plasp 3: Towards Effective ASP Planning

Yannis Dimopoulos

University of Cyprus

Martin Gebser

University of Klagenfurt, Graz University of Technology, and University of Potsdam

Patrick Lühne Javier Romero

University of Potsdam

Torsten Schaub

INRIA Rennes and University of Potsdam

submitted [n/a]; revised [n/a]; accepted [n/a]

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 multivalued fluents in order to capture both PDDL as well as SAS planning formats. Third, enabled by multishot 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.

Under consideration for publication in Theory and Practice of Logic Programming (TPLP)

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 (Dimopoulos et al. 1997) was among the first substantial applications of Answer Set Programming (ASP; (Lifschitz 2002)). Meanwhile this has led to manifold action languages (Gelfond and Lifschitz 1998), various applications in dynamic domains (Baral and Gelfond 2000), but only few adaptions of Automated Planning techniques (Son et al. 2006). Although such approaches have 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; (Biere et al. 2009)), 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 (Gebser et al. 2011a) foresaw to compile planning problems formulated in the Planning Domain Definition Language (PDDL; (McDermott 1998)) into ASP facts and to use ASP metaencodings for modeling alternative planning techniques.

PDDL ⋮ ASP

PDDL specification		**translator**	**ASP facts** **ASP encoding**

Fast Downward preprocessor → **SAS specification** → **translator**
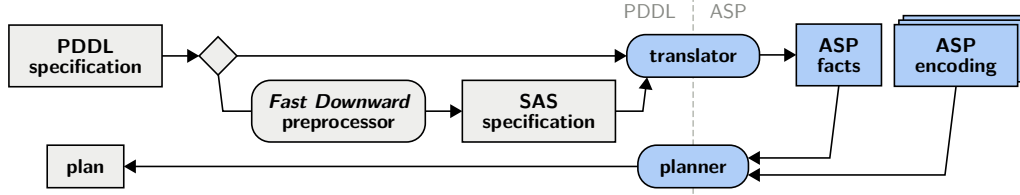
plan ← **planner**

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

These could then be solved with fixed horizons (and optimization) or in an incremental fashion. The redesigned *plasp* 3 system processes planning problems specified in PDDL according to the workflow visualized in Figure 1. At the beginning, a PDDL input may be subject to optional preprocessing by the state-of-the-art planning system *Fast Downward* (Helmert 2006) via the intermediate SAS format. The translator component of *plasp* 3 otherwise performs a normalization step to transform complex PDDL expressions into a simplified core format, which results in a homogeneous factual representation capturing both PDDL and SAS inputs (with multivalued fluents). Moreover, *plasp* 3 provides a spectrum of ASP encodings ranging from adaptions of known SAT encodings (Rintanen et al. 2006) to novel encodings taking advantage of ASP-specific concepts. Finally, the planner component of *plasp* 3 offers sophisticated planning algorithms, also stemming from SAT planning (Rintanen et al. 2006), by taking advantage of multishot ASP solving. Given the common structure of various incremental ASP encodings, *plasp*'s planning framework is also applicable to dynamic domains beyond PDDL, e.g., the planner could be run on ASP encodings of finite model finding (Gebser et al. 2011b) instead of planning.

The outline of this paper is as follows. In Section 2, we introduce STRIPS-like planning tasks and devise ASP encodings for sequential as well as a range of parallel representations of plans. Section 3 is dedicated to the planner component of *plasp* 3, presenting the planning algorithms, guess-and-check strategies, and the planning heuristic it supports. In Section 4, we turn to the functionalities provided by *plasp*'s translator, including normalization for dealing with advanced PDDL features and the handling of constructs comprised in the intermediate SAS format. Section 5 reports about our experiments, empirically evaluating the devised encodings and planning algorithms on PDDL inputs as well as ASP planning benchmarks. Finally, Section 6 concludes the paper with a summary of the achieved results and future work. This paper extends a previous conference version (Dimopoulos et al. 2017), which did not include the guess-and-check strategies presented in Section 3, the description of the translator component of *plasp* 3 given in Section 4, and experimental results on the benchmark set by (Rintanen 2012) as well as ASP planning benchmarks.

2 ASP Encodings for Planning

We consider STRIPS-like (multivalued) *planning tasks* according to (Helmert 2006), 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 $x \in \tilde{s}_\star$, i.e., the goal conditions specified by $s_\star$ have to hold in s'.

Several *parallel* representations of sequential plans have been investigated in the literature (Dimopoulos et al. 1997; Rintanen et al. 2006; Wehrle and Rintanen 2007). 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 in s, 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 $\langle A_1, \ldots, A_m \rangle$ of confluent sets of actions 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 A_i 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. Also note that, in case of (relaxed) $\exists$-step plans, the confluence requirement achieves tractability of deciding whether a set of actions is (relaxed) $\exists$-step serializable, which becomes NP-hard otherwise (Rintanen et al. 2006).

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 $\langle \{a_1\}, \{a_2\}, \{a_3, a_4\} \rangle$.

Listing 1. ASP fact representation of the planning task from Example 1

```
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).
```

