# A new OWLAPI interface for HEX-Programs applied to Explaining Contingencies in Production Planning

**Peter Schüller**[1]

**Abstract.** Description Logics (DLs) and Logic Programs are formalisms for describing knowledge about the world. In industrial settings, DLs are often used to describe factories and inventories, and logic programs could be useful for planning production - if they can be integrated with DLs. We here introduce a variation of the DL Program formalism for integrating description logics with Answer Set Programming (ASP). We a new implementation based on OWLAPI and a use case in Explainable AI for production planning. Our integration is based on the HEX formalism and the Hexlite solver. Different from previous work, we extend the integration interface to permit an arbitrary number of parallel modifications of the OWL ontology to be considered for the computation of a single answer set. This is useful to model changes of the planning domain over time, where each time step is a separate modification of the ontology.

We present a case study towards the EU recommendations on 'Human-centered, trustworthy AI systems', concretely we present a production planning framework that can be challenged by users with alternative scenarios. With our case study we argue that Explainability in the context of planning is not about the rationale behind decisions (those are defined by formal semantics and by the program) but we argue that Explainability should mainly focus on the possibility of human operators to challenge the system and request alternative solutions under alternative assumptions. We provide the following examples to show that Logic Programming requires only very small modifications to provide insights by means of the following what-if questions: (i) which machines could be deactivated without preventing to reach the production goal and (ii) which product could be skipped to reach a reduced production goal in a shorter time limit.

## 1 Introduction

Description Logics (DLs) are useful tools for describing knowledge about objects in the world, and they are in practical usage in industrial settings, where they are the basis for describing factory inventories and production processes. These descriptions are static and usually considered timeless.

In production planning, processes in a factory change products and materials. For planning it is useful to consider changes in the assertional knowledge of an ontology. One method for planning is the usage of Answer Set Programming (ASP) [4], which is a nonmonotonic knowledge representation formalism based on rules and solver tools based on SAT solving techniques. In ASP planning, time is represented as a sequence of steps and in each step different truth values can hold in the world.

| | |
|---|---|
| $ClosedBox \sqsubseteq Box$ | closed boxes are boxes |
| $OpenBox \sqsubseteq Box$ | open boxes are boxes |
| $ClosedBox \sqcap OpenBox \sqsubseteq \bot$ | no box is both open and closed |
| $MRobot \sqsubseteq Robot$ | there are manipulation robots ... |
| $PRobot \sqsubseteq Robot$ | ... and painting robots |
| $AffordsClosing \sqsubseteq Affordance$ | |
| $AffordsOpening \sqsubseteq Affordance$ | |
| $AffordsPainting \sqsubseteq Affordance$ | |
| $OpenBox \sqsubseteq AffordsClosing$ | open boxes can be closed |
| $ClosedBox \sqsubseteq AffordsOpening$ | closed boxes can be opened |
| $ClosedBox \sqsubseteq AffordsPainting$ | closed boxes can be painted |

**Figure 1.** Production Planning Terminology (OWL TBox).

| | | |
|---|---|---|
| $r1 : MRobot$ | $r2 : MRobot$ | $r2 : PRobot$ |
| $b1 : OpenBox$ | $b2 : OpenBox$ | $b3 : ClosedBox$ |

**Figure 2.** Production Planning Assertions (OWL ABox): we have two boxes $b1$ and $b2$ and two robots $r1$ and $r2$.

In practice, representing planning in DLs or representing ontological knowledge in ASP is usually impractical.[2] Therefore, it is natural to *integrate* DLs and ASP in a way that both formalisms are used in parallel within one integrated reasoning method. HEX is an ASP formalism for integrating ASP with external computations of all kinds, and one existing method for integrating ASP with DLs, DL-programs [8] actually use HEX as an underlying implementation formalism.

As a running example, consider an ontology about robots that can manipulate and paint boxes: the TBox is shown in Figure 1, the ABox of a concrete factory is shown in Figure 2. The goal of our production planning is to find a sequence of actions that paint all boxes where only robots in the class $PRobot$ can paint and only closed boxes afford[3] painting. Finally, robots in the class $MRobot$ can open and close boxes. Consider now the following information we would like to gain by quering this planning domain: (I) is there a plan that finishes the production within $n_1$ steps, using as few actions as possible, and what is that plan; (II) if we would have $n_2 > n_1$ steps available, which robots could we omit from the domain and still finish the job in time; (III) if we have only $n_3 < n_1$ steps available, which product

[1] Technische Universität Wien, Institute of Logic and Computation, Knowledge-Based Systems Group, Austria, peter.schueller@tuwien.ac.at



[2] It is sometimes possible due to complexity results.

[3] Affordances are a popular concept in robotics regarding prototypical actions: if an object affords a certain action, it provides the possibility to be affected by that action. An affordance is a property of an object, not an agent, and some agents might not be capable of performing the action due to geometric constraints or missing actuators.

could we omit from production to finish the other products in time.

