Logic-based Languages For Solving Reversibility in Planning

Wolfgang Faber[0000−0002−0330−5868] and Michael Morak[0000−0002−2077−7672]

University of Klagenfurt, Austria
{wolfgang.faber,michael.morak}@aau.at
https://www.aau.at/

Abstract. In this paper, we compare and contrast existing and novel encodings of the Action Reversibility problem in STRIPS planning, using several logic programming languages. Action Reversibility asks whether the effects of an action can be undone so that the original state is restored. We encode this problem using the formalisms of Answer Set Programming (ASP), ASP with Quantifiers, Quantified ASP, and the Bule QBF programming language. We implement the encodings and perform an experimental evaluation, comparing the performance of these different approaches and associated solvers.

1 Introduction

Automated Planning is a field of AI research that traditionally deals with the problem of generating sequences of actions, called plans, that transform a given initial state of the environment to some goal state [23, 24]. An action, simply put, is a modifier that acts upon and changes the environment. An interesting problem in this field is the question whether an action can be reversed by subsequently applying other actions, thus undoing the effects that the action had on the environment. This problem has been investigated on and off throughout the years [13, 10], and has recently received renewed interest [32, 15, 6, 33, 31].

Action reversibility is an important problem with regard to several aspects. Intuitively, actions whose effects cannot be reversed might lead to dead-end states, that is, states from which the goal state can no longer be reached. Early detection of the possibility of dead-end states is beneficial in the plan generation process [29]. Reasoning in more complex structures is even more prone to dead-ends [7]. An example is the concept of Agent Planning Programs [11], which represent networks of planning tasks where the goal state of one task is an initial state of some other task. Another aspect is online planning, where we can observe that applying reversible actions is safe and hence explicitly providing information about safe states of the environment could be avoided [9]. 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 given circumstances, pairs of inverse actions, which are a special case of action reversibility, can be removed from plans [8].

The general framework for action reversibility introduced in [32] offers a broad definition of the term and generalises existing notions of reversibility, like *undoability* [10], or *reverse plans* [13]. The concept of S-reversibility in this general framework directly incorporates the set of states in which a given action is reversible: S is a set of states where an action must be reversible. In this paper, we deal with universal reversibility, where S is the set of all states.

Membership in Π_2^P of reversibility for polynomial-length plans [32] indicates that some of these problems can be addressed by means of Answer Set Programming (ASP), which can uniformly encode all Σ_2^P-complete problems [14], and related formalisms. For the restricted case of *uniform* reversibility (where a single "reverse plan" must revert the action irrespective of the starting state) this has been done using ASP itself and Epistemic Logic Programs (ELPs) [15], also with ASP with Quantifiers ASP(Q) in [16] and work has started to compile a reasonably broad benchmark set for uniform reversibility [33]. However, general (non-uniform) reversibility has, to our knowledge, not yet been addressed.

Contributions. In this paper, we encode reversibility for STRIPS domains using various formalisms: ASP, ELP, ASP(Q) [1], Quantified ASP (QASP) [17], Bule [26]. All encodings rely on the `plasp` tool [12] to translate PDDL domains into ASP facts and then encode the action reversibility problem. ASP(Q), QASP, and Bule can express all problems in the polynomial hierarchy and are therefore suitable for checking Π_2^P-complete reversibility. For ASP and ELP, the complement problem (finding a counterexample, if one exists) has to be encoded. Finally, we perform preliminary experiments on established benchmarks to compare our encodings in pratice.

2 Preliminaries

STRIPS Planning. Let $\mathcal{F}$ be a set of *facts*, that is, propositional variables describing the environment, which can either be true or false. 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 four-element tuple consisting of a set of *facts* $\mathcal{F} = \{f_1, \ldots, f_n\}$, a set of *actions* $\mathcal{A} = \{a_1, \ldots, a_m\}$, an *initial* *state* $s_0 \subseteq \mathcal{F}$, and a *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 $\pi[s_0] \supseteq G$. 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].

Reversibility of Actions. We review the notion of reversibility for STRIPS actions [32]. Intuitively, we call an action *reversible* if there is a way to undo all the effects that this action caused.

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

The notion of 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 S-reversibility. In this paper, we will deal with the notion of *universal reversibility*, that is, where $S = 2^{\mathcal{F}}$[1]. Several refined notions of reversibility exist [32]. In particular, the notion of *uniform reversibility* requires that the sequence of actions used to reverse the given action a is always the same, independent of the starting state $s \in S$. Hence, the general notion of reversibility is sometimes referred to as *non-uniform* reversibility, to highlight this distinction. Even if the length of the reverse plan is polynomially bounded, the problem of deciding whether an action is reversible is intractable; it is located on or slightly beneath the second level of the polynomial hierarchy [32].

Answer Set Programming (ASP). We assume the reader is familiar with ASP and will only give a very brief overview of the core language. For more information, we refer to standard literature [2, 18, 28], and, in our case, the ASP-Core-2 input language format [5].

