Determining Action Reversibility in STRIPS Using Answer Set and Epistemic Logic Programming*

Lukáš Chrpa[1] [0000-0001-9713-7748], Wolfgang Faber[2] [0000-0002-0330-5868], Daniel Fišer[1] [0000-0003-2383-9477], and Michael Morak[2] [0000-0002-2077-7672]

[1] Czech Technical University in Prague, Czechia
{chrpaluk,fiserdan}@fel.cvut.cz
[2] University of Klagenfurt, Austria
{wolfgang.faber,michael.morak}@aau.at

Abstract. In planning and reasoning about action and change, reversibility of actions is the problem of deciding whether the effects of an action can be reverted by applying other actions in order to return to the original state. While this problem has been studied for some time, recently there as been renewed interest in the context of the language PDDL. After reviewing the concepts, in this paper we propose solutions by leveraging an existing translation from PDDL to Answer Set Programming (ASP), which we then use to solve the problem via ASP and Epistemic Logic Programming (ELP). This work provides sound and complete systems for determining reversibility of PDDL actions (restricted to the STRIPS fragment), while also providing insight into the performance of a state-of-the-art ELP solver and how it compares to ASP solving.

Keywords: Planning · Answer Set Programming · Reasoning about Action and Change · Epistemic Logic Programming.

1 Introduction

Traditionally, the field of Automated Planning [21, 22] deals with the problem of generating a sequence of actions—a plan—that transforms an initial state of the environment to some goal state. Actions, in plain words, stand for modifiers of the environment. One interesting question is whether the effects of an action are reversible (by other actions), or in other words, whether the action effects can be undone. Notions of reversibility have previously been investigated; cf. e.g., works by Eiter et al. [14] or by Daum et al. [11].

Studying action reversibility is important for several reasons. Intuitively, actions whose effects cannot be reversed might lead to dead-end states from which

* This paper is based on [7] and [16]. The work was supported by the S&T Cooperation CZ 05/2019 "Identifying Undoable Actions and Events in Automated Planning by Means of Answer Set Programming."

the goal state is no longer reachable. Early detection of a dead-end state is beneficial in a plan generation process [24]. Reasoning in more complex structures such as Agent Planning Programs [12] which represent networks of planning tasks where a goal state of one task is an initial state of another is even more prone to dead-ends [8]. Concerning non-deterministic planning, for instance Fully Observable Non-Deterministic (FOND) Planning, where actions have non-deterministic effects, determining reversibility or irreversibility of each set of effects of the action can contribute to early dead-end detection, or to generalizing recovery from undesirable action effects which is important for efficient computation of strong (cyclic) plans [6]. Concerning online planning, we can observe that applying reversible actions is safe and hence we might not need to explicitly provide the information about safe states of the environment [10]. Another, although not very obvious, benefit of action reversibility is in plan optimization. If the effects of an action are later reversed by a sequence of other actions in a plan, these actions might be removed from the plan, potentially shortening it significantly. It has been shown that under such circumstances, pairs of inverse actions, which are a special case of action reversibility, can be removed from plans [9].

In [25] we introduced a general framework for action reversibility that offers a broad definition of the term, and generalizes many of the already proposed notions of reversibility, like "undoability" proposed in [11], or the concept of "reverse plans" as introduced in [14]. The concept of reversibility in [25] directly incorporates the set of states in which a given action should be reversible. We call these notions S-reversibility and φ-reversibility, where the set S contains states, and the formula φ describes a set of states in terms of propositional logic. These notions are then further refined to universal reversibility (referring to the set of all states) and to reversibility in some planning task Π (referring to the set of all reachable states w.r.t. the initial state specified in Π). These last two versions match the ones proposed in [11]. Furthermore, our notions can be further restricted to require that some action is reversible by a single "reverse plan" that is not dependent of the state for which the action is reversible. For single actions, this matches the concept of the same name proposed in [14].

The complexity analyses in [25] indicate that several of these tasks can be solved by means of Answer Set Programming (ASP), but also using Epistemic Logic Programming (ELP). The main contribution of this paper is to leverage the translations implemented in plasp [13] to produce encodings to effectively solve some reversibility tasks on PDDL domains, restricted, for now, to the STRIPS [17] fragment. We describe both ASP and ELP encodings in two versions each: one version exploits a shortcut, while the other is more extendible. While ELP might at a first glance might appear unsuitable, as it provides more expressivity than needed, it turns out to permit arguably more natural and extendible encodings. We then also perform a preliminary experimental analysis on synthetic test cases. This analysis shows that the ASP encodings have a clear edge over the ELP encodings, which can be explained by the well-established nature of ASP solvers versus the comparatively experimental nature of ELP solvers, though.

Structure. The remainder of the paper is organized as follows. In Section 2, we introduce basic concepts; Section 3 then reviews definitions and properties of different versions of reversibility from [25]; in Section 4 we review the plasp format and provide some ASP and ELP encodings for reversibility tasks; in Section 5 we report on the preliminary experiments before concluding in Section 6.