DL-programs [8] are well-suited to express planning domains, including the above explanatory queries to the planning domain. DL-programs permit queries to ontologies based on a modification of an ontology that depends on the answer set of the solver. Therefore, DL-programs provide a bidirectional integration between ASP and DLs. Unfortunately, the concrete syntactic realization of DL-programs makes it cumbersome to encode production planning problems, because it is necessary to represent each time step using distinct logical predicates. Moreover, there is no current working implementation of DL-programs available as the licensing and the API of the underlying DL reasoner has changed.

In the following, we use the idea of DL-programs but we develop and present the problem directly within the HEX-formalism. This removes one layer of abstraction and simplifies presentation.

ASP is often praised for its Elaboration Tolerance [18] which is the property to create modular programs with a separation of concerns among program parts, and the possibility to modify each concern separately and easily without a need to modify other parts of the program. ASP also provides multiple answers and alternative solutions. This makes AI systems based on ASP suitable for addressing several concerns of the EU recommendation on 'human-centered, trustworthy AI systems', in particular

- Human Agency, concretely to 'self-assess and challenge the system'; and
- Technical Robustness, concretely to provide 'Fallback Plans'.

We next address above challenges and make the following contributions.

- We introduce a new way of interfacing with DLs from HEX where it is easily possible to describe multiple ontology modifications in the extension of one predicate, which facilitates the encoding of domains where multiple alternative worlds must be considered and we do not know beforehand how many of such alternative worlds exist. Planning is such a domain.
- We implement this integration using the OWLAPI interface and the HEXLITE solver [21], which provides a free and universal API for OWL DLs to be used together with answer set programs. HEXLITE is based on CLINGO [12] and therefore uses the state of the art in ASP solving. OWLAPI interfaces with many existing DL reasoners.
- We present a use case of production planning where action preconditions are defined by the ontology, action effects are realized as ontology modifications in the HEX program, and the initial state of the world is taken from the ontology, including in particular all individual names which are not known to the HEX program prior to solving.
- We formulate our thoughts about how Explainability can be realized in Logic programs and argue that the classical notion of Explainability does not apply to Logic Programming, and that instead the possibility to challenge the system with alternative scenarios is an appropriate way to make a Logic Program more human-centric.
- We discuss a possible usage of the production planning encoding for obtaining explanations about feasible and necessary changes in the production process to reach given production goals. Such reasoning can be useful for optimizing the factory and for hardening it against equipment malfunctions.

In the following, in Section 2 we provide preliminaries of HEX-programs and Description Logics, in Section 3 we describe the new framework for integrating OWLAPI in HEX and comment on the implementation, in Section 4 we use the framework for demonstrating how to explain contingencies in production planning, we discuss Explainability in Section 5 and we conclude in Section 6 with a brief discussion.

## 2 Preliminaries

### 2.1 HEX - Answer Set Programming with External Computations

We give syntax and semantics of the HEX formalism [5] which generalizes logic programs under answer set semantics [13] with external computations. A comprehensive introduction to HEX is given in [10]. HEXLITE[4,5] [21] is a solver for the HEX formalism which is based on Clingo [12] and enables implementation of external computations using PYTHON.

#### HEX *Syntax*

Let $\mathcal{C}$, $\mathcal{X}$, and $\mathcal{G}$ be mutually disjoint sets whose elements are called *constant names*, *variable names*, and *external predicate names*, respectively. Usually, elements from $\mathcal{X}$ and $\mathcal{C}$ are denoted with first letter in upper case and lower case, respectively; while elements from $\mathcal{G}$ are prefixed with '&'. Elements from $\mathcal{C} \cup \mathcal{X}$ are called *terms*. An (ordinary) *atom* is a tuple $p(Y_1, \ldots, Y_n)$ where $p \in \mathcal{C}$ is a predicate name and $Y_1, \ldots, Y_n$ are terms and $n \geq 0$ is the *arity* of the atom. An atom is *ground* if all its terms are constants. An *external atom* is of the form $\&g[Y_1, \ldots, Y_n](X_1, \ldots, X_m)$, where $Y_1, \ldots, Y_n$ and $X_1, \ldots, X_m$ are two lists of terms, called *input* and *output* lists, respectively, and $\&g \in \mathcal{G}$ is an external predicate name. We assume that input and output lists have fixed lengths $in(\&g) = n$ and $out(\&g) = m$. With each term $Y_i$ in the input list, $1 \leq i \leq n$, we associate a type $t_i \in \{\mathbf{cons}\} \cup \mathbb{N}$. We call the term *constant input* iff $t_i = \mathbf{cons}$, otherwise we call it *predicate input of arity* $t_i$.

A *rule* $r$ is of the form $\alpha_1 \vee \cdots \vee \alpha_k \leftarrow \beta_1, \ldots, \beta_n, not\ \beta_{n+1}, \ldots, not\ \beta_m$ with $m, k \geq 0$ where all $\alpha_i$ are atoms and all $\beta_j$ are either atoms or external atoms. We let $H(r) = \{\alpha_1, \ldots, \alpha_k\}$ and $B(r) = B^+(r) \cup B^-(r)$, where $B^+(r) = \{\beta_1, \ldots, \beta_n\}$ and $B^-(r) = \{\beta_{n+1}, \ldots, \beta_m\}$. A HEX-*program* is a finite set $P$ of rules.