Listing 2. 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 :- holds(X,V,t), not holds(X,V,t-1), not occurs(A,t) : post(A,X,V).
```

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 from Example 1 by facts as given in Listing 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 2 form the common core of respective incremental encodings (Gebser et al. 2014) 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 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.

Listing 3. Extension of Listing 2 for encoding sequential plans

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

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

In more detail, the rule in Line 1 of Listing 2 maps facts specifying s_0 to atoms over the predicate holds/3, in which the third argument 0 refers to the initial 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 the respective postcondition(s), which guarantees the confluence of any set of actions to be applied in parallel. On the other hand, fluents unaffected by applied actions must remain unchanged, and the integrity constraint in Line 15 thus requires changed fluents to be established by means of applied actions.[1]

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 3 address *sequential plans* by, in Line 17, asserting the precondition of an applied action to hold in the state referred to by t-1 and, in Line 19, 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 2 and 3 permits idle states in which no action is applied. While idle states cannot emerge when using the basic *iclingo* control loop (Gebser et al. 2014) 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 4 shows additions dedicated to ∀-*step plans*, where the integrity constraint in Line 17 is the same as in Listing 3 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 of Listing 2. It thus remains to make sure that applied actions do not interfere in any way that would disable a serialization, which

---

[1]The variables X and V occur outside the scope of *conditional literals*, composed by the ':' connective, and are thus *global* in Line 15, while A is *local* to 'not occurs(A,t) : post(A,X,V)'. An according instantiation of the integrity constraint in Line 15, taking the values x1 and 1 for X and V relative to the facts in Listing 1, is ':- holds(x1,1,t), not holds(x1,1,t-1), not occurs(a1,t), not occurs(a2,t).' Note that the actions a1 and a2, which have $x_1 = 1$ in their postconditions, are taken as values for the local variable A, where the resulting instances of the literal 'not occurs(A,t)' on the left-hand side of ':' are connected conjunctively in a propositional rule with the constant t as placeholder for integers. For a detailed account of the language of the ASP system *clingo*, we refer the reader to (Gebser et al. 2017a).

Listing 4. Extension of Listing 2 for encoding ∀-step plans

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

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

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

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 19, 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 $a'^e(x) \neq a^c(x)$, while the integrity constraint in Line 13 readily requires x to remain unchanged in case $a^e(x) = a^c(x)$. 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 21 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 22. 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 2 and 3, 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 4 instead of the one in Listing 3, 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 $\{a_1, a_2\}$ 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 2 addressing ∃-*step plans* are given in Listing 5. As before, the integrity constraint in Line 17 is included to assert the precondition of an applied action to hold in 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 19–20 expresses that an action can be safely

Listing 5. Extension of Listing 2 for encoding ∃-step plans

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

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

22  ready(A,t) :- action(A), not occurs(A,t).
23  ready(A,t) :- apply(A,t).
24  :- action(A), not ready(A,t).
```

Listing 6. Replacement of Lines 19–24 in Listing 5 by an `#edge` directive

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

Listing 7. Extension of Listing 2 for encoding relaxed ∃-step plans

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

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

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

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 22 and 23. In fact, the least fixpoint obtained via the rules in Lines 19–23 covers all actions if and only if the applied actions do not circularly invalidate their preconditions, and the integrity constraint in Line 24 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 (Gebser et al. 2016) of *clingo*, in which case built-in acyclicity checking (Bomanson et al. 2016) is used. A respective replacement of Lines 19–24 is shown in Listing 6, where the `#edge` directive in Lines 19–20 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 7 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, 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 20–24 are almost identical to their counterparts in Listing 5, and the difference amounts to the additional prerequisite 'reach(X,V,t) : prec(A1,X,V)' in Line 20. Instances of reach(X,V,t) are derived by means of the rules in Lines 17 and 18 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 20–21 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.[2] Similar to its counterpart in Listing 5, the integrity constraint in Line 25 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 2 along with Listing 5 or 6 include occurs(a_1,1), occurs(a_2,1), occurs(a_3,2), and occurs(a_4,2). Regarding the #edge directive in Listing 6, 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 5, 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 5 and 6, 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 2 and 7 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 2, $Q(i)$ 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 3, $S^\forall(i)$ for Listing 4, $S^\exists(i)$ for Listing 5, $S^E(i)$ for Line 17 of Listing 5 along with Listing 6, and $S^R(i)$ for Listing 7. Given that $S^E(i)$ includes an

---

[2]The ';' symbol in Line 20 separates the (conjunctively connected) conditional literals 'reach(X,V,t) : prec(A1,X,V)' and 'ready(A2,t) : post(A1,X,V1), prec(A2,X,V2), A1 != A2, V1 != V2'.

`#edge` directive subject to acyclicity checking, we understand stable models in the sense of (Bomanson et al. 2016), 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 a 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

$$I \cup B \cup Q(0) \cup \bigcup_{i=1}^{n}(Q(i) \cup S(i) \cup S^s(i)) \cup \{\texttt{query}(n).\}$$

  has a stable model M such that $\{\langle a, i \rangle \mid \texttt{occurs}(a, i) \in M\} = \{\langle a_i, i \rangle \mid 1 \le i \le n\}$;
- $\langle A_1, \ldots, A_m \rangle$ is a $\forall$-step (resp., $\exists$-step or relaxed $\exists$-step) plan iff

$$I \cup B \cup Q(0) \cup \bigcup_{i=1}^{m}(Q(i) \cup S(i) \cup S^p(i)) \cup \{\texttt{query}(m).\}$$

  with $S^p(i) = S^\forall(i)$ (resp., $S^p(i) \in \{S^\exists(i), S^E(i)\}$ or $S^p(i) = S^R(i)$) has a stable model M such that $\{\langle a, i \rangle \mid \texttt{occurs}(a, i) \in M\} = \{\langle a, i \rangle \mid 1 \le i \le m, a \in A_i\}$.

Proof

We distinguish sequential and parallel representations of plans.

- Let

$$P^s(n) \;=\; I \cup B \cup Q(0) \cup \bigcup_{i=1}^{n}(Q(i) \cup S(i) \cup S^s(i)) \cup \{\texttt{query}(n).\}.$$

  ($\Rightarrow$) If $\langle a_1, \ldots, a_n \rangle$ is a sequential plan, for any stable model M of $P^s(n)$ such that $\{\langle a, i \rangle \mid \texttt{occurs}(a, i) \in M\} = \{\langle a_i, i \rangle \mid 1 \le i \le n\}$, the rule in B (Line 1 of Listing 2) together with the choice rule in Line 9 of Listing 2, included in $S(i)$, and the integrity constraints in $S(i)$ (Lines 13 and 15 of Listing 2) make sure that

$$\{\langle x, v, i \rangle \mid \texttt{holds}(x, v, i) \in M\} = \{\langle x, v, i \rangle \mid x \in \mathcal{F}, 0 \le i \le n, o(\langle a_1, \ldots, a_i \rangle, s_0)(x) = v\}.$$

  Hence, $P^s(n)$ cannot have any stable model apart from

$$\begin{aligned}
M = I &\cup \{\texttt{occurs}(a_i, i) \mid 1 \le i \le n\} \\
&\cup \{\texttt{holds}(x, v, i) \mid x \in \mathcal{F}, 0 \le i \le n, o(\langle a_1, \ldots, a_i \rangle, s_0)(x) = v\} \\
&\cup \{\texttt{query}(n)\}
\end{aligned}$$

  such that $\{\langle a, i \rangle \mid \texttt{occurs}(a, i) \in M\} = \{\langle a_i, i \rangle \mid 1 \le i \le n\}$. As one can check, M satisfies all rules and integrity constraints in $P^s(n)$, so that M is indeed a stable model of $P^s(n)$.

