plasp 3: Towards Effective ASP Planning

Y. Dimopoulos[1], M. Gebser[2], P. Lühne[2], J. Romero[2], and T. Schaub[2]

[1] University of Cyprus
[2] University of Potsdam, Germany

Abstract. We describe the new version of the PDDL-to-ASP translator *plasp*. First, it widens the range of accepted PDDL features. Second, it contains novel planning encodings, some inspired by SAT planning and others exploiting ASP features such as well-foundedness. All of them are designed for handling multi-valued fluents in order to capture both PDDL as well as SAS planning formats. Third, enabled by multi-shot ASP solving, it offers advanced planning algorithms also borrowed from SAT planning. As a result, *plasp* provides us with an ASP-based framework for studying a variety of planning techniques in a uniform setting. Finally, we demonstrate in an empirical analysis that these techniques have a significant impact on the performance of ASP planning.

1 Introduction

Reasoning about actions and change constitutes a major challenge to any formalism for knowledge representation and reasoning. It therefore comes as no surprise that Automated Planning [4] was among the first substantial application of Answer Set Programming (ASP [12]). Meanwhile this has led to manifold action languages [9], various applications in dynamic domains [1], but only few adaptions of Automated Planning techniques [16]. Although this has provided us with diverse insights into how relevant concepts are expressed in ASP, almost no attention has been paid to making reasoning about actions and change effective. This is insofar surprising as a lot of work has been dedicated to planning with techniques from the area of Satisfiability Testing (SAT [2]), a field often serving as a role model for ASP.

We address this shortcoming with the third series of the *plasp* system. From its inception, the purpose of *plasp* was to provide an elaboration-tolerant platform to planning by using ASP. Already its original design [7] foresaw to compile planning problems formulated in the Planning Domain Definition Language (PDDL [13]) into ASP facts and to use ASP meta-encodings for modeling alternative planning techniques. These could then be solved with fixed horizons (and optimization) or in an incremental fashion. The redesigned *plasp* 3 system features optional preprocessing by the state-of-the-art planning system *Fast Downward* [10] (via the intermediate SAS format), a homogeneous factual representation capturing both PDDL and SAS input (with multi-valued fluents), and a normalization step to support advanced PDDL features. Moreover, *plasp* 3 provides a spectrum of ASP encodings ranging from adaptions of known SAT encodings [15] to novel encodings taking advantage of ASP-specific concepts. Finally, *plasp* 3 offers sophisticated planning algorithms, also stemming from SAT planning [15], by taking advantage of multi-shot ASP solving. The common structure of various incremental

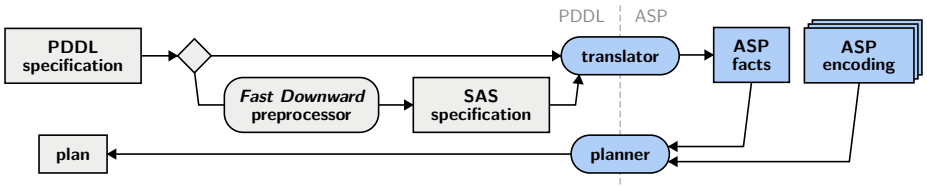

Fig. 1. Solving PDDL inputs with *plasp*'s workflow (highlighted in blue)

ASP encodings makes *plasp*'s planning framework also applicable to dynamic domains beyond PDDL. The usual workflow of *plasp* 3, though, is summarized in Figure 1.

2 ASP Encodings for Planning

We consider STRIPS-like (multi-valued) *planning tasks* according to [10], given by a 4-tuple $\langle \mathcal{F}, s_0, s_\star, \mathcal{O} \rangle$, in which

- $\mathcal{F}$ is a finite set of state variables, also called *fluents*, where each $x \in \mathcal{F}$ has an associated finite domain x^d of possible values for x,
- s_0 is a *state*, i.e., a (total) function such that $s_0(x) \in x^d$ for each $x \in \mathcal{F}$,
- $s_\star$ is a *partial state* (listing goal conditions), i.e., a function such that $s_\star(x) \in x^d$ for each $x \in \tilde{s}_\star$, where $\tilde{s}_\star$ denotes the set of all $x \in \mathcal{F}$ such that $s_\star(x)$ is defined, and
- $\mathcal{O}$ is a finite set of operators, also called *actions*, where a^c and a^e in $a = \langle a^c, a^e \rangle$ are partial states denoting the *precondition* and *postcondition* of a for each $a \in \mathcal{O}$.

Given a state s and an action $a \in \mathcal{O}$, the *successor state* $o(a, s)$ obtained by applying $a = \langle a^c, a^e \rangle$ in s is defined if $a^c(x) = s(x)$ for each $x \in \tilde{a}^c$, and undefined otherwise. Provided that $s' = o(a, s)$ is defined, $s'(x) = a^e(x)$ for each $x \in \tilde{a}^e$, and $s'(x) = s(x)$ for each $x \in \mathcal{F} \setminus \tilde{a}^e$. That is, if the successor state $o(a, s)$ is defined, it includes the postcondition of a and keeps any other fluents unchanged from s. We extend the notion of a successor state to sequences $\langle a_1, \ldots, a_n \rangle$ of actions by letting $o(\langle a_1, \ldots, a_n \rangle, s) = o(a_n, o(\ldots, o(a_1, s) \ldots))$, provided that $o(a_i, o(\ldots, o(a_1, s) \ldots))$ is defined for all $1 \leq i \leq n$. Given this, a *sequential plan* is a sequence $\langle a_1, \ldots, a_n \rangle$ of actions such that $s' = o(\langle a_1, \ldots, a_n \rangle, s_0)$ is defined and $s'(x) = s_\star(x)$ for each $x \in \tilde{s}_\star$.