#### HEX *Semantics*

Given a rule $r$, the grounding $grnd(r)$ of $r$ is obtained by systematically replacing all variables with constants from $\mathcal{C}$. Given a HEX-program $P$, the *Herbrand base* $HB_P$ of $P$ is the set of all possible ground versions of atoms and external atoms occurring in $P$ obtained by replacing variables with constants from $\mathcal{C}$. The grounding $grnd(P)$ of $P$ is given by $grnd(P) = \bigcup_{r \in P} grnd(r)$. Importantly, the set of constants $\mathcal{C}$ that is used for grounding a program is only partially given by the program itself: external computations may introduce new constants, for example external computations can import individual names from an ontology.

*Extensional Semantics* [9, 5] of external atoms are defined as follows: we associate a $(n+1)$-ary extensional evaluation function $F_{\&g}$ with every external predicate name $\&g \in \mathcal{G}$. Given an interpretation $I \subseteq HB_P$ and a ground input tuple

[4] github.com/hexhex/hexlite
[5] www.ai4eu.eu/resource/hexlite

$(x_1, \ldots, x_m)$, $F_{\&g}(I, y_1, \ldots, y_n)$ returns a set of ground output tuples $(x_1, \ldots, x_m)$. The external computation is *restricted* to depend (a) for contant inputs, i.e., $t_i = \mathbf{cons}$, only on the constant value of $y_i$; and (b) for predicate inputs, i.e., $t_i \in \mathbb{N}$, only on the extension of predicate $y_i$ of arity $t_i$ in $I$.[6]

An interpretation $I \subseteq HB_P$ is a *model* of an atom $a$, denoted $I \models a$ if $a$ is an ordinary atom and $a \in I$. $I$ is a model of a ground external atom $a = \&g[y_1, \ldots, y_n](x_1, \ldots, x_m)$ if $(x_1, \ldots, x_m) \in F_{\&g}(I, y_1, \ldots, y_n)$. Given a ground rule $r$, $I \models H(r)$ if $I \models a$ for some $a \in H(r)$; $I \models B(r)$ if $I \models a$ for all $a \in B^+(r)$ and $I \not\models a$ for all $a \in B^-(r)$; and $I \models r$ if $I \models H(r)$ whenever $I \models B(r)$. Given a HEX-program $P$, $I \models P$ if $I \models r$ for all $r \in grnd(P)$; the *FLP-reduct* [11] of $P$ with respect to $I \subseteq HB_P$, denoted $fP^I$, is the set of all $r \in grnd(P)$ such that $I \models B(r)$; $I \subseteq HB_P$ is an *answer set* of $P$ if $I$ is a minimal model of $fP^I$, and we denote by $\mathcal{AS}(P)$ the set of all answer sets of $P$.

An important shortcut notation that we use in this work is the *guess* in the head of a rule: $\{\alpha_1; \cdots; \alpha_k\} \leftarrow \beta_1, \ldots, \beta_n, not\ \beta_{n+1}, \ldots, not\ \beta_m$ with $m, k \geq 0$ is rewritten into a set of rules such that, given the rule body is satisfied, candidate answer sets are generated which contain all subsets of $\{\alpha_1, \ldots, \alpha_k\}$.

## 2.2 Description Logics and OWLAPI

Description Logics (DLs) [1] are logics of limited expressivity with decidable reasoning. In particular a DL usually permits only unary and binary predicates, called concepts and roles, respectively. DL reasoning is usually organized in a TBox (terminological axioms) and in an ABox (assertional axioms).

For the purposes of this paper it is sufficient to introduce concept inclusion axioms, concept intersection, the empty concept, and concept assertions. Concept inclusion axioms of form $C \sqsubseteq D$ denote that $\forall x : C(x) \rightarrow D(x)$, i.e., every instance in concept $C$ is also in concept $D$. Concept intersection $C \sqcap D$ is a concept expression that contains the intersection of two concepts, i.e., all instances that are both in $C$ and in $D$. The empty concept, $\bot$, denotes an impossible class that is always empty. A concept membership axiom of form $i : C$ denotes that the individual/constant $i$ is in concept $C$.

*Example* 1. In our running example TBox in Figure 1, the meaning of each axiom is written next to the axiom, except for the affordances which can be read as '$AffordsClosing$ is an $Affordance$', etc. The ABox in Figure 2 introduces individual constants into the theory and assigns concept memberships to them. □

Answer Set Programs and DLs have been combined in Description Logic Programs (DL-Programs) [8]. This formalism paved the way for the development of HEX-programs. In brief, DL-programs are Answer Set Programs with special DL-atoms of form

$$DL[C_1\ op_1\ p_1, \ldots, C_m\ op_m\ p_m; Q](\vec{t})$$

which evaluates the DL-Query $Q$ on an ontology modified by operations of form $C\ op\ p$ and evaluates to true for all tuples $\vec{t}$ in the result of query $Q$. The modifications can add ($op = \uplus$) or subtract ($op = \uplus$) the extension of predicate $p$ from the assertions of concept $C$. Note that one DL-atom always takes the whole extension of all predicates $p_1, \ldots, p_m$ to modify the ontology.