Briefly, 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 and $t_1, \ldots, t_n$ are terms, that is, either variables or constants. *Literals* are negated or unnegated atoms. The domain of constants in an ASP program P is given implicitly by the set of all constants that appear in it. Generally, before evaluating an ASP program, variables are removed by a process called *grounding*, that is, for every rule, each variable is replaced by all possible combination of constants, and appropriate ground copies of the rule are added to the resulting program *ground*(P). In practice, several optimizations have been implemented in state-of-the-art grounders that try to minimize the size of the grounding.

[1] We note that, in pratice, universal reversibility may not be widely applicable. However, in all our encodings, it would be easy to add additional constraints that limit set S to a desired subset of $2^{\mathcal{F}}$.

The result of a (ground) ASP program P is calculated as follows [22]. 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 (i.e. whether it is satisfiable). This has been shown to be Σ_2^P-complete [14] for ground programs.

Epistemic Logic Programs (ELPs). Extensions to the ASP language with epistemic operators have been proposed, which we only briefly mention in this work. We use the classical G94 semantics [20, 21].

Quantified Answer Set Programming (QASP). An extension of ASP with quantified atoms, called QASP has been proposed recently [17], providing a formalism reminiscent of quantified Boolean formulas (QBFs), but based on ASP. A *QASP program* is of the form

$$Q_1 X_1 Q_2 X_2 \cdots Q_n X_n P$$

where P is an ASP program, and for each *quantifier level* $i \in \{1, \dots, n\}$, $Q_i \in \{\exists, \forall\}$ and X_i is a set of atoms from P. Atoms in X_i are, therefore, (existentially or universally) *quantified at level* i.

The intuitive reading of a QASP program $\exists X_1 \forall X_2 P$ is that there exists a subset $Y_1 \subseteq X_1$ of atoms such that for all subsets $Y_2 \subseteq X_2$ there exists an answer set M of P such that exactly the atoms Y_1 out of X_1 and Y_2 out of X_2 are true in M.

Formally, this is done via a recursive definition, where the function $\mathit{fixcons}(Y, X)$ represents a set of ASP constraints that enforce that the atoms Y need to be true and all other atoms in X false: $\mathit{fixcons}(Y, X) = \{\bot \leftarrow \neg x \mid x \in Y\} \cup \{\bot \leftarrow x \mid x \in X \setminus Y\}$. Then satisfiability of a QASP program is defined as follows:

- $\exists X\, P$ is satisfiable if $P \cup \mathit{fixcons}(Y, X)$ has at least one answer set for some $Y \subseteq X$.
- $\forall X\, P$ is satisfiable if $P \cup \mathit{fixcons}(Y, X)$ has at least one answer set for each $Y \subseteq X$.
- $\exists X\, \mathcal{Q}\, P$ is satisfiable if $\mathcal{Q}\, (P \cup \mathit{fixcons}(Y, X))$ has at least one answer set for some $Y \subseteq X$.
- $\forall X\, \mathcal{Q}\, P$ is satisfiable if $\mathcal{Q}\, (P \cup \mathit{fixcons}(Y, X))$ has at least one answer set for each $Y \subseteq X$.

Deciding satisfiability for a QASP program is known to be PSpace-complete [17, Theorem 5.2].

ASP with Quantifiers (ASP(Q)). Another extension of ASP, referred to as ASP(Q), has been proposed [1], also providing a formalism reminiscent of Quantified Boolean Formulas, but based on ASP and a different kind of quantifiers. We use existing ASP(Q) encodings in our benchmark section.

The Bule Language. The *Bule* modelling language [26] was created to solve problems in PSPACE and is based on quantified Boolean formulas (QBFs) enhanced with a non-ground, rule-like syntax that allows for Datalog-inspired problem encodings that separate the problem domain and concrete problem instance, called intensional and extensional programs, respectively. To this end, the set of predicate symbols is divided into extensional and intensional predicates. Atoms formed by these predicates are called extensional and intensional atoms, respectively. Extensional programs consist of rules of the form

$$L_1, \ldots, L_k :: H,$$

where H is an extensional atom and L_i $(0 < i \leqslant k)$ are extensional (non-ground) literals, where atoms are defined as in ASP. Variables must appear in at least one non-negated literal on the left-hand side of the rule. Intensional programs consist of rules of the form

$$L_1, \ldots, L_k :: Q[T]H$$

or

$$L_1, \ldots, L_k :: C_1, \ldots C_n,$$

where H is an intensional atom, $Q \in \{\exists, \forall\}$ and T is a non-negative integer. Each C_i $(0 < i \leqslant n)$ is a *conditional literal* of the form $M_1, \ldots, M_m : B$, where $M_1, \ldots, M_m$ are extensional literals and B is an intensional literal. The first type of rule serves as a variable declaration for existentially or universally quantified variables at some level T. The second type of rule generates clauses.

A Bule program $P = (P_{ext}, P_{int})$ consists of an extensional and an intensional program, containing only the respective types of rules. Note that P_{ext} therefore only contains extensional rules, which are in fact plain positive Datalog rules (when viewing :: as $\rightarrow$). Hence, given a set of ground facts $\mathcal{D}$, let $P_{ext}(\mathcal{D})$ be the unique minimal model of the logical theory $P_{ext} \cup \mathcal{D}$, i.e. its Datalog extension.