  ($\Leftarrow$) If M is a stable model of $P^s(n)$ such that $\{\langle a, i \rangle \mid \texttt{occurs}(a, i) \in M\} = \{\langle a_i, i \rangle \mid 1 \le i \le n\}$, the rule in B establishes that

$$\{\langle x, v \rangle \mid \texttt{holds}(x, v, 0) \in M\} = \{\langle x, v \rangle \mid x \in \mathcal{F}, s_0(x) = v\}.$$

  For any $1 \le i \le n$, assume that

$$\{\langle x, v \rangle \mid \texttt{holds}(x, v, i-1) \in M\} = \{\langle x, v \rangle \mid x \in \mathcal{F}, o(\langle a_1, \ldots, a_{i-1} \rangle, s_0)(x) = v\}.$$

  Then, the integrity constraint in Line 17 of Listing 3, included in $S^s(i)$, guarantees that $o(a_i, o(\langle a_1, \ldots, a_{i-1} \rangle, s_0))$ is defined. Moreover, the choice rule in Line 9 of Listing 2 and the integrity constraints in $S(i)$ make sure that

$$\{\langle x, v \rangle \mid \texttt{holds}(x, v, i) \in M\} = \{\langle x, v \rangle \mid x \in \mathcal{F}, o(\langle a_1, \ldots, a_i \rangle, s_0)(x) = v\}.$$

Finally, given the fact $\texttt{query}(n)$, the integrity constraint in $Q(n)$ (Line 5 of Listing 2) yields that $o(\langle a_1,\ldots,a_n\rangle,s_0)(x)=s_\star(x)$ for each $x\in\tilde{s}_\star$, so that $\langle a_1,\ldots,a_n\rangle$ is a sequential plan.

- Let

$$P^p(m) \;=\; I\cup B\cup Q(0)\cup\textstyle\bigcup_{i=1}^m(Q(i)\cup S(i)\cup S^p(i))\cup\{\texttt{query}(m)\,.\}$$

where $p\in\{\forall,\exists,E,R\}$.

$(\Rightarrow)$ If $\langle A_1,\ldots,A_m\rangle$ is a $\forall$-step (resp., $\exists$-step or relaxed $\exists$-step) plan, for $1\le i\le m$, let s_i be the state such that $s_i(x)=a^e(x)$ for each $a\in A_i$ and $x\in\tilde{a}^e$, and $s_i(x)=s_{i-1}(x)$ for each $x\in\mathcal{F}\setminus\bigcup_{a\in A_i}\tilde{a}^e$. Then, for any stable model M of $P^p(m)$ such that $\{\langle a,i\rangle\mid\texttt{occurs}(a,i)\in M\}=\{\langle a,i\rangle\mid 1\le i\le m,a\in A_i\}$, the rule in B (Line 1 of Listing 2) together with the choice rule in Line 9 of Listing 2, included in $S(i)$, and the integrity constraints in $S(i)$ (Lines 13 and 15 of Listing 2) make sure that

$$\{\langle x,v,i\rangle\mid\texttt{holds}(x,v,i)\in M\}=\{\langle x,v,i\rangle\mid x\in\mathcal{F},0\le i\le m,s_i(x)=v\}.$$

In the following, we consider the different kinds of plans.

$\forall$**-step:** Assume that $\langle A_1,\ldots,A_m\rangle$ is a $\forall$-step plan. Given that an atom of the form $\texttt{single}(x,i)$ is derived by the rule in $S^\forall(i)$ (Line 21 of Listing 4) iff there is some $a\in A_i$ such that $x\in\tilde{a}^c\cap\tilde{a}^e$ and $a^e(x)\ne a^c(x)$ for $1\le i\le m$, the program $P^\forall(m)$ cannot have any stable model apart from

$$\begin{aligned}
M = \;&I\cup\{\texttt{occurs}(a,i)\mid 1\le i\le m,a\in A_i\}\\
&\cup\{\texttt{holds}(x,v,i)\mid x\in\mathcal{F},0\le i\le m,s_i(x)=v\}\\
&\cup\{\texttt{single}(x,i)\mid 1\le i\le m,a\in A_i,x\in\tilde{a}^c\cap\tilde{a}^e,a^e(x)\ne a^c(x)\}\\
&\cup\{\texttt{query}(m)\}
\end{aligned}$$

such that $\{\langle a,i\rangle\mid\texttt{occurs}(a,i)\in M\}=\{\langle a,i\rangle\mid 1\le i\le m,a\in A_i\}$. As for $1\le i\le m$, $o(\langle a_1,\ldots,a_k\rangle,s_{i-1})$ is defined for any sequence $\langle a_1,\ldots,a_k\rangle$ such that $\{a_1,\ldots,a_k\}=A_i$, we have that M satisfies the integrity constraints in $S^\forall(i)$ (Lines 17, 19, and 22 of Listing 4), and one can check that further rules and integrity constraints in $P^\forall(m)$ are satisfied as well, so that M is indeed a stable model of $P^\forall(m)$.

$\exists$**-step:** Assume that $\langle A_1,\ldots,A_m\rangle$ is an $\exists$-step plan. Then, for any $1\le i\le m$, there is some sequence $\langle a_1,\ldots,a_k\rangle$ such that $\{a_1,\ldots,a_k\}=A_i$ and $a_j^e(x)=a_{j'}^c(x)$ if $x\in\tilde{a}_{j'}^c\cap\tilde{a}_j^e$ for $1\le j<j'\le k$. That is, the edges of the graph

$$G_i \;=\; (\mathcal{O},\{\langle a,a'\rangle\mid a\in A_i,a'\in\mathcal{O}\setminus\{a\},x\in\tilde{a}'^c\cap\tilde{a}^e,a^e(x)\ne a'^c(x)\})$$

can connect some a_j for $1\le j\le k$ to elements of $\mathcal{O}\setminus\{a_j,\ldots,a_k\}$ only, so that G_i is acyclic. This implies that the graph induced by

$$\begin{aligned}
M = \;&I\cup\{\texttt{occurs}(a,i)\mid 1\le i\le m,a\in A_i\}\\
&\cup\{\texttt{holds}(x,v,i)\mid x\in\mathcal{F},0\le i\le m,s_i(x)=v\}\\
&\cup\{\texttt{query}(m)\}
\end{aligned}$$

in view of the `#edge` directive in Lines 19–20 of Listing 6, which is the disjoint union of graphs G_i for $1\le i\le m$, is acyclic as well. Moreover, the program $P^E(m)$ cannot have any stable model apart from M such that $\{\langle a,i\rangle\mid\texttt{occurs}(a,i)\in M\}=\{\langle a,i\rangle\mid 1\le i\le m,a\in A_i\}$, and one can check that M satisfies all rules and integrity constraints in $P^E(m)$, so that M is indeed a stable model of $P^E(m)$. Regarding