[6] Formally, this is the set $\{y_i(v_1, \ldots, v_{t_i}) \in I\}$.

# 3 Parameterized Multi-Modification-Integration of OWLAPI in HEX

Our novel integration interface has two major components: a representation for ontology modifications in ASP atoms (Section 3.1) and external atoms that query the ontology based on these modifications (Section 3.2).

## 3.1 Ontology Modifications

Ontology modifications are represented as atoms in ASP as follows.

**Definition 1.** A *modification atom* is of the form

$$\delta(\tau, \mu) \tag{1}$$

with $\delta$ some predicate name, $\tau \in \mathbb{N}$ and $\mu$ one of the following terms:

$$addc(C, I) \quad addop(OP, I_1, I_2) \quad adddp(DP, I, D) \tag{2}$$
$$delc(C, I) \quad delop(OP, I_1, I_2) \quad deldp(DP, I, D) \tag{3}$$

where $C$, $OP$, and $DP$ are OWL Class, Object Property, and Data Property IRIs, respectively, $I$, $I_1$, and $I_2$ are OWL Individual IRIs, and $D$ is either an integer or a string parseable as OWL Data value.

Intuitively the modifications in (2) add ABox assertions and those in (3) remove ABox assertions.

## 3.2 External Atoms

External atoms for querying the ontology are of the following form, where $\omega$ is a specifier for the ontology and $\delta/\tau$ are as above.

| | | |
|---|---|---|
| $\&dlConsistent[\omega, \delta, \tau]()$ | for consistency of the ontology | (4) |
| $\&dlC[\omega, \delta, \tau, C](I)$ | for querying class instances | (5) |
| $\&dlDP[\omega, \delta, \tau, DP](I_1, I_2)$ | for querying data properties | (6) |
| $\&dlOP[\omega, \delta, \tau, OP](I_1, D)$ | for querying object properties | (7) |

Intuitively, these external atoms query the ontology $\omega$ after it has been modified using all modifications in the extension of $\delta$ in the current answer set candidate $I$, selected by $\tau$. Formally, the above external atoms perform reasoning on the ontology modified by

$$\{\mu \mid \delta(\tau, \mu) \in I\}.$$

*Example* 2. In our running example, the external atom

```
&dlC[onto,delta,T,"ex:AffordsOpening"](B)
```

with `T=0` and an empty extension of `delta` is false for `B=b1` and true for `B=b2`, because the ontology contains assertions $b1 : ClosedBox$ and $b2 : OpenBox$ from which the reasoner infers that $b1$ affords opening (via Axiom $ClosedBox \sqsubseteq AffordsOpening$) but $b2$ does not.

The HEX-program defines in each time step `T` that the modification of the ontology corresponds with the planning state which is represented in predicate `s(Box,State,T)`. Therefore, if at `T=2` some action has caused a change in the state such that `s(b1,open,2)` and `s(b2,open,2)` are true, then the above external atom, with `T=2`, is false for both `B=b1` and `B=b2`. □

The format of the ontology specifier $\omega$ is a string pointing to a JSON file that holds the location of the OWL ontology file and provides namespaces for usage in ASP.

*Example* 3. In our running example, the file `meta.json` has the following content.

```
{
 "load-uri": "sample.owl",
 "namespaces": {
  "owl": "http://www.w3.org/2002/07/owl#",
  "ex": "http://www.kr.tuwien.ac.at/hexlite/example#"
 }
}
```

□

### 3.3 Implementation

The external atoms (4)–(7) have been implemented in the OWLAPI Plugin[7,8] for the HEXLITE [21] solver. The plugin uses JPYPE[9] to access OWLAPI[10] [17] and as reasoner currently uses HERMIT[11] [16].

To facilitate value invention from the ontology, i.e., to allow for importing all individual names from the ontology instead of specifying them in the HEX program, the OWLAPI plugin contains 'read-only' external atoms of the form

$$\&dlCro[\omega, C](I)$$
$$\&dlDPro[\omega, DP](I_1, I_2)$$
$$\&dlOPro[\omega, OP](I_1, D)$$

which correspond to (5)–(7) but operate on the unmodified ontology.

## 4 Use Case: Explaining Contingencies in Production Planning

Here we concretely develop the use case described in the Introduction as a HEX-program.

Figure 3 shows a HEX-program that integrates action planning with reasoning over the ontologies in Figures 1 and 2.

The following predicates are particularly important in this encoding:

- the state of the world is represented in a fluent predicate of the form s(<item>,<state>,<timestep>) which represents which item is in which state at a given time step;
- actions in the world are represented in a predicate of the form do(<action>,<timestep>) which represents what is done at which time step.

Conceptually, actions are applied to a state at time step $T$ and their effect is realized in time step $T + 1$.

### 4.1 Encoding

From top to bottom, the encoding contains the following sections (see also the comments in the figure).