The semantics of a Bule program executed on a set of input ground facts $\mathcal{D}$ is now defined via a QBF formula $\exists \mathbf{x}_1 \forall \mathbf{y}_1 \ldots \exists \mathbf{x}_n . \varphi$, generated as follows:

Let θ be a homomorphism mapping variables to constants. For each such homomorphism θ, whenever $\theta(L_i) \subseteq P_{ext}(\mathcal{D})$ for each $0 < i \leqslant k$ in a rule $L_1, \ldots, L_k :: Q[T]H$, variable $\theta(H) \in \mathbf{x}_T$ if $Q = \exists$ or $\theta(H) \in \mathbf{y}_T$ if $Q = \forall$. Furthermore, for each homomorphism θ, whenever $\theta(L_i) \subseteq P_{ext}(\mathcal{D})$ for each $0 < i \leqslant k$ in a rule $L_1, \ldots, L_k :: C_1, \ldots C_n$ the QBF formula φ contains the clause $\bigvee_{0<i\leqslant n} \bigvee_{\ell \in \hat{C}_i} \ell$, where

$$\hat{C}_i = \bigcup_{\substack{\theta' \\ \forall j,\, 0<j\leqslant m: \\ \theta'(\theta(M_j)) \subseteq P_{ext}(\mathcal{D})}} \theta'(\theta(B)),$$

that is, the set of ground literals obtained from B for each homomorphism θ' extending θ and mapping all M_j to the extensional program $P_{ext(\mathcal{D})}$.

Intuitively, first the (positive) extensional program part is evaluated. Based on the ground facts so derived, each ground instance of a variable declaration rule generates an appropriately quantified Boolean variable. Each ground instance of the body of a clause generating rule generates a clause with one literal for each ground instance of a conditional literal.

Example 1. An example Bule encoding for the well-known graph colouring problem can be formulated as follows[2]:

```
col[C1], col[C2], C1 != C2 :: #ground ecol[t(C1,C2)].
edg[V1, V2], ecol[T] :: #exists set(V1,V2,T).

edg[V1,V2], edg[V1,V3], V2 < V3,
    ecol[t(C1,C2)], ecol[t(C1P,C3)], C1 != C1P
    :: ~set(V1,V2,t(C1,C2)) | ~set(V1,V3,t(C1P,C3)).
edg[V2,V1], edg[V3,V1], V2 < V3,
    ecol[t(C2,C1)], ecol[t(C3, C1P)], C1 != C1P
    :: ~set(V2,V1,t(C2,C1)) | ~set(V3,V1,t(C3,C1P)).
edg[V2,V1], edg[V1,V3],
    ecol[t(C2,C1)], ecol[t(C1P, C3)], C1 != C1P
    :: ~set(V2,V1,t(C2,C1)) | ~set(V1,V3,t(C1P,C3)).

edg[V1,V2] :: ecol[T] : set(V1,V2,T).
edg[V1,V2], ecol[T1], ecol[T2], T1 < T2
    :: ~set(V1,V2,T1) | ~set(V1,V2,T2).
```

3 ASP-based Reversibility Encodings

In this section, we propose several encodings for (non-uniform) universal reversibility of STRIPS actions using various ASP-based formalisms. These make use of the *plasp* translator transforms planning tasks given in the widely-used PDDL language into ASP databases. We start with a brief review of the syntax of this tool.

3.1 The *plasp* Format

The *plasp* system [12] 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 rules will be created by *plasp*:

`variable(variable("f")).` for all `f` $\in \mathcal{F}$
`action(action("a")).` for all `a` $\in \mathcal{A}$
`precondition(action("a"),variable("f"),value(`
  `variable("f"),true)) :- action(action("a")).`
  for each `a` $\in \mathcal{A}$ and `f` $\in$ *pre*(a)

[2] Taken from `https://github.com/vale1410/bule/`, ~ denotes $\neg$.

```
postcondition(action("a"),_,variable("f"),value(
   variable("f"),true)) :- action(action("a")).
```
for each $\mathtt{a} \in \mathcal{A}$ and $\mathtt{f} \in \mathit{add}(\mathrm{a})$
```
postcondition(action("a"),_,variable("f"),
   value(variable("f"),false)) :- action(action("a")).
```
for each $\mathtt{a} \in \mathcal{A}$ and $\mathtt{f} \in \mathit{del}(\mathrm{a})$

In addition, a predicate `contains` encodes all possible values for a given variable (for STRIPS: values true or false).

3.2 Reversibility in ASP

We first sketch our ASP encoding, which will form the basis for the encodings of the other ASP-based formalisms. It takes a time horizon and a chosen action as input, in the form `horizon(n)` and `chosen(action-name)`, respectively, and uses the *plasp*-generated database as input facts. The encoding is based on an existing encoding for the uniform reversibility task [15] and re-uses the core idea, which itself is based on the *sequential-horizon* encoding for STRIPS planning, which is supplied as part of *plasp*[3].

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

– `chosen/1` encodes the action to be reversibility-tested.
– `holds/3` encodes that some fact (or "variable" 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.

With the meaning of our main predicates defined, we describe the main parts of our ASP encoding. Reversibility requires that for all states there exists a sequence of actions that reverses the chosen action in that state. In ASP, for complexity-theoretic reasons, we need to model the converse: there exists a state, such that no plan exists that reverses the action in this state. Hence, if (and only if) the chosen action is reversible, our encoding for this action will be unsatisfiable. The first step, hence, is to guess a starting state.