Several *parallel* representations of sequential plans have been investigated in the literature [4, 15, 17]. We call a set $\{a_1, \ldots, a_k\} \subseteq \mathcal{O}$ of actions *confluent* if $a_i^e(x) = a_j^e(x)$ for all $1 \leq i < j \leq k$ and each $x \in \tilde{a}_i^e \cap \tilde{a}_j^e$. Given a state s and a confluent set $A = \{a_1, \ldots, a_k\}$ of actions, A is

- $\forall$*-step serializable* in s if $o(\langle a'_1, \ldots, a'_k \rangle, s)$ is defined for any sequence $\langle a'_1, \ldots, a'_k \rangle$ such that $\{a'_1, \ldots, a'_k\} = A$;
- $\exists$*-step serializable* in s if $a^c(x) = s(x)$, for each $a \in A$ and $x \in \tilde{a}^c$, and $o(\langle a'_1, \ldots, a'_k \rangle, s)$ is defined for some sequence $\langle a'_1, \ldots, a'_k \rangle$ such that $\{a'_1, \ldots, a'_k\} = A$;
- *relaxed* $\exists$*-step serializable* in s if $o(\langle a'_1, \ldots, a'_k \rangle, s)$ is defined for some sequence $\langle a'_1, \ldots, a'_k \rangle$ such that $\{a'_1, \ldots, a'_k\} = A$.

Note that any $\forall$-step serializable set A of actions is likewise $\exists$-step serializable, and similarly any $\exists$-step serializable A is relaxed $\exists$-step serializable. In particular, the condition that any sequence built from a $\forall$-step serializable A leads to a (defined) successor state implies that the precondition of each action in A must already be established, which is also required for $\exists$-step serializable sets, but not for relaxed $\exists$-step serializable sets. We extend the three serialization concepts to plans by calling a sequence $\langle A_1, \ldots, A_m \rangle$ a $\forall$-*step*, $\exists$-*step*, or *relaxed* $\exists$-*step plan* if $s_m(x) = s_\star(x)$, for each $x \in \tilde{s}_\star$, and each set A_i of actions is $\forall$-step, $\exists$-step, or relaxed $\exists$-step serializable, respectively, in s_{i-1} for $1 \leq i \leq m$, where $s_i(x) = a^e(x)$ for each $a \in A_i$ and $x \in \tilde{a}^e$, and $s_i(x) = s_{i-1}(x)$ for each $x \in \mathcal{F} \setminus \bigcup_{a \in A_i} \tilde{a}^e$. That is, parallel representations partition some sequential plan such that each part A_i is $\forall$-step, $\exists$-step, or relaxed $\exists$-step serializable in the state obtained by applying the actions preceding A_i.

Example 1. Consider a planning task $\langle \mathcal{F}, s_0, s_\star, \mathcal{O} \rangle$ with $\mathcal{F} = \{x_1, x_2, x_3, x_4, x_5\}$ such that $x_1^d = x_2^d = x_3^d = x_4^d = x_5^d = \{0, 1\}$, $s_0 = \{x_1 = 0, x_2 = 0, x_3 = 0, x_4 = 0, x_5 = 0\}$, $s_\star = \{x_4 = 1, x_5 = 1\}$, and $\mathcal{O} = \{a_1, a_2, a_3, a_4\}$, where $a_1 = \langle \{x_1 = 0\}, \{x_1 = 1, x_2 = 1\} \rangle$, $a_2 = \langle \{x_3 = 0\}, \{x_1 = 1, x_3 = 1\} \rangle$, $a_3 = \langle \{x_2 = 1, x_3 = 1\}, \{x_4 = 1\} \rangle$, and $a_4 = \langle \{x_2 = 1, x_3 = 1\}, \{x_5 = 1\} \rangle$. One can check that $\langle a_1, a_2, a_3, a_4 \rangle$ and $\langle a_1, a_2, a_4, a_3 \rangle$ are the two sequential plans consisting of four actions. The $\forall$-step plan with fewest sets of actions is given by $\langle \{a_1\}, \{a_2\}, \{a_3, a_4\} \rangle$. Similarly, $\langle \{a_1, a_2\}, \{a_3, a_4\} \rangle$ is the $\exists$-step plan with fewest sets of actions. Finally, the relaxed $\exists$-step plan $\langle \{a_1, a_2, a_3, a_4\} \rangle$ consists of one set of actions only. ■

In ASP, we represent a planning task like the one in Example 1 by facts as follows:

```
fluent(x1).  fluent(x2).  fluent(x3).  fluent(x4).  fluent(x5).
value(x1,0). value(x2,0). value(x3,0). value(x4,0). value(x5,0).
value(x1,1). value(x2,1). value(x3,1). value(x4,1). value(x5,1).

init(x1,0). init(x2,0). init(x3,0). init(x4,0). init(x5,0).
                                    goal(x4,1). goal(x5,1).

action(a1).    action(a2).    action(a3).    action(a4).
prec(a1,x1,0). prec(a2,x3,0). prec(a3,x2,1). prec(a4,x2,1).
post(a1,x1,1). post(a2,x1,1). prec(a3,x3,1). prec(a4,x3,1).
post(a1,x2,1). post(a2,x3,1). post(a3,x4,1). post(a4,x5,1).
```

The facts can then be combined with encodings such that stable models correspond to sequential, $\forall$-step, $\exists$-step, or relaxed $\exists$-step plans. The rules as well as integrity constraints in Listing 1 form the common core of respective incremental encodings [6] and are grouped into three parts: a subprogram `base`, including the rule in Line 1, which is not preceded by any **`#program`** directive; a parameterized subprogram `check(t)`, containing the integrity constraint in Line 5, in which the parameter `t` serves as placeholder for successive integers starting from `0`; and a parameterized subprogram `step(t)`, comprising the rules and integrity constraints below the **`#program`** directive in Line 7, whose parameter `t` stands for successive integers starting from `1`. By first instantiating the `base` subprogram along with `check(t)`, where `t` is replaced by `0`, and then proceeding with integers from `1` for `t` in `check(t)` and `step(t)`, an incremental