- The constant onto is defined that points to the JSON file describing ontology location and namespaces;
- a sequence of time steps, starting from 0 until finaltimestep is defined, and all but the last time step are steps where actions can happen (actStep);
- the set of known robot and box constants is imported read-only from the ontology into domain predicates box and robot;

```
#const onto="meta.json".

% timesteps
step(0..finaltimestep).
next(T,T+1) :- step(T), step(T+1).
actStep(T) :- next(T,_).

% robots and boxes are taken from the ontology
robot(Robot) :- &dlCro[onto,"ex:Robot"](Robot).
box(Box) :- &dlCro[onto,"ex:Box"](Box).

% derive initial s from the ontology
s(Box,open,0) :- &dlCro[onto,"ex:OpenBox"](Box).
s(Box,closed,0) :- &dlCro[onto,"ex:ClosedBox"](Box).

% fluent inertia, mutual exclusivity of open and closed
s(A,B,T') :- s(A,B,T), next(T,T'), not -s(A,B,T').
-s(Box,closed,T) :- s(Box,open,T).
-s(Box,open,T) :- s(Box,closed,T).

% the state of open/closed is propagated to the ontology
delta(T,addc("ex:ClosedBox",Box)) :- s(Box,closed,T).
delta(T,delc("ex:OpenBox",Box)) :- s(Box,closed,T).
delta(T,addc("ex:OpenBox",Box)) :- s(Box,open,T).
delta(T,delc("ex:ClosedBox",Box)) :- s(Box,open,T).

% require consistent ontology at each time step
:- not &dlConsistent[onto,delta,T], step(T).

% actions are licensed through the ontology
{ do(act(open,Robot,Box),T) } :- actStep(T),
  &dlC[onto,delta,T,"ex:AffordsOpening"](Box), box(Box),
  &dlC[onto,delta,T,"ex:MRobot"](Robot), robot(Robot).
{ do(act(close,Robot,Box),T) } :- actStep(T),
  &dlC[onto,delta,T,"ex:AffordsClosing"](Box), box(Box),
  &dlC[onto,delta,T,"ex:MRobot"](Robot), robot(Robot).
{ do(act(paint,Robot,Box),T) } :- actStep(T),
  &dlC[onto,delta,T,"ex:AffordsPainting"](Box), box(Box),
  &dlC[onto,delta,T,"ex:PRobot"](Robot), robot(Robot).

% only one action per robot per time point
:- step(T), robot(R), 2<=#count{ A,B : do(act(A,R,B),T) }.
% only one act per box per time point
:- step(T), box(B), 2<=#count{ A,R : do(act(A,R,B),T) }.

% action effects
s(Box,open,T') :- do(act(open,_,Box),T), next(T,T').
s(Box,closed,T') :- do(act(close,_,Box),T), next(T,T').
s(Box,painted,T') :- do(act(paint,_,Box),T), next(T,T').
```

**Figure 3.** Production Planning Domain (HEX-program).

- the initial state of the planning domain is obtained from the ontology;
- all fluents are inertial, i.e., the fluent state it is maintained unless otherwise specified, moreover a box cannot be open and closed at the same time;
- the fluents are represented in delta predicates so that the OWLAPI integration can represent the fluent values not only in ASP but also in the ontology;[12]
- we require consistency of the ontology for all time steps, using the &dlConsistent external atom;
- we guess which action is applied, where actions are licensed by certain affordances on boxes and can be performed by certain robot types;
- we constrain actions so that each robot can only perform one action at each step, and each box can be affected by only one action at each step; finally
- we define the effect of actions on the fluents.

[7] github.com/hexhex/hexlite-owlapi-plugin/
[8] www.ai4eu.eu/resource/hexlite-owlapi-plugin/
[9] github.com/jpype-project/jpype/
[10] github.com/owlcs/owlapi
[11] www.hermit-reasoner.com/

[12] Note, that we remove the OWL assertions that correspond with non-true fluent values, and we add assertions for true fluent values. This ensures that the initial state (which is represented in the ontology in form of $OpenBox$ and $ClosedBox$ assertions does not interfere with reasoning in time steps $T > 0$. We could get rid of all 'del' operations by using a separate ontology for specifying the initial state. This is possible using the plugin but it would complicate the example.

(c)
```
% make actions as early as possible
% and as few as possible
:~ do(A,T). [T+1@1,A]
```

(i)
```
% aim to finish in time step 3
#const finaltimestep=3.

% require boxes to be painted in the final step
:- box(B), not state(B,painted,finaltimestep).
```

(ii)
```
% aim to finish in time step 5
#const finaltimestep=5.

% deactivate as many robots as possible
{ explanation(deactivate(R)) } :- robot(R).
:~ not explanation(deactivate(R)), robot(R). [1@2,R]

% deactivated robots cannot do actions
:- explanation(deactivate(R)),
   actStep(S), do(action(_,R,_),T).

% require boxes to be painted in the final step
:- box(B), not state(B,painted,finaltimestep).
```

(iii)
```
% aim to finish in time step 2
#const finaltimestep=2.

% guess which boxes to skip for painting
{ explanation(skip(B)) } :- box(B).
% skip as few as possible
:~ explanation(skip(B)), box(B). [1@2,B]

% require non-skipped boxes to be painted
:- box(B), not explanation(skip(B)),
   not state(B,painted,finaltimestep).
```

**Figure 4.** Production Planning Queries (HEX-program fragments): (c) common optimization constraint, (i) planning query "paint all boxes within 3 time steps", (ii) explanation query "which robots could be omitted when achieving (i) within 5 time steps", (iii) explanation query "which boxes need to remain unpainted when attempting (i) within 2 time steps".