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

The two rules above guess a random starting state (via the `holds/3` predicate at time point 0) that satisfies the precondition of the chosen action. We now need to verify, based on this starting state, that there is no sequence of actions that can be applied after the chosen action, so that we return to the starting state. In

[3] See `https://github.com/potassco/plasp/tree/master/encodings`.

ASP, this can be done using a technique called *saturation*, first used to show Σ_2^P-completeness of answer set existence [14]. With this, we can perform a universal check. In our case, we will therefore test that, whatever sequence of actions we choose, it does not revert the chosen action. First, we set up our planning task with the input horizon, and set the first action to be the chosen action.

```
time(0..H+1) :- horizon(H).
occurs(A, 1) :- chosen(A).
applied(0).
```

We are now ready to make sure that no sequence of actions that start with the chosen action can ever reach the origin state again. The first rule in the following guess a sequence of actions, and the second and third rule together set up the *saturation* technique, and can be read as follows: verify that `noReversal` is derived for *every* sequence of actions guessed by the first rule.

```
occurs(A, T) : action(action(A)) :- time(T), T > 1.
occurs(A, T) :- noReversal, action(action(A)),
    time(T), T > 1.
:- not noReversal.
```

The remaining rules in the saturation part of our encoding derive the states which are reached after each action is executed. Here, `affected/2` contains those facts `V` which are modified by some action `A`. The last rule encodes inertia, that is, facts that are not affected should remain the same after applying an action:

```
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).
```

Finally, we need to check that the sequence of actions actually does not revert the chosen action. This is the case under two conditions: (1) the sequence of actions is not applicable; or (2) the final state differs from the original state in at least one fact. These two checks are performed by the following two rules:

```
noReversal :- holds(V, Val1, 0), holds(V, Val2, H+1),
    horizon(H), Val1 != Val2.
noReversal :- occurs(A, T), applied(T - 1),
    holds(V, Val1, T - 1), precondition(action(A),
    variable(V), value(variable(V), Val2)), Val1 != Val2.
```

This completes the encoding. Correctness follows by construction:

Theorem 1. *Given a STRIPS planning task $\Pi = \langle \mathcal{F}, \mathcal{A}, s_0, G \rangle$ and chosen action $a \in \mathcal{A}$, the ASP encoding above, for Π, has exactly one answer set per state $s \in 2^{\mathcal{F}}$ where a is not $\{s\}$-reversible.*

Proof (Proof (Sketch).). The rules deriving the `holds` predicate at time point 0 ensure that there is one candidate answer set for each possible starting state that satisfies the preconditions of a.

The rule guessing the `occurs` predicate then guesses a sequence of actions to revert the action a which is applied first. The rules deriving the `holds` and `applicable` predicates then execute action a and the sequence of actions, keeping track of which value each variable has after each step (represented by time points `T`).

Finally, `noReversal` is derived if either any action in the sequence is not applicable at its time point, or if the result of executing the sequence of actions is not the original state (i.e. comparing time point 0 and $H+1$, where H is the time horizon). Now, the two saturation rules guarantee the following: The only answer set that can exist for a given guessed starting state is one where `noReversal` is true and hence one that contains all `occurs` facts. However, `noReversal` needs to be true, hence all possible sequences of actions need to derive it. Otherwise, there would be a model of the reduct of the ASP program that would only contain the succeeding sequence of actions, and not the fact `noReversal`, a contradiction to the fact that each answer set must be subset-minimal w.r.t. its reduct. Hence, we have that there exists exactly one answer set per starting state where the chosen action is not reversible.

3.3 Reversibility via ELPs

Since the **K** operator provides a limited form of universal quantification, we can modify the ASP encoding provided above as follows, in order to make use of this feature and thereby simplify the construction, where `&k{...}` represents the **K** operator. The idea is that there should be one world view (i.e. set of answer sets) for each state where the action is not reversible. Within this world view, each answer set represents a sequence of actions, and in each of these answer sets, the fact `noReversal` must be true, that is, **K** *noReversal* must hold.

Firstly, instead of guessing a state in one answer set, we now guess a set of answer sets (i.e. a possible world view), one per state.

```
holds(V, Val1, 0) :-
    variable(variable(V)),
    contains(variable(V), value(variable(V), Val1)),
    contains(variable(V), value(variable(V), Val2)),
    Val1 != Val2,
    not &k{holds(V, Val2, 0)}.
:- holds(V, Val1, 0), holds(V, Val2, 0), Val1 < Val2.
```

Instead of the hard-to-read saturation technique of ASP, we can now simply guess an action sequence of actions for the reversal:

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

Finally, we make sure that the fact `noReversal` needs to be derived in every answer set of the respective world view:

```
:- not &k{noReversal}.
```

All remaining rules in the ASP encoding from Section 3.2 remain. Adapted in this way, this results in an epistemic logic program (ELP) for which there exists exactly one world view per state s where the chosen action is not $\{s\}$-reversible. This can be seen via a similar argument as for the ASP encoding (cf. Theorem 1).