Listing 1. Common part of sequential and parallel encodings for STRIPS-like planning

```
1  holds(X,V,0) :- init(X,V).

3  #program check(t).

5  :- query(t), goal(X,V), not holds(X,V,t).

7  #program step(t).

9  {holds(X,V,t) : value(X,V)} = 1 :- fluent(X).

11 {occurs(A,t)} :- action(A).

13 :- occurs(A,t), post(A,X,V), not holds(X,V,t).

15 change(X,t) :- holds(X,V,t-1), not holds(X,V,t).
16 effect(X,t) :- occurs(A,t), post(A,X,V).
17 :- change(X,t), not effect(X,t).
```

Listing 2. Extension of Listing 1 for encoding sequential plans

```
19 :- occurs(A,t), prec(A,X,V), not holds(X,V,t-1).

21 :- #count{A : occurs(A,t)} > 1.
```

encoding can be gradually unrolled. We take advantage of this to capture plans of increasing length, expressed by the latest integer used to replace `t` with.

In more detail, the rule in Line 1 of Listing 1 maps facts specifying s_0 to atoms over the predicate `holds`/3, in which the third argument `0` refers to the given state. Starting from `0` for the parameter `t`, the integrity constraint in Line 5 then tests whether the conditions of $s_\star$ are established, where the dedicated atom `query(t)` is set to true only for the latest integer taken for `t`. This allows for increasing the plan length by successively instantiating the subprograms `check(t)` and `step(t)` with further integers. The latter subprogram includes the choice rule in Line 9 to generate a successor state such that each fluent $x \in \mathcal{F}$ is mapped to some value in its domain x^d. The other choice rule in Line 11 permits to unconditionally pick actions to apply, expressed by atoms over `occurs`/2, in order to obtain a corresponding successor state. Given that both sequential and parallel plans are such that the postcondition of an applied action holds in the successor state, the integrity constraint in Line 13 asserts respective postcondition(s). On the other hand, fluents unaffected by applied actions must remain unchanged, which is reflected by the rules in Lines 15 and 16 along with the integrity constraint in Line 17, restricting changed fluents to postconditions of applied actions.

The common encoding part described so far takes care of matching successor states to postconditions of applied actions, while requirements regarding preconditions are subject to the kind of plan under consideration and expressed by dedicated additions to the `step(t)` subprogram. To begin with, the two integrity constraints added in Listing 2 address sequential plans by, in Line 19, asserting the precondition of an applied

Listing 3. Extension of Listing 1 for encoding ∀-step plans

```
19 :- occurs(A,t), prec(A,X,V), not holds(X,V,t-1).

21 :- occurs(A,t), prec(A,X,V), not post(A,X,_), not holds(X,V,t).

23 single(X,t) :- occurs(A,t), prec(A,X,V1), post(A,X,V2), V1 != V2.
24 :- single(X,t), #count{A : occurs(A,t), post(A,X,V)} > 1.
```

action to hold at the state referred to by `t-1` and, in Line 21, denying multiple actions to be applied in parallel. Note that, if the plan length or the latest integer taken for `t`, respectively, exceeds the minimum number of actions required to establish the conditions of $s_\star$, the encoding of sequential plans given by Listings 1 and 2 permits idle states in which no action is applied. While idle states cannot emerge when using the basic *iclingo* control loop [6] of *clingo* to compute shortest plans, they are essential for the planner presented in Section 3 in order to increase the plan length in more flexible ways.

Turning to parallel representations, Listing 3 shows additions dedicated to ∀-step plans, where the integrity constraint in Line 19 is the same as in Listing 2 before. This guarantees the preconditions of applied actions to hold, while their confluence is already taken care of by means of the integrity constraint in Line 13. It thus remains to make sure that applied actions do not interfere in a way that would disable any serialization, which essentially means that the precondition of an applied action a must not be invalidated by another action applied in parallel. For a fluent $x \in \tilde{a}^c$ that is not changed by a itself, i.e., $x \notin \tilde{a}^e$ or $a^e(x) = a^c(x)$, the integrity constraint in Line 21, which applies in case of $x \notin \tilde{a}^e$, suppresses a parallel application of actions a' such that $x \in \tilde{a}'^e$ and $a'^e(x) \neq a^c(x)$. (If $a^e(x) = a^c(x)$, the integrity constraint in Line 13 already requires x to remain unchanged.) On the other hand, the situation becomes slightly more involved when $x \in \tilde{a}^e$ and $a^e(x) \neq a^c(x)$, i.e., the application of a invalidates its own precondition. In this case, no other action a' such that $x \in \tilde{a}'^e$ can be applied in parallel, either because $a'^e(x) \neq a^e(x)$ undermines confluence or since $a'^e(x) = a^e(x)$ disrespects the precondition of a. To account for such situations and address all actions invalidating their precondition regarding x at once, the rule in Line 23 derives an atom over `single`/2 to indicate that at most (and effectively exactly) one action affecting x can be applied, as asserted by the integrity constraint in Line 24. As a consequence, no action applied in parallel can invalidate the precondition of another action, so that any serialization leads to the same successor state as obtained in the parallel case.