2 Background

STRIPS Planning. Let $\mathcal{F}$ be a set of *facts*, that is, atomic statements about the world. Then, a subset $s \subseteq \mathcal{F}$ is called a *state*, which intuitively represents a set of facts considered to be true. An action is a tuple $a = \langle pre(a), add(a), del(a) \rangle$, where $pre(a) \subseteq \mathcal{F}$ is the set of *preconditions* of a, and $add(a) \subseteq \mathcal{F}$ and $del(a) \subseteq \mathcal{F}$ are the add and delete effects of a, respectively. W.l.o.g., we assume actions to be well-formed, that is, $add(a) \cap del(a) = \emptyset$ and $pre(a) \cap add(a) = \emptyset$. An action a is *applicable* in a state s iff $pre(a) \subseteq s$. The result of applying an action a in a state s, given that a is applicable in s, is the state $a[s] = (s \setminus del(a)) \cup add(a)$. A sequence of actions $\pi = \langle a_1, \ldots, a_n \rangle$ is applicable in a state s_0 iff there is a sequence of states $\langle s_1, \ldots, s_n \rangle$ such that, for $0 < i \leq n$, it holds that a_i is applicable in s_{i-1} and $a_i[s_{i-1}] = s_i$. Applying the action sequence π on s_0 is denoted $\pi[s_0]$, with $\pi[s_0] = s_n$. The *length* of action sequence π is denoted $|\pi|$.

A *STRIPS planning task* $\Pi = \langle \mathcal{F}, \mathcal{A}, s_0, G \rangle$ is a tuple consisting of a set of *facts* $\mathcal{F} = \{f_1, \ldots, f_n\}$, a set of *(ground) actions* $\mathcal{A} = \{a_1, \ldots, a_m\}$, an *initial state* $s_0 \subseteq \mathcal{F}$, and a *goal specification* (or, simply, *goal*) $G \subseteq \mathcal{F}$. A state $s \subseteq \mathcal{F}$ is a *goal state (for Π)* iff $G \subseteq s$. An action sequence π is called a *plan* iff $\pi[s_0] \supseteq G$. We further define several relevant notions w.r.t. a planning task Π. A state s is *reachable from state s'* iff there exists an applicable action sequence π such that $\pi[s'] = s$. A state $s \in 2^{\mathcal{F}}$ is simply called *reachable* iff it is reachable from the initial state s_0. The set of all reachable states in Π is denoted by $\mathcal{R}_\Pi$. An action a is *reachable* iff there is some state $s \in \mathcal{R}_\Pi$ such that a is applicable in s.

Deciding whether a STRIPS planning task has a plan is known to be PSPACE-complete in general and it is NP-complete if the length of the plan is polynomially bounded [3].

Answer Set Programming (ASP) and Epistemic Logic Programming (ELP). We assume the reader is familiar with ASP and ELP and will only give a very brief overview of the core language(s). For more information on ASP, we refer to standard literature [2, 18, 23], and, in our case, the ASP-Core-2 input language format [5]. For more information on ELP, we refer to the original paper proposing ELPs [19] (therein named *Epistemic Specifications*), whose semantics we will use in the present paper.

ASP programs consist of sets of *rules* of the form

$$a_1 \mid \cdots \mid a_n \leftarrow b_1, \ldots, b_\ell, \neg b_{\ell+1}, \ldots, \neg b_m.$$

In these rules, all a_i and b_i are *atoms* of the form $p(t_1, \ldots, t_n)$, where p is a predicate name, and $t_1, \ldots, t_n$ are terms, that is, either variables or constants.

ELP programs consist of sets of *rules* of the form

$$a_1 \vee \ldots \vee a_n \leftarrow \ell_1, \ldots, \ell_m.$$

Here, all a_i are *atoms* of the form $p(t_1, \ldots, t_n)$, where p is a predicate name, and $t_1, \ldots, t_n$ are terms, that is, either variables or constants. Each ℓ_i is either an objective or subjective literal, where objective literals are of the form a or $\neg a$ (for a an atom), and subjective literals are of the form $\mathbf{K}\, l$ or $\neg\mathbf{K}\, l$, where l is an objective literal. Note that often the operator $\mathbf{M}$ is also used, which we will simply treat as a shorthand for $\neg\mathbf{K}\,\neg$. So the difference to ASP rules is that objective literals are permitted in addition in rule bodies.

The domain of constants in an ASP and ELP program P is given implicitly by the set of all constants that appear in it. Generally, before evaluating a program, variables are removed by a process called *grounding*, that is, for every rule, each variable is replaced by all possible combinations of constants, and appropriate ground copies of the rule are added to the resulting program $\mathit{ground}(P)$. In practice, several optimizations have been implemented in state-of-the-art grounders that try to minimize the size of the grounding.

The result of a (ground) ASP program P is calculated as follows [20]. An *interpretation* I (i.e., a set of ground atoms appearing in P) is called a *model* of P iff it satisfies all the rules in P in the sense of classical logic. It is further called an *answer set* of P iff there is no proper subset $I' \subset I$ that is a model of the so-called reduct P^I of P w.r.t. I. P^I is defined as the set of rules obtained from P where all negated atoms on the right-hand side of the rules are evaluated over I and replaced by $\top$ or $\bot$ accordingly. The main decision problem for ASP is deciding whether a program has at least one answer set. This has been shown to be Σ_P^2-complete [15] (for ground programs, or as data complexity for non-ground programs).