3.4 Reversibility in QASP

The formalism of Quantified ASP (QASP) allows for explicit quantification over the atoms in an answer set program. We can make use of this feature by again replacing the saturation part of the ASP encoding by this explicit quantification. However, since we can now explicitly quantify over atoms, we no longer need to encode the converse of action reversibility, as in ASP. Instead, we can now directly encode reversibility as defined in Definition 1: Verify that for all states, there exists an action sequence that reverts the chosen action.

Again we describe the relevant modifications to the ASP encoding from Section 3.2. Firstly, instead of guessing a single state via the `holds` predicate at time point 0, we universally quantify over the guessed state as follows:

```
_forall(2, holds(V,Val,0)) :- contains(variable(V),
    value(variable(V),Val)).
{holds(V,Val,0)} :- contains(variable(V),
    value(variable(V),Val)).
fail :- #count{Val:holds(V,Val,0)} != 1,
    variable(variable(V)).
fail :- not holds(V, Val, 0), chosen(A),
    precondition(action(A), variable(V),
    value(variable(V), Val)).
```

The universal quantification is done via a special predicate `_forall`, with the first position representing the quantification level (universal quantification in QASP needs to have an even level, hence we start at level 2). Note that in order to establish that the starting state needs to satisfy the preconditions of the chosen action, we cannot simply write a constraint, since there must be an answer set for all combinations of universally quantified facts. Hence, the atom `fail` is derived in this case and will be checked later.

Now, in order to guess a sequence of actions and verify that it reverses the chosen action, we existentially quantify over guessed the sequence of actions, using the special `_exists` predicate, as follows:

```
_exists(3, occurs(A, T)) :- action(action(A)),
    time(T), T > 1.
{occurs(A, T)} :- action(action(A)), time(T), T > 1.
:- #count{A:occurs(A, T)}!=1,time(T), T > 1.
```

Hence, we now have that for all states guessed above, there must exist a sequence of actions. The effects of these actions are determined as in the ASP encoding. However, successful reversal can now be simply checked using plain ASP constraints instead of having to employ the saturation technique, as in ASP:

```
:- not fail, holds(V, Val, 0),
   not holds(V, Val, H+1), horizon(H).
:- not fail, holds(V, Val, H+1),
   not holds(V, Val, 0), horizon(H).
:- not fail, occurs(A, T),
   precondition(action(A), variable(V),
   value(variable(V), Val)),
   not holds(V, Val, T - 1).
```

The constraints above verify that the sequence of actions actually revert to the original starting state and are applicable. However, if the starting state is invalid (i.e. when `fail` is true), then the constraints are disabled. This completes the adaptation of the ASP encoding to QASP. The correctness argument is similar the the one in Theorem 1, but in reverse. The QASP program is satisfiable iff for each assignment to the universally quantified atoms there exists an assignment to the existentially quantified atoms that leads to an answer set. In our case, it means that for all possible starting states, there exists a sequence of actions such that either `fail` is derived or the constraints above are satisfied. But this can only be the case if for each starting state that satisfies the preconditions, a sequence of actions exists that reverts the chosen action.

3.5 Reversibility in ASP(Q)

For ASP(Q), we only slightly adapt an existing encoding for (non-uniform) action reversibility [16, Section 4.3]. While this encoding guesses an action to check for reversibility, in our setting, we want to supply this action externally via the `chosen/1` predicate. Hence, the outermost existential quantifier of the ASP(Q) encoding, which guesses the chosen action, is removed. Consequently, the resulting ASP(Q) program is of the form $\forall^{st} P_1 \exists^{st} P_2 : C$, where P_1 generates all states, P_2 guesses a sequence of actions, and the constraints C check that the sequence actually reverses the chosen action. This program is coherent iff the chosen action is universally reversible.

4 Reversibility in Bule

For the Bule encoding, we first ground the plasp output described in Section 3.1 using the ASP grounder gringo, and then apply some minor syntactic modifications to obtain valid Bule input.

The reversibility encoding is based on a planning encoding given in [26]. Time steps and action to be reversed are defined using

```
#ground time[0..n+1].    #ground horizon[n].
#ground chosen[action(a)].
```

The main reversibility problem is set up stating that for all initial states there exists a sequence of single actions:

```
contains[V, Val] :: #forall[1] holds(V, Val, 0).
chosen[A] :: occurs(A, 1).
action[A], time[T], T > 1 :: #exists[2] occurs(A, T).
time[T], T > 0 :: action[A]:occurs(A, T).
action[A],action[B],time[T],T>0,A<B :: ~occurs(A,T)|~occurs(B,T).
```

Then we identify invalid initial states (in which a variable has multiple or no values) and initial states in which the chosen action is not applicable, since these should not be considered for checking the reversal condition.

```
contains[V, V1], contains[V, V2], V1 < V2 ::
  holds(V, V1, 0) & holds(V, V2, 0) -> invvar(V).
contains[V, V1], contains[V, V2], V1 < V2 ::
  ~holds(V, V1, 0) & ~holds(V, V2, 0) -> invvar(V).
contains[V, V1], contains[V, V2], V1 != V2 ::
  invvar(V) & holds(V, V1, 0) -> holds(V, V2, 0).
contains[V, V1], contains[V, V2], V1 != V2 ::
  invvar(V) & ~holds(V, V1, 0) -> ~holds(V, V2, 0).
precondition[A, V, Val] ::
  occurs(A, 1) & ~holds(V, Val, 0) -> invprec(V).
variable[V], action[A] ::
  invprec(V) & occurs(A, 1) -> precondition[A, V, Val]:
  ~holds(V, Val, 0).
```