$P^{\exists}(m)$, the acyclicity of G_i yields that $\mathtt{apply}(a,i)$ and $\mathtt{ready}(a,i)$ belong to the least fixpoint of the rules in $S^{\exists}(i)$ (Lines 19–23 of Listing 5) for each $a \in \mathcal{O}$ and $1 \le i \le m$, and thus $P^{\exists}(m)$ cannot have any stable model apart from

$$M' = M \cup \{\mathtt{apply}(a,i) \mid a \in \mathcal{O}, 1 \le i \le m\}$$
$$\cup \{\mathtt{ready}(a,i) \mid a \in \mathcal{O}, 1 \le i \le m\}$$

such that $\{\langle a,i \rangle \mid \mathtt{occurs}(a,i) \in M'\} = \{\langle a,i \rangle \mid 1 \le i \le m, a \in A_i\}$. As one can check, M' satisfies all rules and integrity constraints in $P^{\exists}(m)$, so that M' is indeed a stable model of $P^{\exists}(m)$.

relaxed $\exists$-step: Assume that $\langle A_1, \ldots, A_m \rangle$ is a relaxed $\exists$-step plan. Then, for any $1 \le i \le m$, there is some sequence $\langle a_1, \ldots, a_k \rangle$ such that $\{a_1, \ldots, a_k\} = A_i$ and $o(a_j, o(\langle a_1, \ldots, a_{j-1} \rangle, s_{i-1}))$ is defined for $1 \le j \le k$, which implies that $a_j^e(x) = a_{j'}^c(x)$ if $x \in \tilde{a}_j^c \cap \tilde{a}_j^e$ for $j < j' \le k$. Given this, the least fixpoint of the rules in $S^R(i)$ (Lines 17–24 of Listing 7) contains $\mathtt{reach}(x,v,i)$ for each $x \in \mathcal{F}$ and $v \in \{s_{i-1}(x), s_i(x)\}$, $\mathtt{apply}(a,i)$ for each $a \in \mathcal{O}$ such that $a^c(x) \in \{s_{i-1}(x), s_i(x)\}$ for all $x \in \tilde{a}^c$, as well as $\mathtt{ready}(a,i)$ for each $a \in \mathcal{O}$, and thus $P^R(m)$ cannot have any stable model apart from

$$
\begin{aligned}
M = I &\cup \{\mathtt{occurs}(a,i) \mid 1 \le i \le m, a \in A_i\} \\
&\cup \{\mathtt{holds}(x,v,i) \mid x \in \mathcal{F}, 0 \le i \le m, s_i(x) = v\} \\
&\cup \{\mathtt{reach}(x,v,i) \mid x \in \mathcal{F}, 1 \le i \le m, v \in \{s_{i-1}(x), s_i(x)\}\} \\
&\cup \{\mathtt{apply}(a,i) \mid a \in \mathcal{O}, 1 \le i \le m, a^c(x) \in \{s_{i-1}(x), s_i(x)\} \text{ for all } x \in \tilde{a}^c\} \\
&\cup \{\mathtt{ready}(a,i) \mid a \in \mathcal{O}, 1 \le i \le m\} \\
&\cup \{\mathtt{query}(m)\}
\end{aligned}
$$

such that $\{\langle a,i \rangle \mid \mathtt{occurs}(a,i) \in M\} = \{\langle a,i \rangle \mid 1 \le i \le m, a \in A_i\}$. As one can check, M satisfies all rules and integrity constraints in $P^R(m)$, so that M is indeed a stable model of $P^R(m)$.

($\Leftarrow$) If M is a stable model of $P^p(m)$ such that $\{\langle a,i \rangle \mid \mathtt{occurs}(a,i) \in M\} = \{\langle a,i \rangle \mid 1 \le i \le m, a \in A_i\}$, for any $1 \le i \le m$, the choice rule in Line 9 of Listing 2 makes sure that s_i such that $s_i(x) = v$ iff $\mathtt{holds}(x,v,i) \in M$ is a state. Moreover, the rule in B together with the integrity constraints in $S(i)$ yield that $s_i(x) = a^e(x)$ for each $a \in A_i$ and $x \in \tilde{a}^e$, which implies that A_i is confluent, and $s_i(x) = s_{i-1}(x)$ for each $x \in \mathcal{F} \setminus \bigcup_{a \in A_i} \tilde{a}^e$. In view of the fact $\mathtt{query}(m)$, the integrity constraint in $Q(m)$ (Line 5 of Listing 2) further asserts that $s_m(x) = s_\star(x)$ for each $x \in \tilde{s}_\star$. It remains to show that $A_i = \{a_1, \ldots, a_k\}$ is $\forall$-step (resp., $\exists$-step or relaxed $\exists$-step) serializable in s_{i-1} for $p = \forall$ (resp., $p \in \{\exists, E\}$ or $p = R$).