The result of a (ground) ELP program P is calculated as follows [19]. A set of interpretations $\mathcal{I}$ satisfies a subjective literal $\mathbf{K}\, l$ (denoted $\mathcal{I} \vDash \mathbf{K}\, l$) iff the objective literal l is satisfied in all interpretations in $\mathcal{I}$. The *epistemic reduct* $P^{\mathcal{I}}$ of P w.r.t. $\mathcal{I}$ is obtained from P by replacing all subjective literals ℓ with either $\top$ in case where $\mathcal{I} \vDash \ell$, or with $\bot$ otherwise. $P^{\mathcal{I}}$, therefore, is an ASP program, that is, a program without subjective literals. The solutions to an ELP P are called *world views*. A set of interpretations $\mathcal{I}$ is a world view of P iff $\mathcal{I} = AS(P^{\mathcal{I}})$ [19], where $AS(P^{\mathcal{I}})$ denotes the set of stable models (or answer sets) of the logic program $P^{\mathcal{I}}$ according to the semantics of answer set programming [20]. Checking whether a world view exists for an ELP is known to be Σ_3^P-complete in general [26].

3 Reversibility of Actions

In this section, we describe the notion of reversibility of actions. In particular, we focus on the notion of uniform reversibility, but note that there are other notions of reversibility which are lied out and explained in detail by Morak et al. [25]. Intuitively, we call an action reversible if there is a way to undo all the effects that this action caused, and we call an action *uniformly reversible* if its effects can be undone by a single sequence of actions irrespective of the state where the action was applied.

While this intuition is fairly straightforward, when formally defining this concept, we also need to take several other factors into account—in particular, the set of possible states where an action is considered plays an important role [25].

Definition 1. *Let $\mathcal{F}$ be a set of facts, $\mathcal{A}$ be a set of actions, $S \subseteq 2^{\mathcal{F}}$ be a set of states, and $a \in \mathcal{A}$ be an action. We call a* uniformly S-reversible *iff there exists a sequence of actions $\pi = \langle a_1, \ldots, a_n \rangle \in \mathcal{A}^n$ such that for each $s \in S$ wherein a is applicable it holds that π is applicable in $a[s]$ and $\pi[a[s]] = s$.*

The notion of uniform reversibility in the most general sense does not depend on a concrete STRIPS planning task, but only on a set of possible actions and states w.r.t. a set of facts. Note that the set of states S is an explicit part of the notion of uniform S-reversibility.

Based on this general notion, it is then possible to define several concrete sets of states S that are useful to consider when considering whether an action is reversible. For instance, S could be defined via a propositional formula over the facts in $\mathcal{F}$. Or we can consider a set of all possible states ($2^{\mathcal{F}}$) which gives us a notion of uniform reversibility that applies to all possible planning tasks that share the same set of facts and actions (i.e., the tasks that differ only in the initial state or goals). Or we can move our attention to a specific STRIPS instance and ask whether a certain action is uniformly reversible for all states reachable from the initial state.

Definition 2. *Let $\mathcal{F}$, $\mathcal{A}$, S, and a be as in Definition 1. We call the action a*

1. uniformly φ-reversible *iff a is uniformly S-reversible in the set S of models of the propositional formula φ over $\mathcal{F}$;*
2. uniformly reversible in $\mathit{\Pi}$ *iff a is uniformly $\mathcal{R}_{\mathit{\Pi}}$-reversible for some STRIPS planning task $\mathit{\Pi}$; and*
3. universally uniformly reversible*, or, simply,* uniformly reversible*, iff a is uniformly $2^{\mathcal{F}}$-reversible.*

Given the above definitions, we can already observe some interrelationships. In particular, universal uniform reversibility (that is, uniform reversibility in the set of all possible states) is obviously the strongest notion, implying all the other, weaker notions. It may be particularly important when one wants to establish uniform reversibility irrespective of the concrete STRIPS instance.

The notion of uniform reversibility naturally gives rise to the notion of the reverse plan. We say that some action a has an *(S-)reverse plan* π iff a is uniformly (S-)reversible using the sequence of actions π. It is interesting to note that this definition of the reverse plan based on uniform reversibility now coincides with the same notion as defined by [14]. Note, however, that in that paper the authors use a much more general planning language.

Even if the length of the reverse plan is polynomially bounded, the problem of deciding whether an action is uniformly (φ-)reversible is intractable. In particular, deciding whether an action is universally uniformly reversible (resp. uniformly φ-reversible) by a polynomial length reverse plan is NP-complete (resp. in Σ_2^P) [25].

4 Reversibility Encodings in ASP and ELP

After reviewing the relevant features of *plasp* [13] in Section 4.1, we present our encodings for determining reversibility in Sections 4.2 and 4.3.

4.1 The *plasp* Format

The system *plasp* [13] transforms PDDL domains and problems into facts. Together with suitable programs, plans can then be computed by ASP solvers—and hence also by ELP solvers, since ELPs are a superset of ASP programs. Given a STRIPS domain with facts $\mathcal{F}$ and actions $\mathcal{A}$, the following relevant facts and rules will be created by *plasp*:

- `variable(variable("f")).` for all $\texttt{f} \in \mathcal{F}$
- `action(action("a")).` for all $\texttt{a} \in \mathcal{A}$
- `precondition(action("a"),variable("f"),value(variable("f"),true)) :- action(action("a")).`
  for each $\texttt{a} \in \mathcal{A}$ and $\texttt{f} \in pre(\texttt{a})$
- `postcondition(action("a"),effect(unconditional),variable("f"), value(variable("f"),true)) :- action(action("a")).`
  for each $\texttt{a} \in \mathcal{A}$ and $\texttt{f} \in add(\texttt{a})$
- `postcondition(action("a"),effect(unconditional),variable("f"), value(variable("f"),false)) :- action(action("a")).`
  for each $\texttt{a} \in \mathcal{A}$ and $\texttt{f} \in del(\texttt{a})$