Given a value for finaltimestep this encoding does not contain a planning goal, that means it will produce all possible plans and all resulting states without a restriction on the final state. For example, resulting plans include the plan without actions, and the plan that repeatedly opens and closes a single box.

## 4.2 Queries and Results

Figure 4 contains HEX-program fragments (C) and (I)–(III) that can be added to the encoding in Figure 3 to query the domain for certain planning goals.

(C) The weak constraint in (C) incurs a cost of $T$ for each action at time step $T$ at priority level @1, that means the solution will aim to use as few actions as possible and they will be done as early as possible.

(I) Fragment (I) defines a query for plans that end at time step 3. The constraint requires that all boxes must be painted at the final time step.

(II) Fragment (III) queries for plans with final time step 5 and guesses for each robot whether it shall be deactivated or not. A maximum of robots will be deactivated at priority level @2. Moreover, for a deactivated robot all actions are forbidden and we again require all boxes to be painted in the final step.

(III) Fragment (III) defines a query for plans that end at time step 2. A guess is done for skipping certain boxes. The weak constraint incurs a cost for each box skipped at priority level @2. Finally we require all non-skipped boxes to be painted at the final time step.

When we compute answer sets using the above program fragments integrated with the above ontology, we obtain the following results.

**(C)+(I) Painting all boxes until time step 3:** Table 1 shows two optimal answer sets for this query. We observe that the integration successfully permitted only those actions that are licensed by affordances in the ontology. Multiple plans of the same quality exist, they differ in the order of closed and painted boxes. Table 2 provides the ontology modifications that are used in each time step of the planning to obtain the first plan in Table 1.

**(C)+(II) Explaining how to reduce the number of robots by achieving the goal only at time step 5:** Table 3 shows one of the optimal answer sets: an explanation together with a witnessing plan. We observe that the robot $r2$, which can both paint and manipulate boxes, was chosen to do the whole work, and $r1$, which can only manipulate boxes, was deactivated. As in the previous example, multiple plans of same quality exist. However, all plans deactivate $r1$ because without $r2$ the goal can not be reached.

**Table 1.** Two optimal plans (states + actions) for query (C)+(I): painting all boxes until step 3.

| Step | $b1$ | $b2$ | $b3$ | Actions |
|---|---|---|---|---|
| 0 | open | open | closed | $r1$ closes $b1$<br>$r2$ paints $b3$ |
| 1 | closed | open | closed<br>painted | $r1$ closes $b2$<br>$r2$ paints $b1$ |
| 2 | closed<br>painted | closed | closed<br>painted | $r2$ paints $b2$ |
| 3 | closed<br>painted | closed<br>painted | closed<br>painted | |

| Step | $b1$ | $b2$ | $b3$ | Actions |
|---|---|---|---|---|
| 0 | open | open | closed | $r1$ closes $b2$<br>$r2$ paints $b3$ |
| 1 | open | closed | closed<br>painted | $r1$ closes $b1$<br>$r2$ paints $b2$ |
| 2 | closed | closed<br>painted | closed<br>painted | $r2$ paints $b1$ |
| 3 | closed<br>painted | closed<br>painted | closed<br>painted | |

**Table 2.** Ontology states corresponding to time steps for the first plan in Table 1.

| Step | Effective Ontology Modifications | |
|---|---|---|
| 0 | delc("ex:ClosedBox",b1)<br>delc("ex:ClosedBox",b2)<br>delc("ex:OpenBox",b3) | addc("ex:OpenBox",b1)<br>addc("ex:OpenBox",b2)<br>addc("ex:ClosedBox",b3) |
| 1 | delc("ex:OpenBox",b1)<br>delc("ex:ClosedBox",b2)<br>delc("ex:OpenBox",b3) | addc("ex:ClosedBox",b1)<br>addc("ex:OpenBox",b2)<br>addc("ex:ClosedBox",b3) |
| 2 | delc("ex:OpenBox",b1)<br>delc("ex:OpenBox",b2)<br>delc("ex:OpenBox",b3) | addc("ex:ClosedBox",b1)<br>addc("ex:ClosedBox",b2)<br>addc("ex:ClosedBox",b3) |
| 3 | Ontology is not evaluated (step 3 is not an actStep) | |

**(C)+(III) Explaining how to finish the work until time step 2 by skipping a product:** Table 4 displays one of the optimal explanations together with its witnessing plan. Note that $b2$ is not touched by the robots, although $r1$ is idle in Step 1 and could close it. This

**Table 3.** Optimal plan for (C)+(II): explaining how to reduce the number of robots by postponing the goal to time step 5.