Example 2. The two sequential plans from Example 1 correspond to two stable models, obtained with the encoding of sequential plans given by Listings 1 and 2, both including the atoms `occurs(`a_1`,1)` and `occurs(`a_2`,2)`. In addition, one stable model contains `occurs(`a_3`,3)` along with `occurs(`a_4`,4)`, and the other `occurs(`a_4`,3)` as well as `occurs(`a_3`,4)`, thus exchanging the order of applying a_3 and a_4. Given that a_3 and a_4 are confluent, the independence of their application order is expressed by a single stable model, obtained with the encoding part for ∀-step plans in Listing 3 instead of the one in Listing 2, comprising `occurs(`a_3`,3)` as well as `occurs(`a_4`,3)` in addition to `occurs(`a_1`,1)` and `occurs(`a_2`,2)`. Note that, even though the set

Listing 4. Extension of Listing 1 for encoding ∃-step plans

```
19 :- occurs(A,t), prec(A,X,V), not holds(X,V,t-1).

21 apply(A1,t) :- action(A1),
22    ready(A2,t) : post(A1,X,V1), prec(A2,X,V2), A1 != A2, V1 != V2.

24 ready(A,t) :- action(A), not occurs(A,t).
25 ready(A,t) :- apply(A,t).
26 :- action(A), not ready(A,t).
```

Listing 5. Replacement of Lines 21–26 in Listing 4 by **#edge** statement

```
21 #edge((A1,t),(A2,t)) : occurs(A1,t),
22                        post(A1,X,V1), prec(A2,X,V2), A1 != A2, V1 != V2.
```

$\{a_1, a_2\}$ is confluent, it is not ∀-step serializable (in s_0), and a parallel application is suppressed in view of the atom `single(`x_1`,1)`, derived since a_1 invalidates its precondition regarding x_1. Moreover, the requirement that the precondition of an applied action must be established in the state before permits only $\langle\{a_1\}, \{a_2\}, \{a_3, a_4\}\rangle$ as ∀-step plan or its corresponding stable model, respectively, with three sets of actions. ■

Additions to Listing 1 addressing ∃-step plans are given in Listing 4. As before, the integrity constraint in Line 19 is included to assert the precondition of an applied action to hold at the state referred to by `t-1`. Unlike with ∀-step plans, however, an applied action may invalidate the precondition of another action, in which case the other action must come first in a serialization, and the aim is to make sure that there is some compatible serialization. To this end, the rule in Lines 21–22 expresses that an action can be safely applied, as indicated by a respective instance of the head atom `apply(A1,t)`, once *all* other actions whose preconditions it invalidates are captured by corresponding instances of `ready(A2,t)`. The latter provide actions that are not applied or whose application is safe, i.e., no yet pending action's precondition gets invalidated, and are derived by means of the rules in Lines 24 and 25. In fact, the least fixpoint obtained via the rules in Lines 21–25 covers all actions exactly if the applied actions do not circularly invalidate their preconditions, and the integrity constraint in Line 26 prohibits any such circularity, which in turn means that there is a compatible serialization. Excluding circular interference also lends itself to an alternative implementation by means of the **#edge** directive [5] of *clingo*, in which case built-in acyclicity checking [3] is used. A respective replacement of Lines 21–26 is shown in Listing 5, where the **#edge** directive in Lines 21–22 asserts edges from an applied action to all other actions whose preconditions it invalidates, and acyclicity checking makes sure that the graph induced by applied actions remains acyclic.

The encoding part for relaxed ∃-step plans in Listing 6 deviates from those given so far by not necessitating the precondition of an applied action to hold in the state before. Rather, the preconditions of actions applied in parallel may be established successively,

Listing 6. Extension of Listing 1 for encoding relaxed ∃-step plans

```
19 reach(X,V,t) :- holds(X,V,t-1).
20 reach(X,V,t) :- occurs(A,t), apply(A,t), post(A,X,V).

22 apply(A1,t) :- action(A1), reach(X,V,t) : prec(A1,X,V);
23    ready(A2,t) : post(A1,X,V1), prec(A2,X,V2), A1 != A2, V1 != V2.

25 ready(A,t) :- action(A), not occurs(A,t).
26 ready(A,t) :- apply(A,t).
27 :- action(A), not ready(A,t).
```

where confluence along with the condition that an action is applicable only after other actions whose preconditions it invalidates have been processed guarantee the existence of a compatible serialization. In fact, the rules in Lines 22–26 are almost identical to their counterparts in Listing 4, and the difference amounts to the additional prerequisite '`reach(X,V,t) : prec(A1,X,V)`' in Line 22. Instances of `reach(X,V,t)` are derived by means of the rules in Lines 19 and 20 to indicate fluent values from the state referred to by `t-1` along with postconditions of actions whose application has been determined to be safe. The prerequisites of the rule in Lines 22–23 thus express that an action can be safely applied once its precondition is established, possibly by means of other actions preceding it in a compatible serialization, *and* if it does not invalidate any pending action's precondition. Similar to its counterpart in Listing 4, the integrity constraint in Line 27 then makes sure that actions are not applied unless their application is safe in the sense of a relaxed ∃-step serializable set.

Example 3. The ∀-step plan $\langle\{a_1\},\{a_2\},\{a_3,a_4\}\rangle$ from Example 1 can be condensed into $\langle\{a_1,a_2\},\{a_3,a_4\}\rangle$ when switching to ∃-step serializable sets. Corresponding stable models obtained with the encodings given by Listing 1 along with Listing 4 or 5 include `occurs(`a_1`,1)`, `occurs(`a_2`,1)`, `occurs(`a_3`,2)`, and `occurs(`a_4`,2)`. Regarding the **`#edge`** directive in Listing 5, these atoms induce the graph $(\{$`(`a_1`,1)`, `(`a_2`,1)`$\},\{\langle$`(`a_2`,1)`, `(`a_1`,1)`$\rangle\})$, which is clearly acyclic. Its single edge tells us that a_1 must precede a_2 in a compatible serialization, while the absence of a cycle means that the application of a_1 does not invalidate the precondition of a_2. In terms of the encoding part in Listing 4, `apply(`a_1`,1)` and `ready(`a_1`,1)` are derived first, which in turn allows for deriving `apply(`a_2`,1)` and `ready(`a_2`,1)`. The requirement that the precondition of an applied action must be established in the state before, which is shared by Listings 4 and 5, however, necessitates at least two sets of actions for an ∃-step plan or a corresponding stable model, respectively. Unlike that, the encoding of relaxed ∃-step plans given by Listings 1 and 6 yields a stable model containing `occurs(`a_1`,1)`, `occurs(`a_2`,1)`, `occurs(`a_3`,1)`, and `occurs(`a_4`,1)`, corresponding to the relaxed ∃-step plan $\langle\{a_1,a_2,a_3,a_4\}\rangle$. The existence of a compatible serialization is witnessed by first deriving, amongst other atoms, `reach(`x_1`,0,1)` and `reach(`x_3`,0,1)` in view of s_0. These atoms express that the preconditions of a_1 and a_2 are readily established, so that `apply(`a_1`,1)` along with `reach(`x_2`,1,1)` and `ready(`a_1`,1)` are derived next. The latter atom indicates that a_1 can be safely