Example 1. The STRIPS domain with $\mathcal{F} = \{\texttt{f}\}$ and actions $\texttt{del-f} = \langle\{\texttt{f}\}, \emptyset, \{\texttt{f}\}\rangle$ and $\texttt{add-f} = \langle\emptyset, \{\texttt{f}\}, \emptyset\rangle$ is written in PDDL as follows:

```
(define (domain example1 )
(:requirements :strips)
(:predicates (f) )
(:action del-f
 :precondition (f)
 :effect (not (f)))
(:action add-f
 :effect (f)))
```

plasp translates this domain to the following set of rules (plus a few technical facts and rules):

```
variable(variable("f")).
action(action("del-f")).
precondition(action("del-f"), variable("f"),
             value(variable("f"), true))
  :- action(action("del-f")).
postcondition(action("del-f"), effect(unconditional),
              variable("f"), value(variable("f"), false))
  :- action(action("del-f")).
action(action("add-f")).
postcondition(action("add-f"), effect(unconditional),
              variable("f"), value(variable("f"), true))
  :- action(action("add-f")).
```

4.2 Reversibility Encodings in ASP

In this section, we present our ASP encoding for checking whether, in a given domain, there is an action that is uniformly reversible. As we have seen in Section 4.1, the *plasp* tool is able to rewrite STRIPS domains into ASP even when no concrete planning instance for that domain is given. We will present two encodings, one for (universal) uniform reversibility, and one that can be used for uniform φ-reversibility.

Note that *universal* uniform reversibility is computationally easier than φ-uniform reversibility (under standard complexity-theoretic assumptions). For a given action (and polynomial-length reverse plans), the former can be decided in NP, while the latter is harder [25, Theorem 18 and 20]. We will hence start with the encoding for the former problem, which follows a standard guess-and-check pattern.

Universal Uniform Reversibility. As a "database" the encoding takes the output of *plasp*'s translate action [13]. The problem can be solved in NP due to the following Observation (*): in any (universal) reverse plan for some action a, it is sufficient to consider only the set of facts that appear in the precondition of a. If any action in a candidate reverse plan π for a (resp. a itself) contains any other fact than those in $pre(a)$, then π cannot be a reverse plan for a (resp. a is not uniformly reversible) [25, Theorem 18]. With this observation in mind, we can now describe the (core parts of) our encoding[3].

The encoding makes use of the following main predicates (in addition to several auxiliary predicates, as well as those imported from *plasp*):

- `chosen/1` holds the action to be tested for reversibility.
- `holds/3` encodes that some fact (or variable, as they are called in *plasp* parlance) is set to a certain value at a given time step.
- `occurs/2` encodes the candidate reverse plan, saying which action occurs at which time step.

[3] The full encoding is available here: https://seafile.aau.at/d/e0aedc92b4c546d5bf9a/.

With the intuitive meaning of the predicates defined, firstly, we chose an action from the available actions and set the initial state as the facts in the precondition of the chosen action. We also say, in line with the Observation (*) above, that only those variables in the precondition are relevant to check for a reverse plan.

```
1 {chosen(A) : action(action(A))} 1.
holds(V, Val, 0) :-
  chosen(A),
  precondition(action(A), variable(V), value(variable(V), Val)).
relevant(V) :- holds(V, _, 0).
```

These rules set the stage for the inherent planning problem to be solved to find a reverse plan. In fact, from the initial state defined above, we need to find a plan π that starts with action a (the chosen action), such that after executing π we end up in the initial state again. Such a plan is a (universal) reverse plan. This idea is encoded in the following:

```
time(0..horizon+1).

occurs(A, 1) :- chosen(A).
1 {occurs(A, T) : action(action(A))} 1 :- time(T), T > 1.

caused(V, Val, T) :-
  occurs(A, T),
  postcondition(action(A), _, variable(V), value(variable(V), Val)),
  holds(V2, Val2, T - 1) :
  precondition(action(A), variable(V2), value(variable(V2), Val2)).
modified(V, T) :- caused(V, _, T).

holds(V, Val, T) :- caused(V, Val, T).
holds(V, Val, T) :- holds(V, Val, T - 1), not modified(V, T), time(T).
```

The above rules guess a potential plan π as described above, and then execute the plan on the initial state (changing facts if this is caused by the application of a rule, and keeping the same facts if they were not modified). The notation in the rule body for `caused` is an abbreviation for requiring `holds` for each `precondition`. Finally, we simply need to check that the plan is (a) executable, and (b) leads from the initial state back to the initial state. This can be done with the following constraints:

```
:- occurs(A, T),
   precondition(action(A), variable(V), value(variable(V), Val)),
   not holds(V, Val, T - 1).

:- occurs(A, T),
   precondition(action(A), variable(V), _),
   not relevant(V).
:- occurs(A, T),
   postcondition(action(A), _, variable(V), _),
   not relevant(V).
```

```
:- holds(V, Val, 0), not holds(V, Val, horizon+1).
:- holds(V, Val, horizon+1), not holds(V, Val, 0).
```

The first rule checks that rules in the candidate plan are actually applicable. The next two check that the rules do not contain any facts other than those that are relevant (cf. Observation (*) above). Finally, the last two rules make sure that at the maximum time point (i.e., the one given by the externally defined constant "horizon") the initial state and the resulting state of plan π are the same. It is not difficult to verify that any answer set of the above program (combined with the *plasp* translation of a STRIPS problem domain) will yield a plan π (encoded by the `occurs` predicate) that contains the sequence $a, a_1, \ldots, a_n$ of actions, where $a_1, \ldots, a_n$ is a (universal) reverse plan for the action a. Note that our encoding yields reverse plans of length exactly as long as set in the "horizon" constant. This completes our encoding for the problem of deciding universal uniform reversibility.