Explanation: '**deactivate** $r1$'

| Step | $b1$ | $b2$ | $b3$ | Actions |
|---|---|---|---|---|
| 0 | open | open | closed | $r2$ closes $b1$ |
| 1 | closed | open | closed | $r2$ closes $b2$ |
| 2 | closed | closed | closed | $r2$ paints $b2$ |
| 3 | closed | closed painted | closed | $r2$ paints $b3$ |
| 4 | closed | closed painted | closed painted | $r2$ paints $b1$ |
| 5 | closed painted | closed painted | closed painted | |

can be explained by constraint (C) which requires robots to perform only the minimally required actions for reaching the goal.

**Table 4.** Optimal plan for (C)+(II): explaining how to reduce the number of robots by postponing the goal to time step 5.

Explanation: '**skip** $b1$'

| Step | $b1$ | $b2$ | $b3$ | Actions |
|---|---|---|---|---|
| 0 | open | open | closed | $r1$ closes $b1$<br>$r2$ paints $b3$ |
| 1 | closed | open | closed painted | $r2$ paints $b1$ |
| 2 | closed painted | open | closed painted | |

## 5 Explainability

In this section, we argue that Explainability as it is usually understood is not applicable to Answer Set Programming and propose an alternative.

The classical notion of Explanation in AI aims to provide to the user of a system at least some parts of the rationale, assumptions, data, and reasoning process that leads to a decision. In the case of Logic Programming, decisions are made on (i) the basis of a mathematical framework, i.e., based on the mathematical definition of Logic Programming, (ii) based on the program given by the programmer, and (iii) based on the query we pose to the program, again in the shape of program rules that have clearly defined syntax and semantics. Therefore, explanations of the classical form would amount to teaching formal semantics and presenting the program that is reasoned upon by the solver software, which is neither feasible for an end-user nor does it serve the intention of Explainability. Classical explanations apply mostly to Machine Learning systems where the system makes a decision based on input data and a model learned from training data, and where usually a big amount of input data (such as audio or video data) is fed to an algorithm that makes a decision that is based on a small part of that input data.

Classical Logic Programming makes no decisions in the presence of uncertainty, it produces multiple solutions that are equally viable. In case of optimization criteria, it is also clearly determined which solution(s) are preferred.

Teaching end-users the formal semantics of ASP so that they can understand the reasoning process and programs requires to make end-users experts in order to use explanations. This approach would be too fine-grained to have any practical value. Therefore, a more macroscopic approach for Explainability is necessary.

We therefore argue that the best way to gain Explainability in Logic Programming is an interaction possibility where users can (i) challenge the system with alternative scenarios, (ii) explore differences between alternative solutions to a single scenario, and (iii) change optimization criteria to explore their effect on solutions. The short query snippets we presented in the previous section (Figure 4) provide such a way of challenging the system using concise program additions or modifications.

The ASP literature contains several studies about such types of Explainability under the names Diagnosis [3], Debugging [20], and Explaining Inconsistency [22].

## 6 Discussion, Related and Future Work

We have described a new interface for integrating Description Logics, in particular every DL reasoner that is compatible with OWLAPI, with logic programming under Answer Set Semantics, concretely using the HEX formalism. The benefit of such an integration is, that the ontology is used in its original form, and that the ASP planning domain permits all freedoms that ASP provides for planning, as demonstrated by our example use cases with explanatory queries. An alternative solution for integrating ontologies with logic programs would be to convert the ontology into a logic program, but this creates the need for a conversion process and for maintenance of not only the ontology but also the conversion process.

Using logic programming for planning is only one possibility among many others. The benefit of using Answer Set Programming is the power and freedom that comes with a guess-and-check paradigm where guesses and constraints and optimizations can be combined freely. In particular, Answer Set Programming permits expressive representations such as indirect effects/ramifications and domain-global constraints and optimization criteria that are difficult or impossible to model in STRIPS and PDDL.

## Related Work

DL-Programs [8] permit the integration of DL reasoning into an ASP planning domain similar to our domain above, however for each time step a separate DL atom with a separate $delta$ predicate would be required (i.e., for $n$ time steps we would need to define $delta_1$, …, $delta_n$ and create $n$ instances of all rules in Figure 3 where $\delta$ is used). Therefore, our new integration interface does not increase expressivity but it greatly improves conciseness, readability, and maintainability of the encoding, because ASP variables can be used to express selection of a certain sub-extension of $delta$ to be used as ontology modification.

Action Languages, e.g., PDDL [19] or $\mathcal{AR}$ [14] are custom languages for representing planning domains. While basic PDDL does not provide support for non-deterministic effects, its extension PPDDL supports representation of probabilistic effects and $\mathcal{AR}$ (and several other planning languages) support representation of possibilistic effects (i.e., effects that can happen without providing a probability). These action languages usually come with specific reasoning methods and do not support what-if scenarios out of the box, unless they can be represented directly in the respective planning input language. ASP planning has the advantage that everything is represented as an ASP rule and the programmer is free to modify rules in order

to represent additional what-if scenarios just by replacing a deterministic rule by a rule with a guessing construction in the head, or by weakening or strengthening the rule in other ways. However, the advantages of dedicated action languages and the freedom of representation in ASP can be combined by rewriting a planning domain to ASP rules for evaluating it in a generic ASP solver. This has been done, e.g., for the $\mathcal{K}$ [6, 7] and $\mathcal{C}+$ [2, 15] planning languages.