$p = \forall$**:** Let $\langle a_1, \ldots, a_k \rangle$ be some sequence such that $\{a_1, \ldots, a_k\} = A_i$. Then, the integrity constraint in Line 17 of Listing 4 guarantees that $o(a_j, s_{i-1})$ is defined for $1 \le j \le k$, and assume that $o(\langle a_1, \ldots, a_{j-1} \rangle, s_{i-1})$ is defined as well. For any $x \in \tilde{a}_j^c \cap \bigcup_{1 \le j' < j} \tilde{a}_{j'}^e$, we have that $o(\langle a_1, \ldots, a_{j-1} \rangle, s_{i-1})(x) = s_i(x)$. If $x \notin \tilde{a}_j^e$, the integrity constraint in Line 19 of Listing 4 yields that $s_i(x) = a_j^c(x) = s_{i-1}(x)$. Otherwise, if $x \in \tilde{a}_j^e$, the integrity constraint in Line 22 of Listing 4 implies that $\mathtt{single}(x,i) \notin M$, which means that the prerequisites in the instance

```
single(x,i) :- occurs(a_j,i), prec(a_j,x,v_1), post(a_j,x,v_2), v_1 != v_2.
```

of the rule in Line 21 of Listing 4 cannot hold and $s_i(x) = a_j^e(x) = a_j^c(x) = s_{i-1}(x)$

must be the case. This shows that $o(\langle a_1,\ldots,a_{j-1}\rangle, s_{i-1})(x) = a_j^c(x)$, so that $o(\langle a_1,\ldots,a_j\rangle, s_{i-1})$ and, in particular, $o(\langle a_1,\ldots,a_k\rangle, s_{i-1})$ is defined.

$p \in \{\exists, E\}$**:** For any $a \in A_i$, the integrity constraint in Line 17 of Listing 5 guarantees that $o(a, s_{i-1})$ is defined. Regarding $P^E(m)$, the `#edge` directive in Lines 19–20 of Listing 6 makes sure that the graph

$$G_i \;=\; (\mathcal{O}, \{\langle a, a'\rangle \mid a \in A_i, a' \in \mathcal{O} \setminus \{a\}, x \in \tilde{a}'^c \cap \tilde{a}^e, a^e(x) \neq a'^c(x)\})$$

is acyclic. Hence, there is some sequence $\langle a_1,\ldots,a_k\rangle$ such that $\{a_1,\ldots,a_k\} = A_i$ and no edge connects any element of $\{a_1,\ldots,a_{j-1}\}$ to a_j in G_i for $1 \leq j \leq k$. This yields that $a_{j'}^e(x) = a_j^c(x) = s_{i-1}(x)$ if $x \in \tilde{a}_j^c \cap \tilde{a}_{j'}^e$ for $1 \leq j' < j$, so that $o(\langle a_1,\ldots,a_j\rangle, s_{i-1})$ and, in particular, $o(\langle a_1,\ldots,a_k\rangle, s_{i-1})$ is defined. Concerning $P^{\exists}(m)$, the integrity constraint in Line 24 of Listing 5 implies that $\texttt{ready}(a,i) \in M$ for each $a \in \mathcal{O}$. The rules in Lines 22 and 23 of Listing 5 further yield that $\texttt{apply}(a,i) \in M$ for each $a \in A_i$. Given the rule in Lines 19–20 of Listing 5, for any $1 \leq j \leq k$, there must be some $a_j \in A_i$ such that $\{a_{j'} \in A_i \mid j < j' \leq k, x \in \tilde{a}_{j'}^c \cap \tilde{a}_j^e, a_j^e(x) \neq a_{j'}^c(x)\} = \emptyset$. In turn, we have that $a_{j'}^e(x) = a_j^c(x) = s_{i-1}(x)$ if $x \in \tilde{a}_j^c \cap \tilde{a}_{j'}^e$ for $1 \leq j' < j$, so that $o(\langle a_1,\ldots,a_j\rangle, s_{i-1})$ and, in particular, $o(\langle a_1,\ldots,a_k\rangle, s_{i-1})$ is defined.

$p = R$**:** In view of the rule in Line 17 of Listing 7, we have that $\texttt{reach}(x,v,i) \in M$ if $s_{i-1}(x) = v$. The integrity constraint in Line 25 of Listing 7 further implies that $\texttt{ready}(a,i) \in M$ for each $a \in \mathcal{O}$, and the rules in Lines 23 and 24 of Listing 7 yield that $\texttt{apply}(a,i) \in M$ for each $a \in A_i$. Given the rules in Lines 18 and 20–21 of Listing 7, for any $1 \leq j \leq k$, there must be some $a_j \in A_i$ such that $\{a_{j'} \in A_i \mid j < j' \leq k, x \in \tilde{a}_{j'}^c \cap \tilde{a}_j^e, a_j^e(x) \neq a_{j'}^c(x)\} = \emptyset$ and $a_j^c(x) \in \{s_{i-1}(x)\} \cup \{a_{j'}^e(x) \mid 1 \leq j' < j, x \in \tilde{a}_{j'}^e\}$ for all $x \in \tilde{a}_j^c$. Hence, for each $x \in \tilde{a}_j^c$, we have that $a_j^c(x) = s_{i-1}(x)$ and $x \notin \bigcup_{1 \leq j' < j} \tilde{a}_{j'}^e$ or $a_{j'}^e(x) = a_j^c(x)$ if $x \in \tilde{a}_{j'}^e$, for $1 \leq j' < j$, so that $o(\langle a_1,\ldots,a_j\rangle, s_{i-1})$ and, in particular, $o(\langle a_1,\ldots,a_k\rangle, s_{i-1})$ is defined. ∎

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 $\texttt{occurs}/2$, 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 7, 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 21 of Listing 7 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 check the noncircularity of establishment and invalidation of preconditions in separation, no matter the respective implementation techniques.

As regards encoding techniques, common ASP-based approaches, e.g., (Lifschitz 2002), define successor states, i.e., the predicate $\texttt{holds}/3$, in terms of actions given by atoms over
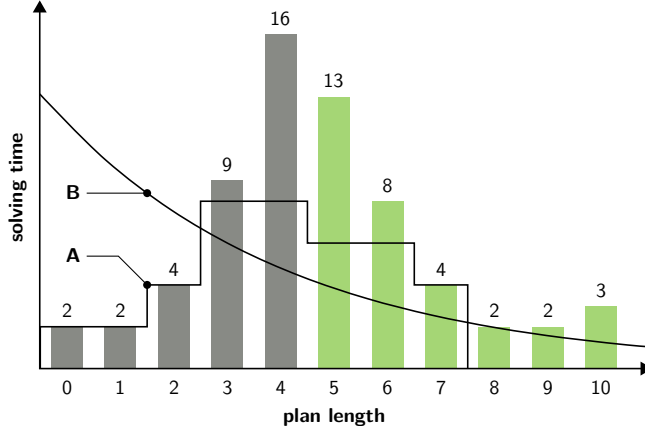
Fig. 2. Exemplary solving times required by the planning algorithms A and B

occurs/2. Listing 2, 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 (Kautz et al. 1996). Concerning (relaxed) ∃-step plans, the encoding parts in Listings 5 and 7 make use of the built-in well-foundedness requirement in ASP and do, unlike (Rintanen et al. 2006), not unfold the order of actions applied in parallel. In contrast to the SAT approach to relaxed ∃-step plans in (Wehrle and Rintanen 2007), we do not rely on a fixed (static) order of actions, and to our knowledge, no encoding similar to the one in Listing 7 has been proposed so far.