Other Forms of Uniform Reversibility. Using a similar guess-and-check idea as in the previous encoding, we can also check for uniform reversibility for a specified set of states (that is, uniform S-reversibility). Generally, the set S of relevant states is encoded in some compact form, and our encoding therefore, intentionally, does not assume anything about this representation, but leaves the precise checking of the set S open for implementations of a concrete use case. The predicates used in this more advanced encoding are similar to the ones used in the previous for the universal case above, and hence we will not list them here again. However, in order to encode the for-all-states check (i.e., the check that the candidate reverse plan works in *all* states inside the set S), we now need an advanced ASP encoding technique called *saturation* [15]. The saturation part occurs only towards the end.

The encoding starts off much like the previous one:

```
1 {chosen(A) : action(action(A))} 1.
holds(V, Val, 0) :-
  chosen(A),
  precondition(action(A), variable(V), value(variable(V), Val)).
affected(A, V) :- postcondition(action(A), _, variable(V), _).
```

Note that we no longer need to keep track of any set of "relevant" facts, since we now need to consider all the facts that appear inside the actions and in the set S of states. However, we need to keep track of those facts that are affected (i.e., potentially changed) by the application of an action. We assume that a predicate `opposites/2` exists that holds, in both possible orders, the values "true" and "false". This will later be used to find the opposite value of some fact at a particular time step.

Next, we again guess and execute a plan, keeping track of whether the actions were able to be applied at each particular time step:

```
occurs(A, 1) :- chosen(A).
```

```
1 {occurs(A, T) : action(action(A))} 1 :- time(T), T > 1.

applied(0). % no action needs to be applied at time step 0
applicable(A, T) :-
  occurs(A, T),
  applied(T - 1),
  holds(V, Val, T - 1) :
    precondition(action(A), variable(V), value(variable(V), Val)).
applied(T) :- applicable(_, T).
holds(V, Val, T) :-
  applicable(A, T),
  postcondition(action(A), _, variable(V), value(variable(V), Val)).
holds(V, Val, T) :-
  holds(V, Val, T - 1), occurs(A, T), applied(T), not affected(A, V).
```

Again, the rules above choose a candidate reverse plan π, starting with the action-to-be-checked a, as before. Furthermore, we set up the goal conditions: π should be applicable (i.e., at each time step, the relevant action must have been applied), and furthermore, the state at the beginning must be equal to the state at the end.

```
same(V) :- holds(V, Val, 0), holds(V, Val, horizon + 1).
samestate :- same(V) : variable(variable(V)).

planvalid :- applied(horizon + 1).

reversePlan :- samestate, planvalid.
```

Finally, we need to specify that for all the states specified in the set S the candidate reverse plan must work. This is done as follows:

```
holds(V, Val1, 0) | holds(V, Val2, 0) :-
  variable(variable(V)),
  opposites(Val1, Val2), Val1 < Val2.
holds(V, Val, T) :-
  reversePlan,
  contains(variable(V), value(variable(V), Val)),
  time(T).
:- not reversePlan.
```

As stated above, this is done using the technique of *saturation* [15]. We encourage the reader to refer to the relevant publication for more details on the "inner workings" of this encoding technique. In our case, intuitively, the rules state the following:

The first rule above specifies that some initial state should be guessed where the candidate reverse plan π is to be checked. The second and third rule together say that, for each such possible guess (i.e., for each possible initial state), the atom `reversePlan` must be derived for that particular guess. If it does, the second rule derives all possible `holds` atoms (it saturates the answer set on the predicate `holds`). If there should be a guess (initial state) that does not derive `reversePlan` (so the plan is not reversing for that initial state), then this saturation does not happen. In that case, the resulting answer set is a subset of all the saturated ones, and so it will form an answer set not containing `reversePlan`, violating the third rule. The third rule is therefore only satisfied if no such unsaturated case exists, and that means that the plan provides a uniform reversal.

This concludes the main part of our encoding. In its current form, the encoding given above produces exactly the same results as the first encoding given in this section; that is, it checks for *universal* uniform reversibility. However, the second encoding can be easily modified in order to check uniform S-reversibility. Simply add a rule of the following form to it:

```
reversePlan :- < check guess against set S >
```

This rule should derive the atom `reversePlan` precisely when the current guess (that is, the currently considered starting state) does not belong to the set S. This can of course be generalized easily. For example, if set S is given as a formula φ, then the rule should check whether the current guess conforms to formula φ (i.e., encodes a model of φ). Other compact representations of S can be similarly checked at this point. Hence, we have a flexible encoding for uniform S-reversibility that is easy to extend with various forms of representations of set S[4]. This concludes the description of our encodings.

4.3 Reversibility Encodings using ELP

In this section, we present ELP encodings, similar in spirit to the ASP encodings in Section 4.2. While the saturation encoding is arguably not easy to understand and also not easy to extend, we believe that the corresponding ELP encoding is better in both respects.