These conditions are fused into a single variable `invalid`.

```
invalid -> invalidvar | invalidprec.
invalidvar -> invalid.   invalidprec -> invalid.
variable[V] :: invvar(V) -> invalidvar.
invalidvar -> variable[V]:invvar(V).
variable[V] :: invprec(V) -> invalidprec.
invalidprec -> variable[V]:invprec(V).
```

Eventually, actions are applied. For unmet preconditions, contradictions, or if the plan is not a reversal, `noplan` will hold.

```
postcondition[A,_,V,_],time[T],T>0 :: occurs(A,T)->mod(V,T).
variable[V], time[T], T > 0 ::
  mod(V, T) -> postcondition[A, _, V, _]:occurs(A, T).
postcondition[A, _, V, Val], time[T], T > 0 ::
  occurs(A, T) -> holds(V, Val, T).
contains[V, Val], time[T], time[T-1] ::
  holds(V, Val, T-1) & ~mod(V, T) -> holds(V, Val, T).
contains[V, V1], contains[V, V2], time[T], T > 0, V1 < V2 ::
  holds(V, V1, T) & holds(V, V2, T) -> noplan.
contains[V, V1], contains[V, V2], time[T], T > 0, V1 < V2 ::
  ~holds(V, V1, T) & ~holds(V, V2, T) -> noplan.
precondition[A, V, Val], time[T], time[T-1] ::
  occurs(A, T) & ~holds(V, Val, T-1) -> noplan.
```

```
contains[V, Val], horizon[H] ::
  holds(V, Val, 0) & ~holds(V, Val, H+1) -> noplan.
contains[V, Val], horizon[H] ::
  ~holds(V, Val, 0) & holds(V, Val, H+1) -> noplan.
```

Finally, the initial state is irrelevant or the plan is a reversal.

```
invalid | ~noplan.
```

Correctness follows from the correctness of Jung et al.'s planning encoding [26], the quantifier pattern, and relevance, applicability, and reversal conditions. The Bule encoding is satisfiable iff the `chosen` action is universally reversible.

5 Experimental Evaluation

We have conducted preliminary experiments on universal reversibility, as it allows for comparison to existing logic programming solving methods. We benchmark the well-known ASP solver `clingo` [19], version 5.7.1, as well as an extension for ELPs built on top of `clingo`, called `eclingo` [4], version 1.2.0. For QASP, we use the `qasp2qbf` converter [17] paired with the `depqbf` QBF solver [30] in their most recent versions as of May 28, 2025. We have obtained an as-yet-unpublished prototype implementation for ASP(Q) from Francesco Ricca, called `qasp` version 0.1.2. It converts ASP(Q) to Quantified Boolean Formulas (QBFs) and then uses the QuAbS QBF solver [25] internally. Finally, we use the `bule` solver for the Bule QBF programming language, available from `https://github.com/vale1410/bule`. The complete benchmark archive is available[4].

For comparability, we use an established set of PDDL benchmark instances [15, 16]. The action `del-all` has a universal uniform reverse plan ⟨ `add-f0`, . . . , `add-fi` ⟩. We have generated instances from `i` = 1 to `i` = 90. We have analyzed runtime and memory consumption of two problems: (a) establishing reversibility of the `del-all` action for size `i` (by setting the constant `horizon` to `i`) and (b) proving that no `del-all` is not reversible (by setting the constant `horizon` to `i-1`). We compare and contrast our encodings from previous sections.

For preprocessing, we used `plasp` version 3.1.1 (`https://potassco.org/labs/plasp/`), as well as the grounder contained in `clingo` and simple regular expressions for conversion to Bule syntax. Our preliminary ASP(Q) solver uses the programs to Quantified Boolean Formulas (QBFs) and then use the well-known QuAbS QBF solver [25] internally. We preformed benchmarks on a computer with a 2.3 GHz AMD EPYC 7601 CPU with 8 virtualized cores and 500 GB RAM running CentOS 8. We have set a timeout of 20 minutes and a memory limit of 16GB. All solvers determined satisfiabiliy/validity.