3 A Multishot 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 (Rintanen et al. 2006), 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 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, (Rintanen et al. 2006) suggests not to minimize parallel plan length, but rather make use of *planning algorithms* that avoid costly unsatisfiable parts by moving early to easier satisfiable lengths.

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 (Rintanen et al. 2006) 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, after 4 more units per length from 3 to 7, 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 (Rintanen et al. 2006) distributes time nonuniformly 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, and the amount t of time spent on the initially shortest length 0 is thus multiplied by 0.8^i for lengths $i \geq 1$. Note that, in practice, only lengths whose assigned time is above some threshold are indeed run, which restricts the plan lengths to consider simultaneously. While searching for a plan, the shortest unfinished length n and its assigned time t successively increase, so that $t*\gamma^i$ grows beyond the threshold of running for greater plan lengths $n+i$. Regarding our example with $\gamma = 0.8$, when length 3 has been run for 6 time 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 ($\lceil 6*0.8^5 \rceil$) 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 (yielding S when n is set to 1) and B, and implemented them as planning strategies of *plasp* via multishot 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 (Gebser et al. 2014), 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 $P(n) \cup \{\texttt{query}(n).\}$ is satisfiable, and algorithms S, A, and B provide different strategies to approach such an integer.

The planner[3] is implemented using *clingo*'s multishot solving capacities, where a *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 query(m) is set to true instead of query(n). That is, the search component of *clingo* has to establish conditions in check(m), even though the encoding is unrolled up to $n \geq m$. For this approach to work, we require that $P(m) \cup \{\texttt{query}(m).\}$ is satisfiable if and only if $P(n) \cup \{\texttt{query}(m).\}$ 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.

While planning algorithms tackle the issue of finding a sufficient plan length, the choice of an underlying planning encoding remains, i.e., whether to take $S(i) \cup S^s(i)$, $S(i) \cup S^\forall(i)$, $S(i) \cup S^\exists(i)$, $S(i) \cup S^E(i)$, or $S(i) \cup S^R(i)$ according to the terminology used in Theorem 1 for the subprogram step(i) of $P(n)$. On the one hand, the encoding $S(i) \cup S^R(i)$ of relaxed $\exists$-step plans is guaranteed to become satisfiable first, where the minimal parallel plan

---

[3] https://github.com/potassco/planner

length may still coincide with the sequential encoding $S(i) \cup S^s(i)$ in the "worst" case of an inherently sequential planning task. On the other hand, parallel encodings introduce overhead for checking the existence of a compatible serialization. This particularly applies to $S^\exists(i)$, $S^E(i)$, and $S^R(i)$, aiming at (relaxed) $\exists$-step plans, as the conditions they encode refer to pairs of actions whose preconditions and postconditions interfere. Such quadratic behavior is problematic for planning tasks involving a large number of actions, and our experiments in Section 5 indeed incorporate domains where instances yield several thousand actions. As a consequence, it is sometimes desirable to keep the efforts of checking whether a set of actions is serializable low. Moreover, investigations of common benchmark domains for planning systems (Rintanen et al. 2006; Rintanen 2012) showed that circular interference is in many cases impossible, so that some serialization will exist for any (confluent) set of actions. This observation along with the aforementioned efficiency considerations regarding the (ground) representation of serialization conditions motivate us to augment the planner with *guess-and-check* facilities, detailed in the following.

The general idea of the guess-and-check approach (Eiter and Polleres 2006) is to encode a problem by a pair $\langle G, C \rangle$ of programs, where a stable model M of G constitutes a solution if $C \cup M$ is unsatisfiable. In the context of ASP planning, the role of G is to generate stable models providing sequences of sets of actions, and C checks whether some serialization yields a sequential plan. For making "educated" guesses, the program G we propose combines facts representing a planning task with the incremental encoding in Listing 2 and the integrity constraint in Line 17, shared by Listings 3–5, for asserting the preconditions of applied actions. As a consequence, a stable model M of G is such that all preconditions and postconditions hold for a set of actions to be applied in parallel, which also makes sure that the set is confluent, while (the absence of) circular interference remains to be checked. The latter can be accomplished by taking the atoms over `occurs/2` from M as facts together with a program C comprising the fact representation of a planning task, the rules in Lines 19–23 of Listing 5, and an encoding part as follows:

```
#program step(t).

cycle(t) :- action(A), not ready(A,t).

#program check(t).

:- query(t), not cycle(T) : T = 1..t.
```

Note that instances of the rule in the `step(t)` subprogram yield `cycle(t)` if applied actions are pending, i.e., the respective instances of `ready(A,t)` remain underivable, as each such action invalidates another pending action's precondition. In fact, an atom `cycle(`i`)` means that the set of actions a such that `ready(`a,i`)` does not hold is not (relaxed) $\exists$-step serializable in any state. Given this, the integrity constraint in `check(t)` requires that some set of actions is not $\exists$-step serializable in view of a circular invalidation of preconditions. Letting $m = \max\{i \,|\, \texttt{occurs}(a,i) \in M\}$, a stable model of $C \cup \{\texttt{occurs}(a,i).\,|\,\texttt{occurs}(a,i) \in M\} \cup \{\texttt{query}(m).\}$ thus tells us that the sequence of sets of actions from M is not (relaxed) $\exists$-step serializable, while the absence of stable models indicates the existence of a compatible serialization.