Universal Uniform Reversibility. The first encoding is very similar to the first encoding in Section 4.2, we just describe the differences[5]. Essentially, instead of multiple answer sets, we switch to multiple world views. For this, we change the choice rules to an "epistemic guess," here we show it for `occurs`, we do something similar for `chosen`.

```
occurs(A, T) :- action(action(A)),time(T), T > 1, not &k{-occurs(A, T)}.
-occurs(A, T) :- action(action(A)),time(T), T > 1, not &k{occurs(A, T)}.
:- occurs(A,T), occurs(B,T), A!=B.
oneoccurs(T) :- occurs(A,T), time(T), T > 0.
:- time(T), T>0, not oneoccurs(T).
```

Eventually, we check reversability by means of a subjective literal as well.

```
noreversal :- holds(V, Val, 0), not holds(V, Val, H+1), horizon(H).
noreversal :- holds(V, Val, H+1), not holds(V, Val, 0), horizon(H).
:- not &k{ ~ noreversal}.
```

[4] The full encoding can be found at https://seafile.aau.at/d/e0aedc92b4c546d5bf9a/.
[5] The full encoding is available here: https://seafile.aau.at/d/373cd25718dc4377afec/.

Other Forms of Uniform Reversibility. Using ELP, we can avoid the need for the saturation technique that we employed in Section 4.2.

After an unique epistemic guess for `chosen` and making sure that the precondition of the chosen action is met in the initial state, we create the initial states by means of a non-epistemic guess:

```
holds(V,Val,0) | -holds(V,Val,0) :-
  variable(variable(V)), contains(variable(V),value(variable(V),Val)).

oneholds(V,0) :- holds(V,Val,0).
:- variable(variable(V)), not oneholds(V,0).
:- holds(V,Val,0), holds(V,Val1,0), Val != Val1.
```

The reverse plan is guessed as before, applicability is checked by disallowing the possibility of inapplicability.

```
inapplicable :-
  occurs(A, T),
  precondition(action(A), variable(V), value(variable(V), Val)),
  not holds(V, Val, T - 1).
:- not &k{ ~ inapplicable}.
```

Reversibility is checked as in the previous encoding.

For checking uniform S-reversibility, one can simply add a constraint:

```
:- < check guessed state against set S >
```

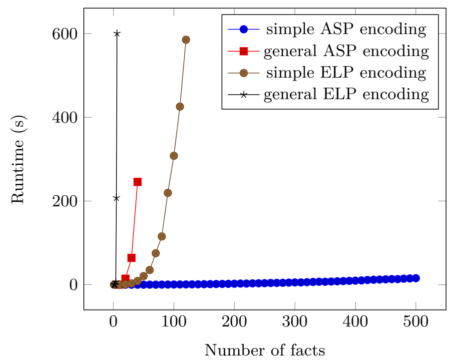
5 Experiments

We have conducted preliminary experiments with artificially constructed domains. The domains are as follows:

```
(define (domain rev-i)
(:requirements :strips)
(:predicates (f0) ... (fi))

(:action del-all
 :precondition (and  (f0) ... (fi) )
 :effect (and  (not (f0)) ... (not (fi))))

(:action add-f0
 :effect (f0))

...

(:action add-fi
 :precondition (fi-1)
 :effect (fi)))
```

The action `del-all` has a universal uniform reverse plan ⟨ `add-f0`, ..., `add-fi` ⟩. We have generated instances from `i` = 1 to `i` = 6 and from `i` = 10 to `i` = 200 with step 10. We have analyzed runtime and memory consumption of two problems: (a) finding the reverse plan of size `i` (by setting the constant `horizon` to `i`) and proving that no other reverse plan exists, and (b) showing that no reverse plan of length `i-1` exists (by setting the constant `horizon` to `i-1`). We compare the encodings described in Sections 4.2 and 4.3, we refer to the respective first ones as *simple encodings* and the second ones as *general encodings*.

We have used plasp 3.1.1 (https://potassco.org/labs/plasp/), clingo 5.4.0 (https://potassco.org/clingo/), and eclingo 0.2.0 (https://github.com/potassco/eclingo) [4] on a computer with a 2.3 GHz AMD EPYC 7601 CPU with 32 cores and 500 GB RAM running CentOS 8. We have set a timeout of 10 minutes and a memory limit of 4GB.



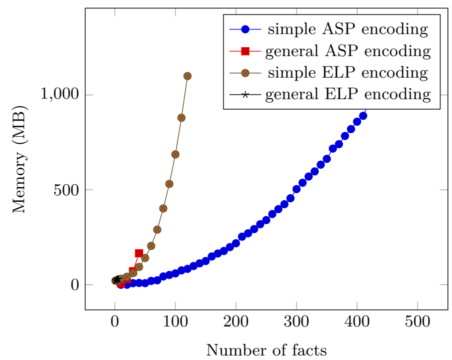


Fig. 1. Calculating the single reverse plan (plan length equals number of facts)

The results for problem (a) are plotted in Figure 1. It is clear that the simple encodings perform much better than the general ones (which was expected), but it is also clear that the ELP encodings perform worse than the ASP ones. This was expected for the simple encoding, as the ELP variant (unnecessarily) uses world views instead of answer sets to encode the same thing. However, the discrepancy in runtime is quite significant (to a lesser degree also for memory consumption). We hoped that the general encoding would show less difference in runtime, however, also in that case the ELP encoding performs clearly worse. The general ELP encoding exceeded the time limit already at the problem with six facts, while the general ASP encoding can solve up to 50 facts. The simple ASP encoding could solve all tested problems in under 20 seconds, while the simple ELP encoding could solve all problems with up to 120 facts within the time limit. The memory consumption increased with `i` for both encodings, proportional to the computation time.

The results for problem (b) are plotted in Figure 2.