## Future Work

As future work on the API side of the integration, we see an extension of the interface to permit arbitrary DL queries using the OWLAPI DL Query Parser, and additional ontology modifications that permit adding and removing axioms. As future work on the implementation side, it will be necessary to generate on-demand constraints that speed up the reasoning. Due to time constraints it was not possible to implement this technique so far, therefore we here do not report timing results.

## ACKNOWLEDGEMENTS

We would like to acknowledge fruitful discussions about Description Logics and Production Planning with Magdalena Ortiz, Ivan Gocev, Raoul Blankertz, Sonja Zillner, and Stephan Grimm. We would also like to thank the referees and chairs for their feedback. This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement 825619 (AI4EU).

## REFERENCES

[1] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider, volume 32, Cambridge University Press, 2003.

[2] Joseph Babb and Joohyung Lee, 'cplus2asp: Computing action language $\mathcal{C}+$ in answer set programming', in *International Conference on Logic Programming and Nonmonotonic Reasoning*, pp. 122–134. Springer, (2013).

[3] Marcello Balduccini and Michael Gelfond, 'Diagnostic reasoning with a-prolog', *Theory and Practice of Logic Programming*, **3**(4-5), 425–461, (2003).

[4] Gerd Brewka, Thomas Eiter, and Miroslaw Truszczynski, *AI Magazine: Special Issue on Answer Set Programming, vol. 37(3)*, AAAI Press, 2016.

[5] T. Eiter, M. Fink, G. Ianni, T. Krennwallner, C. Redl, and P. Schüller, 'A model building framework for Answer Set Programming with external computations', *Theory and Practice of Logic Programming*, **16**(4), (2016).

[6] Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres, 'Planning under incomplete knowledge', in *International Conference on Computational Logic*, pp. 807–821. Springer, (2000).

[7] Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres, 'The dlv k planning system: Progress report', in *European Workshop on Logics in Artificial Intelligence*, pp. 541–544. Springer, (2002).

[8] Thomas Eiter, Giovambattista Ianni, Thomas Lukasiewicz, Roman Schindlauer, and Hans Tompits, 'Combining Answer Set Programming with Description Logics for the Semantic Web', *Artificial Intelligence*, **172**(12-13), 1495–1539, (2008).

[9] Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits, 'Effective integration of declarative rules with external evaluations for Semantic-Web reasoning', in *European Semantic Web Conference (ESWC)*, pp. 273–287, (2006).

[10] Thomas Eiter, Tobias Kaminski, Christoph Redl, Peter Schüller, and Antonius Weinzierl, 'Answer Set Programming with external source access', in *Reasoning Web International Summer School*, pp. 204–275, (2017).

[11] Wolfgang Faber, Gerald Pfeifer, and Nicola Leone, 'Semantics and complexity of recursive aggregates in Answer Set Programming', *Artificial Intelligence*, **175**(1), 278–298, (2011).

[12] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub, 'Multi-shot ASP solving with clingo', *Theory and Practice of Logic Programming*, 1–56, (2018).

[13] Michael Gelfond and Vladimir Lifschitz, 'Classical negation in logic programs and deductive databases', *New Generation Computing*, **9**, 365–385, (1991).

[14] Enrico Giunchiglia, G Neelakantan Kartha, and Vladimir Lifschitz, 'Representing action: Indeterminacy and ramifications', *Artificial Intelligence*, **95**(2), 409–438, (1997).

[15] Enrico Giunchiglia, Joohyung Lee, Vladimir Lifschitz, Norman McCain, and Hudson Turner, 'Nonmonotonic causal theories', *Artificial Intelligence*, **153**(1-2), 49–104, (2004).

[16] Birte Glimm, Ian Horrocks, Boris Motik, Giorgos Stoilos, and Zhe Wang, 'HermiT: an OWL 2 reasoner', *Journal of Automated Reasoning*, **53**(3), 245–269, (2014).

[17] Matthew Horridge and Sean Bechhofer, 'The OWL API: a Java API for OWL ontologies', *Semantic web*, **2**(1), 11–21, (2011).

[18] John McCarthy, 'Elaboration tolerance', in *Common Sense*, volume 98, (1998).

[19] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL – The planning domain definition language, 1998. Technical Report CVC TR-98-003/DCS TR-1165.

[20] Johannes Oetsch, Jörg Pührer, and Hans Tompits, 'Catching the ouroboros: On debugging non-ground answer-set programs', *Theory Pract. Log. Program.*, **10**(4-6), 513–529, (2010).

[21] Peter Schüller, 'The Hexlite solver', in *European Conference on Logics in Artificial Intelligence (JELIA)*, pp. 593–607. Springer, (2019).

[22] Claudia Schulz, Ken Satoh, and Francesca Toni, 'Characterising and explaining inconsistency in logic programs', in *International Conference on Logic Programming and Nonmonotonic Reasoning*, pp. 467–479. Springer, (2015).