In case the program C together with facts for some sequence $\langle A_1, \dots, A_m \rangle$ of actions from a stable model M of G is satisfiable, the corresponding stable model M' is such that

$A_i' = \{a \in A_i \mid \mathtt{ready}(a,i) \notin M'\}$ is nonempty for at least one $1 \le i \le m$. Given that each nonempty A_i' is not (relaxed) $\exists$-step serializable, a guess-and-check control loop could augment G with constraints suppressing a parallel application of the actions in A_i' to eliminate M, but not any $\exists$-step plan, and search for another stable model instead. In fact, we have tried several options to utilize the information from a stable (counter-)model M', i.e., extending G with constraints that deny a parallel application of A_i' and supersets thereof at the ith or all positions of a sequence of actions, respectively, in order to generate alternative sequences. However, we found that the domains used for our experiments in Section 5 belong to two rather extreme categories: either the sequence of actions generated first directly yields a compatible serialization, given that the available actions cannot interfere, or a vast number of sequences that are not $\exists$-step serializable is successively generated, so that denying the parallel application of particular sets of actions turns out to be ineffective. Hence, switching from the program G given above to one of the parallel encodings provided in Section 2, in case the sequence of sets of actions generated first is not $\exists$-step serializable, constitutes a better option to implement the guess-and-check approach. In Section 5, we particularly investigate the strategy to switch to the encoding of $\forall$-step plans in Listing 4, as it avoids the aforementioned issue of referring to pairs of actions to express serialization conditions. Technically, the switch from G to the encoding of $\forall$-step plans is accomplished by including Lines 19–22 of Listing 4 in a separate subprogram instead of `step(t)`, which is then instantiated for the same integers starting from `1` as used for `step(t)` in case the first sequence of sets of actions obtained with G happens to be not $\exists$-step serializable. While the guess-and-check strategies discussed here aim at an efficient problem representation by skipping serialization conditions unless they are needed, we note that other applications, e.g., in conformant or temporal planning (Cimatti et al. 2008; Fisher 2008), may also harness the approach to perform more sophisticated checks.

Our planner is further equipped with a planning-specific *heuristic*, inspired by (Rintanen 2012) and devised in (Gebser et al. 2013) within a framework for domain-specific heuristics in ASP. The general idea is to extend the search heuristic of *clingo* by associating each atom with a *level* (0 by default) and a *sign*, which can be undefined (by default), true, or false. During solving, these values get modified by the activation of **#heuristic** directives, and the search heuristic of *clingo* then selects an atom at the highest level and sets it to true or false according to its associated sign, switching to the default sign heuristic if that sign is undefined. The domain-specific heuristic for planning, which aims at propagating fluent values backwards in time, starting from those in the goal of a planning task, is specified in terms of **#heuristic** directives as follows:

```
#program step(t).

#heuristic holds(X,V,t-1) :     holds(X,V,t). [2147483647-t, true]
#heuristic holds(X,V,t-1) : not holds(X,V,t). [2147483647-t,false]
```

The first directive expresses that, whenever an instance of the atom `holds(X,V,t)` is made true by *clingo*, `holds(X,V,t-1)` should be set to true at the level `2147483647-t`, where `2147483647` happens to be the maximum integer supported by *clingo*. Similarly, the second directive is activated when an instance of `holds(X,V,t)` becomes false, in which case `holds(X,V,t-1)` should be made false as well. Both directives associate atoms over `holds/3` representing earlier states, i.e., the integer taken for `t` is smaller, with higher

levels, which intends to bias the search of *clingo* towards establishing goal conditions as early as possible. As the experiments in Section 5 show, the use of a planning heuristic often helps to improve plan search.

4 Translating PDDL to ASP

Like its predecessors, the third series of *plasp*[4] furnishes a translator from PDDL specifications to ASP facts. These facts are then combined with ASP encodings, such as those provided in Section 2, and solved by an off-the-shelf ASP system, for example, by using the planner presented in Section 3. However, the translator integrated in *plasp* 3 also goes beyond the STRIPS fragment by supporting a range of advanced features from PDDL 3.1 (IPC 2014). Such advanced features include conditional effects and logical connectives as well as quantifiers in preconditions, postconditions, and goals.[5]

To begin with, *plasp* parses a PDDL specification into an abstract syntax tree, which is then subject to a *normalization* step in order to reduce the range of expressions handled in the actual translation to ASP facts. For instance, implications $\phi \rightarrow \psi$ are mapped to disjunctions $\neg\phi \vee \psi$, and universal quantification $\forall x_1 \ldots x_n : \phi$ is turned into $\neg\exists x_1 \ldots x_n : \neg\phi$. The latter allows for eliminating universal quantifiers, as also done by *Fast Downward* (Helmert 2006). As a result, input expressions are brought into a simplified format akin to negation normal form, except that existential quantifiers may deliberately be subject to negation.

Similar to the introduction of Tseitin variables in transforming a formula into conjunctive normal form (Tseitin 1968), *plasp* further associates disjunctions and existential quantifiers occurring in its simplified format with *derived predicates*, available from PDDL 2.2 on (Edelkamp and Hoffmann 2004). Derived predicates are similar to defined fluents used in action languages $\mathcal{AL}_d$ (Gelfond and Inclezan 2013) or $\mathcal{C}+$ (Giunchiglia et al. 2004) and, unlike fluents, they are not subject to inertia, but rules for deriving them are evaluated under well-founded semantics (Van Gelder et al. 1991) in each state. The prerequisites of respective rule instances reflect the elements of a disjunction or substitutions for existentially quantified variables, respectively. As any dependency between derived predicates introduced by *plasp* matches an occurrence of one expression in another, such dependencies are inherently noncircular and yield a total well-founded model in each state. The achievement of representing disjunctions and existential quantifiers by derived predicates is that preconditions and goals (and likewise postconditions) can be uniformly regarded as partial states over fluents as well as derived predicates, while dedicated treatment of more complex expressions were needed otherwise.