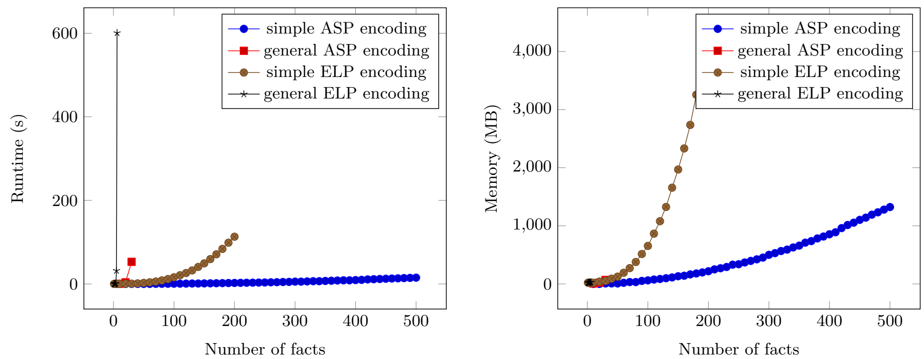


Fig. 2. Determining nonexistence of a reverse plan (plan length equals number of facts minus one)

While the simple ASP encoding shows very similar behavior to problem (a), the general ASP encoding took longer and had a time-out already at $\mathtt{i} = 40$. Memory consumption appears to be similar to (a) for the ASP encodings.

Interestingly, compared to (a), both the general and the simple ELP encodings ran noticably faster, in contrast to the ASP encodings. While the general ELP encoding still hit the time limit for six facts, the simple encoding was able to solve all the instances up to $\mathtt{i} = 200$, but at the expense of increasing memory usage.

The general ELP encoding scales worse, as expected, since the ELP solver needs to evaluate all answer sets inside each possible world view. However, for the simple encoding, especially the task of testing for non-reversibility performed surprisingly well.

From these results, however, we can see that ELP solving is still severely behind ASP solving, in terms of performance.

6 Conclusions

In this paper, we have given a review of several notions of action reversibility in STRIPS planning, as originally presented in [25]. We then proceeded, on the basis of the PDDL-to-ASP translation tool *plasp* [13], to present two ASP and two ELP encodings to solve the task of universal uniform reversibility of STRIPS actions, given a corresponding planning domain.

The encodings use two different approaches. The first makes use of a shortcut that allows it to focus only on those facts that appear in the precondition of the action to check for reversibility [25], this is very similar for ASP and ELP. The second encoding uses a saturation technique for ASP, and the power of world views (as "groupings" of answer sets), which allows for the expression of universal quantifiers via the **K** operator, for ELP. The latter encodes the original definition of uniform reversibility: for an action to be uniformly reversible, a plan must exist, and this plan must revert that action in all possible starting states (where it is applicable). This second encoding is more flexible insofar as it also allows for the checking of non-universal uniform reversibility (e.g. to check for uniform φ-reversibility, where the starting states are given via some formula φ).

In order to compare the encodings, we performed some benchmarks on artificially generated instances by checking whether there is an action that is universally uniformly reversible. It does not come as a surprise that the general encodings were performing much more poorly than the simple encodings. For such encodings to become practical, both ASP and ELP solvers need to be further optimized. However, the simple encodings showed promise, especially for plain ASP. The ELP encodings performed somewhat worse, but of course the ELP solver used is experimental, while the ASP solver is highly optimized. Some behaviors of the ELP solver are interesting, for example it showed better performance for the problem without solution (which was unexpected), but it also required more memory with increasing problem sizes, compared to the ASP encoding.

For future work, we intend to optimize our encodings further, and test them with other established ASP and ELP solvers, notably *DLV2*[6] [1]. t would also be interesting to see how they perform when compared to a procedural implementation of the algorithms proposed for reversibility checking by Morak et al. [25]. We would also like to compare our approach to existing tools *RevPlan*[7] (implementing techniques of [14]) and *undoability* (implementing techniques of [11]).

References

1. Alviano, M., Calimeri, F., Dodaro, C., Fuscà, D., Leone, N., Perri, S., Ricca, F., Veltri, P., Zangari, J.: The ASP system DLV2. In: Balduccini, M., Janhunen, T. (eds.) Logic Programming and Nonmonotonic Reasoning - 14th International Conference, LPNMR 2017, Espoo, Finland, July 3-6, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10377, pp. 215–221. Springer (2017). https://doi.org/10.1007/978-3-319-61660-5_19
2. Brewka, G., Eiter, T., Truszczynski, M.: Answer set programming at a glance. Commun. ACM **54**(12), 92–103 (2011). https://doi.org/10.1145/2043174.2043195
3. Bylander, T.: The computational complexity of propositional STRIPS planning. Artif. Intell. **69**(1-2), 165–204 (1994). https://doi.org/10.1016/0004-3702(94)90081-7
4. Cabalar, P., Fandinno, J., Garea, J., Romero, J., Schaub, T.: eclingo: A solver for epistemic logic programs. CoRR **abs/2008.02018** (2020)
5. Calimeri, F., Faber, W., Gebser, M., Ianni, G., Kaminski, R., Krennwallner, T., Leone, N., Maratea, M., Ricca, F., Schaub, T.: Asp-core-2 input language format. Theory Pract. Log. Program. **20**(2), 294–309 (2020). https://doi.org/10.1017/S1471068419000450