The results for problem (a) are plotted in Figure 1. Missing dots represent timeouts or memouts. Contrary to expectations, the ASP(Q) solver performed best, both in terms of memory consumption and runtime, beating the other

[4] `https://drive.google.com/file/d/17E6HCyM5tadI8ZYC8AET-_hnGbhM09SF/view`

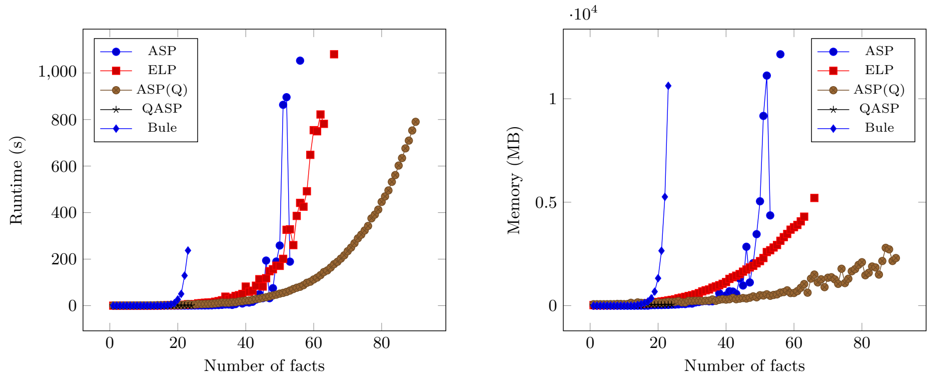

Fig. 1. Establishing that an action is not reversible.

solutions by a large margin. This is contrary to results obtained for the uniform version of the reversibility problem [16], where ASP was able to significantly outperform ASP(Q). Interestingly, also ELPs perform somewhat better than ASP, in particular in terms of memory consumption, but also in the number of instances solved. This is surprising, since an increase in expressive power of a language often comes with a performance trade-off, which does not seem to be the case here. On the other hand, the QASP language solver performed very well with competitive results up to instance size 24, but suddenly, from instance size 25 onwards, it ran into timeouts, with a huge jump in runtime. These timeouts occurred in the QBF solver DepQBF that ultimately solves the rewritten program. It would be interesting to investigate why such a large jump in terms of runtime happens here, in particular, since the generated QBF instance at size 25 is structurally similar to the one generated at size 24. For Bule, the direct QBF encoding does not perform well and was only able to solve less than 25 instances, running into memory issues for instances on the larger end of this spectrum. A preliminary investigation shows that this might be due to the grounder that is built into the `bule` solver using up a lot of space. The results for problem (b) are plotted in Figure 2. Interestingly, compared to (a), most encodings performed significantly better, but overall offer a similar picture. Bule and QASP behave as with problem (a). Interestingly, again, QASP times out above problem size 25.

In total, we obtain different results from previous work [15, 16]: While ASP(Q) and ELPs offer a far more readable and expressive language than ASP (both in terms of computational complexity and intuitiveness), they are able to outperform ASP in this setting. This may be down to the technique of saturation used in the ASP encoding, which is known to inhibit practical solving performance of ASP solvers significantly. It seems, however, that both ELPs and ASP(Q) are able to exploit the relatively simple structure of the underlying reversibility task well.

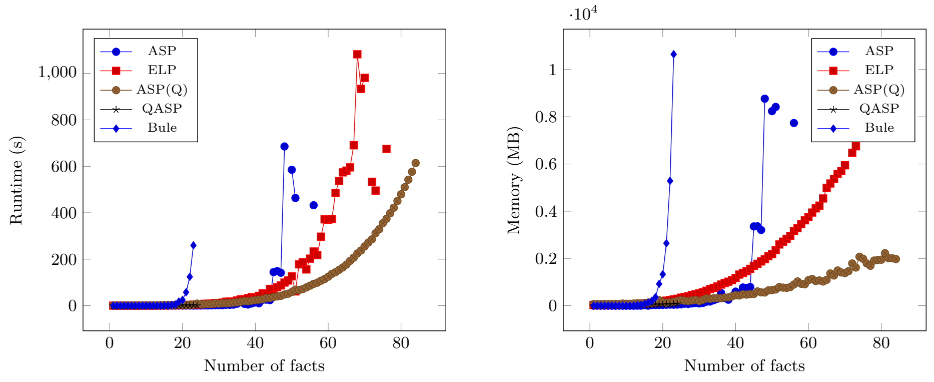

Fig. 2. Verifying that the action is reversible.

6 Conclusions

In this paper, we have provided encodings for the most general version of bounded action reversibility in STRIPS planning, a problem in Π_2^P, using various logic-based languages, and based on the PDDL-to-ASP translation tool *plasp* [12]. One encoding uses ASP, encoding the complement using a saturation-based encoding. A similar encoding for the complement problem is given via an ELP. Direct encodings are given with the more expressive ASP extensions ASP(Q) and QASP. We also provide a direct encoding for the QBF programming language Bule, which uses ideas from ASP grounding but retains the classical logic semantics rather than the stable model semantics (which underlies ASP). The last three encodings are arguably more readable than the ASP (and ELP) ones.

In order to test whether our encodings can be used in practice, we performed a set of benchmarks on artificially generated instances by checking whether a given action is universally non-uniformly reversible. We compared the five encodings using a representative system for each formalism, which in several cases is the only available system for the formalism. All of the tested systems do work, but scale to different degrees. The best scalability is achieved (to us surprisingly) by the ASP(Q) tool qasp (version 0.1.2). Runner-up in scalability is the ELP tool eclingo, followed by the well-known ASP system clingo. QASP and Bule performed comparatively unfavorable with respect to scalability. It has to be stated though that these are preliminary results, and no effort has yet been put in to investigate the precise reasons and whether they can be ameliorated (for example by using different QBF back-ends for some systems).

