256–269.

[7] Cappanera, P., S. Caruso, C. Dodaro, G. Galatà, M. Gavanelli, M. Maratea, C. Marte, M. Mochi, M. Nonato and M. Roma, *Recent answer set programming applications to scheduling problems in digital health*, in: *Proceedings of the Workshop on Experimental evaluation of algorithms for solving problems with combinatorial explosion (RCRA24), CEUR-WS proceedings, vol. 3883*, 2024.

[8] Erdem, E., M. Gelfond and N. Leone, *Applications of answer set programming*, AI Magazine **37** (2016), pp. 53–68.

[9] Gebser, M., R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub and P. Wanko, *Theory solving made easy with clingo 5*, in: *International Conference on Logic Programming*, 2016.

[10] Gebser, M., R. Kaminski, B. Kaufmann and T. Schaub, *Answer set programming*, in: *Handbook of knowledge representation* (2012), pp. 285–340.

[11] Gelfond, M. and V. Lifschitz, *Stable model semantics for logic programming*, Logic programming, Proceedings of the fifth international conference and symposium **2** (1988), pp. 1070–1080.

[4] For an implementation of these challenges in `deolingo`, see [18]

[12] Gelfond, M. and J. Lobo, *Authorization and obligation policies in dynamic systems*, in: M. G. de la Banda and E. Pontelli, editors, *Logic Programming, 24th International Conference, ICLP 2008, Udine, Italy, December 9-13 2008, Proceedings*, Lecture Notes in Computer Science **5366** (2008), pp. 22–36.

[13] Governatori, G., *An ASP implementation of defeasible deontic logic*, Künstliche Intelligenz **38** (2024), pp. 79–88.

[14] Governatori, G., F. Olivieri, A. Rotolo and S. Scannapieco, *Computing strong and weak permissions in defeasible logic*, Journal of Philosophical Logic **42** (2013), pp. 799–829.

[15] Inclezan, D., *An ASP framework for the refinement of authorization and obligation policies*, Theory and Practice of Logic Programming **23** (2023), pp. 832–847.

[16] Kowalski, R., *Logical english for legal applications*, 2020.

[17] Marek, V. and M. Truszczyński, "Stable models and an alternative logic programming paradigm," Springer-Verlag, 1999 pp. 169–181.

[18] Moar, O. M., "Deolingo: a Deontic Logic solver system based on Answer Set Programming," Master's thesis, University of A Coruña, Faculty of Computer Science, A Coruña, Galicia, Spain (2024).
URL `http://hdl.handle.net/2183/41411`

[19] Niemelä, I., *Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm*, Annals of Mathematics and Artificial Intelligence **25** (1999), pp. 241–273.

[20] Pearce, D., *A new logical characterisation of stable models and answer sets*, in: *Non monotonic extensions of logic programming. Proc. NMELP'96. (LNAI 1216)*, Springer-Verlag, 1996 .

[21] Prakken, H. and M. Sergot, *Dyadic deontic logic and contrary-to-duty obligations*, in: D. Nute, editor, *Defeasible Deontic Logic*, Dordrecht, 1997, pp. 223–262.

[22] Soldà, D., P. Cabalar, A. Ciabattoni and E. A. Neufeld, *Tackling temporal deontic challenges with equilibrium logic*, in: *Proceedings of the 24th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2025)*, 2025.

[23] von Wright, G. H., *Deontic logics*, American Philosophical Quarterly **4** (1967), pp. 136–143.

Appendix

A Complete theory specification for deolingo

```
#theory deolingo {
    deontic_term {
        - : 4, unary;
        && : 3, binary, left;
        || : 2, binary, left;
        | : 1, binary, left
    };
    show_term { / : 1, binary, left };
    &obligatory/0 : deontic_term, any;
    &forbidden/0 : deontic_term, any;
    &omissible/0 : deontic_term, any;
    &permitted/0 : deontic_term, any;
    &optional/0 : deontic_term, any;
    &permitted_by_default/0 : deontic_term, any;
    &omissible_by_default/0 : deontic_term, any;
    &holds/0 : deontic_term, any;
    &deontic/0 : deontic_term, any;
    &permitted_implicitly/0 : deontic_term, any;
    &omissible_implicitly/0 : deontic_term, any;
    &violated/0 : deontic_term, any;
    &fulfilled/0 : deontic_term, any;
    &violated_obligation/0 : deontic_term, any;
    &fulfilled_obligation/0 : deontic_term, any;
    &non_violated_obligation/0 : deontic_term, any;
    &non_fulfilled_obligation/0 : deontic_term, any;
    &undetermined_obligation/0 : deontic_term, any;
    &default_obligation/0 : deontic_term, any;
    &violated_prohibition/0 : deontic_term, any;
    &fulfilled_prohibition/0 : deontic_term, any;
    &non_violated_prohibition/0 : deontic_term, any;
    &non_fulfilled_prohibition/0 : deontic_term, any;
    &undetermined_prohibition/0 : deontic_term, any;
    &default_prohibition/0 : deontic_term, any;
    &show/0 : show_term, directive
}.
```

Listing 7: `clingo` theory specification for `deolingo`.