applied before a_2, which then leads to `apply(`a_2`,1)` along with `reach(`x_3`,1,1)`. Together `reach(`x_2`,1,1)` and `reach(`x_3`,1,1)` reflect that the precondition of a_3 as well as a_4 can be established by means of a_1 and a_2 applied in parallel, so that `apply(`a_3`,1)` and `apply(`a_4`,1)` are derived in turn. ■

In order to formalize the soundness and completeness of the presented encodings, let B stand for the rule in Line 1 of Listing 1, $Q(i)$ for the integrity constraint in Line 5 with the parameter `t` replaced by some integer i, and $S(i)$ for the rules and integrity constraints below the **`#program`** directive in Line 7 with i taken for `t`. Moreover, we refer to specific encoding parts extending $S(i)$, where the parameter `t` is likewise replaced by i, by $S^s(i)$ for Listing 2, $S^\forall(i)$ for Listing 3, $S^\exists(i)$ for Listing 4, $S^E(i)$ for Line 19 of Listing 4 along with Listing 5, and $S^R(i)$ for Listing 6. Given that $S^E(i)$ includes an **`#edge`** directive subject to acyclicity checking, we understand stable models in the sense of [3], i.e., the graph induced by a (regular) stable model, which is empty in case of no **`#edge`** directives, must be acyclic.

Theorem 1. *Let I be the set of facts representing planning task $\langle \mathcal{F}, s_0, s_\star, \mathcal{O} \rangle$, $\langle a_1, \ldots, a_n \rangle$ be a sequence of actions, and $\langle A_1, \ldots, A_m \rangle$ be a sequence of sets of actions. Then,*

- *$\langle a_1, \ldots, a_n \rangle$ is a sequential plan iff $I \cup B \cup Q(0) \cup \bigcup_{i=1}^{n}(Q(i) \cup S(i) \cup S^s(i)) \cup \{$`query(`$n$`).`$\}$ has a stable model M such that $\{\langle a, i \rangle \mid$ `occurs(`a`,`i`)` $\in M\} = \{\langle a_i, i \rangle \mid 1 \leq i \leq n\}$;*
- *$\langle A_1, \ldots, A_m \rangle$ is a $\forall$-step (resp., $\exists$-step or relaxed $\exists$-step) plan iff $I \cup B \cup Q(0) \cup \bigcup_{i=1}^{m}(Q(i) \cup S(i) \cup S^p(i)) \cup \{$`query(`$m$`).`$\}$, where $S^p(i) = S^\forall(i)$ (resp., $S^p(i) \in \{S^\exists(i), S^E(i)\}$ or $S^p(i) = S^R(i)$) for $1 \leq i \leq m$, has a stable model M such that $\{\langle a, i \rangle \mid$ `occurs(`a`,`i`)` $\in M\} = \{\langle a, i \rangle \mid 1 \leq i \leq m, a \in A_i\}$.*

Let us note that, with each of the considered encodings, any plan corresponds to a unique stable model, as the latter is fully determined by atoms over `occurs`/2, i.e., corresponding (successor) states as well as auxiliary predicates functionally depend on the applied actions. Regarding the encoding part for relaxed $\exists$-step plans in Listing 6, we mention that acyclicity checking cannot (in an obvious way) be used instead of rules dealing with the safe application of actions. To see this, consider $\langle \mathcal{F}, s_0, s_\star, \mathcal{O} \rangle$ with $\mathcal{F} = \{x_1, x_2, x_3\}$ such that $x_1^d = x_2^d = x_3^d = \{0, 1\}$, $s_0 = \{x_1 = 0, x_2 = 0, x_3 = 0\}$, $s_\star = \{x_3 = 1\}$, and $\mathcal{O} = \{a_1, a_2\}$, where $a_1 = \langle \emptyset, \{x_1 = 1, x_2 = 1\} \rangle$ and $a_2 = \langle \{x_1 = 1, x_2 = 0\}, \{x_3 = 1\} \rangle$. There is no sequential plan for this task since only a_1 is applicable in s_0, but its application invalidates the precondition of a_2. Concerning the (confluent) set $\{a_1, a_2\}$, the graph $(\{$`(`a_1`,1)`$, $`(`$a_2$`,1)`$\}, \{\langle$`(`$a_1$`,1)`$, $`(`$a_2$`,1)`$\rangle\})$ is acyclic and actually includes the information that a_2 should precede a_1 in any compatible serialization. However, if the prerequisite in Line 23 of Listing 6 were dropped to "simplify" the encompassing rule, the application of a_1 would be regarded as safe, and then the precondition of a_2 would seem established as well. That is, it would be unsound to consider the establishment and invalidation of preconditions in separation, no matter the respective implementation techniques.

As regards encoding techniques, common ASP-based approaches, e.g., [12], define successor states, i.e., the predicate `holds`/3, in terms of actions given by atoms over `occurs`/2. Listing 1, however, includes a respective choice rule, which puts it inline with

SAT planning, where our intention is to avoid asymmetries between fluents and actions, as either of them would in principle be sufficient to indicate plans [11]. Concerning (relaxed) $\exists$-step plans, the encoding parts in Listings 4 and 6 make use of the built-in well-foundedness requirement in ASP and do, unlike [15], not unfold the order of actions applied in parallel. In contrast to the SAT approach to relaxed $\exists$-step plans in [17], we do not rely on a fixed (static) order of actions, and to our knowledge, no encoding similar to the one in Listing 6 has been proposed so far.