For future work, we intend to optimize our encodings further, extend them beyond STRIPS, and test them with different configurations of systems and solvers. It would also be interesting to see how the encodings perform when compared to a procedural implementation of the algorithms proposed for reversibility checking by [32]. We would also like to compare our approach to existing reversibility tools proposed in the literature [13, 10].

References

1. Amendola, G., Ricca, F., Truszczynski, M.: Beyond NP: quantifying over answer sets. Theory Pract. Log. Program. **19**(5-6), 705–721 (2019)
2. Brewka, G., Eiter, T., Truszczynski, M.: Answer set programming at a glance. Commun. ACM **54**(12), 92–103 (2011)
3. Bylander, T.: The computational complexity of propositional STRIPS planning. Artif. Intell. **69**(1-2), 165–204 (1994)
4. Cabalar, P., Fandinno, J., Garea, J., Romero, J., Schaub, T.: eclingo : A solver for epistemic logic programs. Theory Pract. Log. Program. **20**(6), 834–847 (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)
6. Chrpa, L., Faber, W., Morak, M.: Universal and uniform action reversibility. In: Proc. KR. pp. 651–654 (2021)
7. Chrpa, L., Lipovetzky, N., Sardiña, S.: Handling non-local dead-ends in agent planning programs. In: Proc. IJCAI. pp. 971–978 (2017)
8. 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`
9. 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)
10. Daum, J., Torralba, Á., Hoffmann, J., Haslum, P., Weber, I.: Practical undoability checking via contingent planning. In: Proc. ICAPS. pp. 106–114 (2016)
11. De Giacomo, G., Gerevini, A.E., Patrizi, F., Saetti, A., Sardiña, S.: Agent planning programs. Artif. Intell. **231**, 64–106 (2016)
12. Dimopoulos, Y., Gebser, M., Lühne, P., Romero, J., Schaub, T.: plasp 3: Towards effective ASP planning. TPLP **19**(3), 477–504 (2019)
13. Eiter, T., Erdem, E., Faber, W.: Undoing the effects of action sequences. J. Applied Logic **6**(3), 380–415 (2008)
14. Eiter, T., Gottlob, G.: On the computational cost of disjunctive logic programming: Propositional case. Ann. Math. Artif. Intell. **15**(3-4), 289–323 (1995)
15. Faber, W., Morak, M., Chrpa, L.: Determining action reversibility in STRIPS using answer set and epistemic logic programming. TPLP **21**(5), 646–662 (2021)
16. Faber, W., Morak, M., Chrpa, L.: Determining action reversibility in STRIPS using answer set programming with quantifiers. In: Proc. PADL. pp. 42–56 (2022)
17. Fandinno, J., Laferrière, F., Romero, J., Schaub, T., Son, T.C.: Planning with incomplete information in quantified answer set programming. Theory Pract. Log. Program. **21**(5), 663–679 (2021)
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)
19. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Multi-shot ASP solving with clingo. Theory Pract. Log. Program. **19**(1), 27–82 (2019)
20. 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`
21. Gelfond, M.: Logic programming and reasoning with incomplete information. Ann. Math. Artif. Intell. **12**(1-2), 89–116 (1994)

22. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Gener. Comput. **9**(3/4), 365–386 (1991)
23. Ghallab, M., Nau, D.S., Traverso, P.: Automated planning - theory and practice. Elsevier (2004)
24. 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`
25. Hecking-Harbusch, J., Tentrup, L.: Solving QBF by abstraction. In: Orlandini, A., Zimmermann, M. (eds.) Proceedings Ninth International Symposium on Games, Automata, Logics, and Formal Verification, GandALF 2018, Saarbrücken, Germany, 26-28th September 2018. EPTCS, vol. 277, pp. 88–102 (2018)
26. Jung, J.C., Mayer-Eichberger, V., Saffidine, A.: QBF programming with the modeling language bule. In: Proc. SAT. LIPIcs, vol. 236, pp. 31:1–31:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022)
27. Leclerc, A.P., Kahl, P.T.: A survey of advances in epistemic logic program solvers. CoRR **abs/1809.07141** (2018), `http://arxiv.org/abs/1809.07141`, also in Proc. of ASPOCP 2018
28. Lifschitz, V.: Answer Set Programming. Springer (2019)
29. 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`
30. Lonsing, F., Egly, U.: Depqbf 6.0: A search-based QBF solver beyond traditional QCDCL. In: Proc. CADE. LNCS, vol. 10395, pp. 371–384. Springer (2017)
31. Med, J., Chrpa, L., Morak, M., Faber, W.: Weak and strong reversibility of nondeterministic actions: Universality and uniformity. In: Proc. ICAPS. pp. 369–377. AAAI Press (2024)
32. Morak, M., Chrpa, L., Faber, W., Fišer, D.: On the reversibility of actions in planning. In: Proc. KR. pp. 652–661 (2020)
33. Schwartz, T., Boockmann, J.H., Martin, L.: Towards the evaluation of action reversibility in STRIPS using domain generators. In: Proc. FoIKS. LNCS, vol. 13388, pp. 226–236. Springer (2022)
34. Shen, Y., Eiter, T.: Evaluating epistemic negation in answer set programming. Artif. Intell. **237**, 115–135 (2016)