[6] https://www.mat.unical.it/DLV2/

[7] http://www.kr.tuwien.ac.at/research/systems/revplan/index.html

6. Camacho, A., Muise, C.J., McIlraith, S.A.: From FOND to robust probabilistic planning: Computing compact policies that bypass avoidable deadends. In: Proc. ICAPS. pp. 65–69 (2016), http://www.aaai.org/ocs/index.php/ICAPS/ICAPS16/paper/view/13188
7. Chrpa, L., Faber, W., Fišer, D., Morak, M.: Determining action reversibility in STRIPS using answer set programming. In: Dodaro, C., Elder, G.A., Faber, W., Fandinno, J., Gebser, M., Hecher, M., LeBlanc, E., Morak, M., Zangari, J. (eds.) International Conference on Logic Programming 2020 Workshops. CEUR Workshop Proceedings, vol. 2678. CEUR-WS.org (9 2020), http://ceur-ws.org/Vol-2678/paper2.pdf
8. Chrpa, L., Lipovetzky, N., Sardiña, S.: Handling non-local dead-ends in agent planning programs. In: Proc. IJCAI. pp. 971–978 (2017). https://doi.org/10.24963/ijcai.2017/135
9. Chrpa, L., McCluskey, T.L., Osborne, H.: Optimizing plans through analysis of action dependencies and independencies. In: Proc. ICAPS (2012), http://www.aaai.org/ocs/index.php/ICAPS/ICAPS12/paper/view/4712
10. Cserna, B., Doyle, W.J., Ramsdell, J.S., Ruml, W.: Avoiding dead ends in real-time heuristic search. In: Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18). pp. 1306–1313 (2018)
11. Daum, J., Torralba, Á., Hoffmann, J., Haslum, P., Weber, I.: Practical undoability checking via contingent planning. In: Proc. ICAPS. pp. 106–114 (2016), http://www.aaai.org/ocs/index.php/ICAPS/ICAPS16/paper/view/13091
12. De Giacomo, G., Gerevini, A.E., Patrizi, F., Saetti, A., Sardiña, S.: Agent planning programs. Artif. Intell. **231**, 64–106 (2016). https://doi.org/10.1016/j.artint.2015.10.001
13. Dimopoulos, Y., Gebser, M., Lühne, P., Romero, J., Schaub, T.: plasp 3: Towards effective ASP planning. Theory and Practice of Logic Programming **19**(3), 477–504 (2019). https://doi.org/10.1017/S1471068418000583
14. Eiter, T., Erdem, E., Faber, W.: Undoing the effects of action sequences. J. Applied Logic **6**(3), 380–415 (2008). https://doi.org/10.1016/j.jal.2007.05.002
15. Eiter, T., Gottlob, G.: On the computational cost of disjunctive logic programming: Propositional case. Ann. Math. Artif. Intell. **15**(3-4), 289–323 (1995). https://doi.org/10.1007/BF01536399, https://doi.org/10.1007/BF01536399
16. Faber, W., Morak, M.: Determining action reversibility in STRIPS using epistemic logic programs. In: Dodaro, C., Elder, G.A., Faber, W., Fandinno, J., Gebser, M., Hecher, M., LeBlanc, E., Morak, M., Zangari, J. (eds.) International Conference on Logic Programming 2020 Workshops. CEUR Workshop Proceedings, vol. 2678. CEUR-WS.org (9 2020), http://ceur-ws.org/Vol-2678/paper10.pdf
17. Fikes, R., Nilsson, N.J.: STRIPS: A new approach to the application of theorem proving to problem solving. Artif. Intell. **2**(3/4), 189–208 (1971). https://doi.org/10.1016/0004-3702(71)90010-5, http://dx.doi.org/10.1016/0004-3702(71)90010-5
18. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Answer Set Solving in Practice. Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan & Claypool Publishers (2012). https://doi.org/10.2200/S00457ED1V01Y201211AIM019
19. Gelfond, M.: Strong introspection. In: Dean, T.L., McKeown, K.R. (eds.) Proc. AAAI. pp. 386–391. AAAI Press / The MIT Press (1991), http://www.aaai.org/Library/AAAI/1991/aaai91-060.php

20. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Gener. Comput. **9**(3/4), 365–386 (1991). https://doi.org/10.1007/BF03037169
21. Ghallab, M., Nau, D.S., Traverso, P.: Automated planning - theory and practice. Elsevier (2004)
22. Ghallab, M., Nau, D.S., Traverso, P.: Automated Planning and Acting. Cambridge University Press (2016), http://www.cambridge.org/de/academic/subjects/computer-science/artificial-intelligence-and-natural-language-processing/automated-planning-and-acting?format=HB
23. Lifschitz, V.: Answer Set Programming. Springer (2019). https://doi.org/10.1007/978-3-030-24658-7, https://doi.org/10.1007/978-3-030-24658-7
24. Lipovetzky, N., Muise, C.J., Geffner, H.: Traps, invariants, and dead-ends. In: Proc. ICAPS. pp. 211–215 (2016), http://www.aaai.org/ocs/index.php/ICAPS/ICAPS16/paper/view/13190
25. Morak, M., Chrpa, L., Faber, W., Fišer, D.: On the reversibility of actions in planning. In: Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning (KR 2020) (2020), to appear.
26. Truszczynski, M.: Revisiting epistemic specifications. In: Balduccini, M., Son, T.C. (eds.) Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning - Essays Dedicated to Michael Gelfond on the Occasion of His 65th Birthday. Lecture Notes in Computer Science, vol. 6565, pp. 315–333. Springer (2011). https://doi.org/10.1007/978-3-642-20832-4_20