3 A Multi-Shot ASP Planner

Planning encodings must be used with a strategy for fixing the plan length. For example, the first approaches to planning in SAT and ASP follow a sequential algorithm starting from 0 and successively incrementing the length by 1 until a plan is found.

For parallel planning in SAT, more flexible strategies were proposed in [15], based on the following ideas. First, minimal parallel plans do not coincide with shortest sequential plans. Hence, it is unclear whether parallel plans should be minimal. Second, solving times for different plan lengths follow a certain pattern, which can be exploited. To illustrate this, consider the solving times of a typical instance in Figure 2. For lengths 0 to 4, in gray, the instance is unsatisfiable, and time grows exponentially. Then, the first satisfiable instances, in light green, are still hard, but they become easier for greater plan lengths. However, for even greater plan lengths, the solving time increases again because the size becomes larger. Accordingly, [15] suggests not to minimize parallel plan length but rather avoid costly unsatisfiable parts by moving early to easier satisfiable lengths.

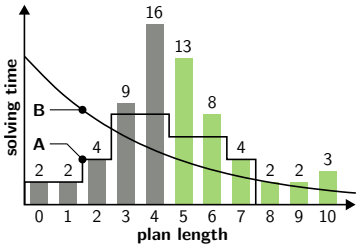

Fig. 2. Exemplary solving times

The sequential algorithm (S) solves the instance in Figure 2 in 46 time units, viz. $2 + 2 + 4 + 9 + 16 + 13$, by trying plan lengths 0 to 4 until it finds a plan at 5. The idea of algorithm A [15] is to simultaneously consider n plan lengths. In our example, fixing n to 5, A starts with lengths 0 to 4. After 2 time units, lengths 0 and 1 are finished, and 5 and 6 are added. Another 2 units later, length 2 is finished, and 7 is started. Finally, at time 8, length 7 yields a plan. The times spent by A for each length are indicated in Figure 2, and summing them up amounts to 40 time units in total. Algorithm B [15] distributes time non-uniformly over plan lengths: if length n is run for t time units, then lengths $n + i$ are run for $t * \gamma^i$ units, where $i \geq 1$ and γ lies between 0 and 1. In our example, we set γ to 0.8. When length 3 has been run for 6 units, previous lengths are already finished, and the times for the following lengths are given by curve B in Figure 2. At this point, length 8 is assigned 2 units ($6 * 0.8^5$) and yields a plan, leading to a total time of 38 units: 8 units for lengths 0 to 2, and 30 for the rest. (The 30 units correspond to the area under the curve from length 3 on.) Note that both A and B find a plan before finishing the hardest instances and, in practice, often save significant time over S.

We adopted algorithms A (S if $n = 1$) and B, and implemented them as planning strategies of *plasp* via multi-shot ASP solving. In general, they can be applied to any

incremental encoding complying with the threefold structure of `base`, `step(t)`, and `check(t)` subprograms. Assuming that the subprograms adhere to *clingo*'s modularity condition [6], they are assembled to ASP programs of the form

$$P(n) \;=\; \texttt{base} \cup \bigcup_{i=0}^{n} \texttt{check}(i) \cup \bigcup_{i=1}^{n} \texttt{step}(i)$$

where n gives the length of the unrolled encoding. The planner then looks for an integer n such that $P(n) \cup \{\texttt{query}(n)\texttt{.}\}$ is satisfiable, and algorithms S, A, and B provide different strategies to approach such an integer.

The planner is implemented using *clingo*'s multi-shot solving capacities, where one *clingo* object grounds and solves incrementally. This approach avoids extra grounding efforts and allows for taking advantage of previously learned constraints. The planner simulates the parallel processing of different plan lengths by interleaving sequential subtasks. To this end, the *clingo* object is used to successively unroll an incremental encoding up to integer(s) n. In order to solve a subtask for some $m < n$, the unrolled part $P(n)$ is kept intact, while $\texttt{query}(m)$ is set to true instead of $\texttt{query}(n)$. That is, the search component of *clingo* has to establish conditions in $\texttt{check}(m)$, even though the encoding is unrolled up to $n \geq m$. For this approach to work, we require that $P(m) \cup \{\texttt{query}(m)\texttt{.}\}$ is satisfiable iff $P(n) \cup \{\texttt{query}(m)\texttt{.}\}$ is satisfiable for $0 \leq m \leq n$. An easy way to guarantee this property is to tolerate idle states in-between m and n, as is the case with the encodings given in Section 2.

4 System and Experiments

Like its predecessor versions, the third series of *plasp*[3] provides a translator from PDDL specifications to ASP facts. Going beyond the STRIPS-like fragment, it incorporates a normalization step to support advanced PDDL features such as nested expressions in preconditions, conditional effects, axiom rules, as well as existential and universal quantifiers. Moreover, *plasp* allows for optional preprocessing by *Fast Downward*, leading to an intermediate representation in the SAS planning format. This format encompasses multi-valued fluents, mutex groups, conditional effects, and axiom rules, which permit a compact (propositional) specification of planning tasks and are (partially) inferred by *Fast Downward* from PDDL inputs. Supplied with PDDL or SAS inputs, *plasp* produces a homogeneous factual representation, so that ASP encodings remain independent of the specific input format.[4]

To empirically contrast the different encodings and planning algorithms presented in Sections 2 and 3, we ran *plasp* on PDDL specifications from the International Planning Competition. For comparison, we also include two variants of the state-of-the-art SAT planning system *Madagascar* [14], where M stands for the standard version and M_p for the use of a specific planning heuristic. All experiments were performed on a Linux

[3] Available at: https://github.com/potassco/plasp

[4] The encodings given in Section 2 focus on STRIPS-like planning tasks with multi-valued fluents as well as mutex groups, where the latter have been omitted for brevity. In contrast to the ease of incorporating mutex groups, extending parallel encodings to conditional effects or axiom rules is not straightforward [15], while sequential encodings for them are shipped with *plasp*.

	solved	∅ time	gripper	logistics	blocks	elevator	depots	driverlog
M_p	76/76	0.97	0.03	0.02	0.54	0.02	0.07	4.73
M	76/76	1.18	0.08	0.02	0.15	0.02	0.15	6.44
$\mathsf{B}_p^\exists$	72/76	98.06	416.21	46.15	79.34	0.77	129.58	15.66
$\mathsf{B}^\exists$	60/76	238.64	39.90	46.16	637.86	10.90	305.45	146.29
$\mathsf{A}^\exists$	59/76	257.40	51.85	46.22	654.63	10.62	392.52	149.02
B^E	56/76	255.12	34.30	46.10	657.20	2.37	413.98	139.97
B^R	54/76	348.67	134.96	44.49	741.69	66.72	547.06	274.72
B^s	49/76	412.90	622.39	62.83	517.26	89.37	669.86	460.33
$\mathsf{B}^\forall$	48/76	402.64	516.33	46.12	537.78	434.63	283.20	251.81
$\mathsf{S}^\exists$	47/76	389.78	720.20	46.18	590.74	317.76	261.66	140.42
$\mathsf{B}_p^\exists$	75/76	15.48	1.73	0.47	3.10	0.89	10.44	74.31

Table 1. Solved instances and average runtimes without preprocessing by *Fast Downward*

machine equipped with Intel Core i7-2600 processor at 3.8 GHz and 16 GB RAM, limiting time and memory per run to 900 seconds and 8 GB, while charging 900 seconds per aborted run in the tables below.

Regarding *plasp*, we indicate the encoding of a particular kind of plan by a superscript to the planning algorithm (denoted by its letter), where ^{s} stands for sequential, $^\forall$ for $\forall$-step, $^\exists$ for $\exists$-step, ^{E} for $\exists$-step by means of acyclicity checking, and ^{R} for relaxed $\exists$-step plans; e.g., $\mathsf{B}^\exists$ refers to algorithm B applied to the encoding of $\exists$-step plans given by Listings 1 and 4. Moreover, each combination of algorithm and encoding can optionally be augmented with a planning heuristic [8], which has been inspired by M_p and is denoted by an additional subscript $_p$, like in $\mathsf{B}_p^\exists$. (The parameters of A and B are set to $n = 16$ or $\gamma = 0.9$, respectively, as suggested in [15].)

Tables 1 and 2 show total numbers of solved instances and average runtimes, in total and for individual domains of PDDL specifications, for the two *Madagascar* variants and different *plasp* settings. In case of Table 1, all systems take PDDL inputs directly, while preprocessing by *Fast Downward* is used for *plasp* in Table 2. Note that the tables refer to different subsets of instances, as we omit instances solved by some *plasp* setting in less than 5 seconds or unsolved by all of them within the given resource limits. To give an account of the impact of preprocessing, we list the best-performing *plasp* setting with or without preprocessing in the last row of the respective other table. As the $\mathsf{B}^\exists$ setting of *plasp*, relying on algorithm B along with the (pure) ASP encoding of $\exists$-step plans, turns out to perform generally robust as well as comparable to the alternative implementation provided by B^E, we choose it as the baseline for varying the planning algorithm, encoding, or heuristic.

In Table 1, where *plasp* takes PDDL inputs directly, we first observe that the *Madagascar* variants solve the respective instances rather easily. Unlike that, we checked that the size of ground instantiations often becomes large and constitutes a bottleneck with all *plasp* settings, which means that the original PDDL specifications are not directly suitable for ASP-based planning. The sequential algorithm or encoding, respectively, of $\mathsf{S}^\exists$ and B^s is responsible for increased search efforts or plan lengths, so that these settings solve considerably fewer instances than the baseline provided by $\mathsf{B}^\exists$, which is

	solved	∅ time	grid	gripper	logistics	mystery	blocks	elevator	freecell	depots	driverlog
M_p	136/136	2.76	0.78	0.04	13.09	4.11	1.92	0.02	0.59	0.22	0.21
M	135/136	12.89	1.40	0.34	13.18	71.07	1.12	0.03	18.91	7.38	3.16
$\mathsf{B}_p^\exists$	121/136	131.03	72.14	5.89	76.82	282.99	27.52	2.22	656.76	40.22	27.91
B^E	116/136	159.52	460.51	19.29	32.12	345.20	7.47	1.92	655.19	135.48	206.84
$\mathsf{B}^\exists$	114/136	168.44	63.01	19.42	32.80	351.92	21.48	2.00	656.47	223.28	208.59
$\mathsf{A}^\exists$	114/136	174.68	297.15	15.62	32.93	362.73	21.50	2.01	656.49	234.51	218.26
$\mathsf{B}^\forall$	107/136	231.49	459.96	277.92	48.00	387.43	4.92	4.95	652.87	344.64	289.63
B^R	105/136	248.73	481.89	35.34	206.25	474.35	58.31	9.15	660.35	374.21	316.21
$\mathsf{S}^\exists$	103/136	255.83	71.41	771.20	32.89	292.88	21.49	82.91	656.95	130.63	203.49
B^s	88/136	367.90	451.49	699.70	754.42	89.69	3.13	9.23	104.12	755.56	597.61
$\mathsf{B}_p^\exists$	51/136	584.52	900.00	668.33	900.00	900.00	625.17	2.10	900.00	429.19	118.06

Table 2. Solved instances and average runtimes with preprocessing by *Fast Downward*

closely followed by the $\mathsf{A}^\exists$ setting that uses algorithm A instead of B. While encodings of (relaxed) $\exists$-step plans help to reduce plan lengths, the more restrictive $\forall$-step plans aimed at by $\mathsf{B}^\forall$ are less effective, especially in the *gripper* and *elevator* domains. Apart from a few outliers, the alternative implementations of $\exists$-step plans in $\mathsf{B}^\exists$ and B^E perform comparable, while further reductions by means of relaxed $\exists$-step plans turn out to be minor and cannot compensate for the more sophisticated encoding of B^R. With the exception of the *gripper* domain, the planning heuristic applied by $\mathsf{B}_p^\exists$ significantly boosts search performance, and the last row of Table 1 indicates that this *plasp* setting even comes close to *Madagascar* once preprocessing by *Fast Downward* is used.

Table 2 turns to *plasp* settings run on SAS inputs provided by *Fast Downward*. Beyond information about mutex groups, which is not explicitly available in PDDL, the preprocessing yields multi-valued fluents that conflate several Booleans from a PDDL specification. This makes ground instantiations much more compact than before and significantly increases the number of instances *plasp* can solve. However, a sequential algorithm or encoding as in $\mathsf{S}^\exists$ and B^s remains less effective than the other settings, also with SAS format. Interestingly, the encoding of $\forall$-step plans used in $\mathsf{B}^\forall$ leads to slightly better overall performance than the relaxed $\exists$-step plans of B^R, although variations are domain-specific, as becomes apparent when comparing *gripper* and *logistics*. Given that the parallel plans permitted by B^R are most general, this tells us that the overhead of encoding them does not pay off in the domains at hand, while yet lacking optimizations, e.g., based on disabling–enabling-graphs [17], still leave room for future improvements. As with PDDL inputs, $\exists$-step plans again constitute the best trade-off between benefits and efforts of a parallel encoding, where the planning algorithm of $\mathsf{B}^\exists$ is comparable to $\mathsf{A}^\exists$. The alternative implementation by means of acyclicity checking in B^E improves the performance on the instances in Table 2 to some extent but not dramatically. Unlike that, the planning heuristic of $\mathsf{B}_p^\exists$ leads to substantial performance gains, now also in the *gripper* domain, whose PDDL specification had been problematic for the heuristic before. The large gap in comparison to $\mathsf{B}_p^\exists$ on direct PDDL inputs in the last row confirms the high capacity of preprocessing by *Fast Downward*. Finally, we note that the two *Madagascar* variants remain significantly ahead of the $\mathsf{B}_p^\exists$ setting of *plasp*, which is

related to their streamlined yet planning-specific implementation of grounding, while *plasp* brings the advantage that first-order ASP encodings can be used to prototype and experiment with different features.

5 Summary

We presented the key features of the new *plasp* system. Although it addresses PDDL-based planning, *plasp*'s major components, such as the encodings as well as its planner, can be applied to dynamic domains at large. While our general-purpose approach cannot compete with planning-specific systems at eye level, we have shown how careful encodings and well-engineered solving processes can boost performance. This is reflected, e.g., by an increase of 25 additional instances solved by $\mathsf{B}_p^\exists$ over $\mathsf{S}^\exists$ in Table 1. A further significant impact is obtained by extracting planning-specific constraints, as demonstrated by the increase of the performance of $\mathsf{B}_p^\exists$ from 51 to 121 solved instances in Table 2.

Acknowledgments. This work was partially funded by DFG grant SCHA 550/9.

References

1. C. Baral, M. Gelfond. Reasoning agents in dynamic domains. In *Logic-Based Artificial Intelligence*, 257–279. Kluwer, 2000.
2. A. Biere, M. Heule, H. van Maaren, T. Walsh. *Handbook of Satisfiability*. IOS, 2009.
3. J. Bomanson, M. Gebser, T. Janhunen, B. Kaufmann, T. Schaub. Answer set programming modulo acyclicity. *Fundamenta Informaticae*, 147(1):63–91, 2016.
4. Y. Dimopoulos, B. Nebel, J. Köhler. Encoding planning problems in nonmonotonic logic programs. In *Proceedings ECP*, 169–181. Springer, 1997.
5. M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, P. Wanko. Theory solving made easy with clingo 5. In *Technical Comm. ICLP*, 2:1–2:15. OASIcs, 2016.
6. M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub. *Clingo* = ASP + control: Preliminary report. In *Technical Comm. ICLP*, arXiv:1405.3694, 2014.
7. M. Gebser, R. Kaminski, M. Knecht, T. Schaub. plasp: A prototype for PDDL-based planning in ASP. In *Proceedings LPNMR*, 358–363. Springer, 2011.
8. M. Gebser, B. Kaufmann, R. Otero, J. Romero, T. Schaub, P. Wanko. Domain-specific heuristics in answer set programming. In *Proceedings AAAI*, 350–356. AAAI, 2013.
9. M. Gelfond, V. Lifschitz. Action languages. *Electronic Transactions on Artificial Intelligence*, 3(6):193–210, 1998.
10. M. Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.
11. H. Kautz, D. McAllester, B. Selman. Encoding plans in propositional logic. In *Proceedings KR*, 374–384. Morgan Kaufmann, 1996.
12. V. Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138(1-2):39–54, 2002.
13. D. McDermott. PDDL — the planning domain definition language. TR Yale, 1998.
14. J. Rintanen. Madagascar: Scalable planning with SAT. In *Proc. IPC*, 66–70, 2014.
15. J. Rintanen, K. Heljanko, I. Niemelä. Planning as satisfiability: Parallel plans and algorithms for plan search. *Artificial Intelligence*, 170(12-13):1031–1080, 2006.
16. T. Son, C. Baral, T. Nam, S. McIlraith. Domain-dependent knowledge in answer set planning. *ACM Transactions on Computational Logic*, 7(4):613–657, 2006.
17. M. Wehrle, J. Rintanen. Planning as satisfiability with relaxed ∃-step plans. In *Proceedings AI*, 244–253. Springer, 2007.