In the final step of its translation, *plasp* outputs a normalized PDDL specification in terms of ASP facts. This includes facts specifying fluents as well as derived predicates along with their possible values, namely, `true` and `false`. Moreover, actions are described by facts providing their preconditions and postconditions, where a postcondition may in turn be subject to a condition in order to encompass conditional effects. Similar facts are used to express the preconditions of rules for concluding the truth of derived predicates,

---

[4] https://github.com/potassco/plasp
[5] PDDL 3.1 further allows for numeric fluents, action costs, durative actions, preferences, and trajectory constraints, which are not yet supported by the current version of *plasp*.

where an argument specifies whether the elements of a precondition contribute to a conjunction or disjunction, respectively. Notably, the falsity of a derived predicate is not addressed explicitly by rules, but rather follows "automatically" whenever all rules for the predicate are inapplicable. Facts giving the values of fluents in state s_0 as well as the goal conditions in $s_\star$ then complete the factual representation of a planning task. The detailed reference documentation of the fact format obtained by translation with *plasp* can be found online.[6]

As an alternative to the direct translation of PDDL specifications, *plasp* supports the intermediate SAS format (Helmert 2006), obtained by preprocessing a PDDL input with the planning system *Fast Downward* (thus following the lower branch in the workflow displayed in Figure 1). On the one hand, the SAS format constitutes a modest extension to propositional STRIPS, so that its translation to ASP facts is rather straightforward and does not involve any sophisticated normalization step. In fact, grounding and simplification of PDDL specifications are delegated to *Fast Downward* in this workflow, which takes care of reducing complex expressions to the core constructs comprised in SAS. On the other hand, SAS brings about some particularities that are worth mentioning and are thus addressed below.

Most notably, the SAS format features (proper) *multivalued fluents*, and *Fast Downward* includes means to infer such fluents from PDDL inputs. For instance, given a blocks world instance with n blocks and Boolean fluents such as $on(1,0), on(1,2), \ldots, on(1,n)$ in a PDDL specification (where $1, \ldots, n$ stand for blocks and 0 for the table), *Fast Downward* may introduce a single multivalued fluent $on(1)$ with the domain $on(1)^d = \{0, 2, \ldots, n\}$ for block 1, as well as a corresponding fluent for each other block. The introduction of multivalued fluents may thus reduce the overall number of fluents and lead to a much more compact propositional representation than obtained when grounding a PDDL specification in a naive fashion. Beyond that, a multivalued fluent makes a functional dependency more explicit than a group of Boolean fluents among which exactly one happens to be true in each state. To see this, note that the choice rule in Line 9 of Listing 2 readily expresses that each state maps a multivalued fluent to some value in its domain, while matching (successor) states to applied actions were required to figure out that several Booleans cannot hold together or be all false in a state, respectively.

In addition to multivalued fluents, the preprocessing by *Fast Downward* may infer *mutex groups*, providing fluent values such that at most one of them can hold in a state. Reconsidering a blocks world instance with n blocks, the values $on(1) = n, \ldots, on(n-1) = n$ exclude each other, and a respective mutex group makes explicit that at most one block can be located on the block denoted by n, no matter the actions leading to a particular state. The mutex groups inferred by *Fast Downward* are reported in the SAS format and provide redundant/implied information that can nevertheless help to improve plan search, as corresponding integrity constraints on (successor) states are easy to express in ASP and readily included in the online versions[4] of the encodings given in Section 2.

Finally, the SAS format features *axiom rules* as a counterpart for rules addressing derived predicates in PDDL. Unlike the latter, however, axiom rules are grouped into layers specifying an evaluation order, rather than relying on well-founded semantics and

---

[6]https://github.com/potassco/plasp/blob/master/doc/output-format.md

stratification (Apt et al. 1987) for guaranteeing a unique outcome of the rules. In view of this difference, the current fact format[6] of *plasp* distinguishes between derived predicates according to PDDL and axiom rules specified in SAS, while PDDL and SAS inputs lead to a homogeneous factual representation otherwise. Given that the encodings in Section 2 focus on (multivalued) STRIPS, so that derived predicates and axiom rules are beyond scope, we rely on the common fact format obtained with *plasp* to compare ASP-based planning with or without preprocessing by *Fast Downward* in Section 5. However, we envisage to overcome the representation gap between derived predicates and axiom rules by furnishing a common fact format for both in future versions of *plasp*, and generalizing (parallel) ASP encodings beyond the STRIPS fragment is a subject to future work as well. Notably, a prototypical approach to encode axiom rules in ASP has been developed in (Miura and Fukunaga 2017) and suggests functionalities for automatic axiom extraction.

Apart from translating PDDL or SAS inputs to ASP facts (by using the `translate` command of *plasp*), *plasp* offers additional functionalities activated by respective commands. These include `normalize` in order to inspect a normalized PDDL specification produced by *plasp* in PDDL syntax instead of ASP facts. Moreover, `check-syntax` and `beautify` allow for verifying whether a given PDDL specification is supported by *plasp* or pretty-printing it with a uniform indentation to improve readability. Further commands will be added to future versions of *plasp* for automated support of PDDL requirements analysis and plan verification, among others.

Last but not least, we note that *plasp* is implemented in C++, pursuing a modular design geared for both efficiency and extensibility. In particular, *plasp* builds on top of a dedicated `pddl` library, which provides the parsing and normalization functionalities used in its translation. Given that such functionalities are independent of the ASP target formalism, the `pddl` library might be useful for third-party planner developers as well.

5 Experiments

To empirically contrast the different encodings and planning algorithms presented in Sections 2 and 3, we ran *plasp* on PDDL specifications[7] from the International Planning Competition. For comparison, we also include two variants of the state-of-the-art SAT planning system *Madagascar* (Rintanen 2014), where M stands for the standard version and M_p for the use of a specific planning heuristic. The experiments were performed sequentially on a Linux 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., $B^\forall$ refers to algorithm B applied to the encoding of $\forall$-step plans given by Listings 2 and 4. The parameters of A and B are set to $n = 16$ or $\gamma = 0.9$, respectively, as suggested in (Rintanen et al. 2006). With all three planning algorithms, i.e., S, A, and B, we fix the increment amount for increasing the plan length to five states rather than

---

[7]https://github.com/potassco/pddl-instances

domain	M_p	M	B_p^G	$B_p^\forall$	B_p^G	$B_p^\forall$	B^E	$B^\exists$	B^R	$B^\forall$	B^s	$A^\forall$	$S^\forall$
gripper (1998)	10/10	10/10	10/10	10/10	5/10	4/10	10/10	10/10	10/10	5/10	4/10	5/10	2/10
logistics (1998)	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3
blocks (2000)	19/19	19/19	19/19	19/19	19/19	19/19	11/19	8/19	7/19	12/19	14/19	11/19	8/19
elevator (2000)	20/20	20/20	20/20	20/20	20/20	20/20	19/20	20/20	20/20	14/20	20/20	13/20	3/20
depots (2002)	11/11	11/11	11/11	11/11	11/11	11/11	7/11	9/11	7/11	8/11	4/11	8/11	9/11
driverlog (2002)	13/13	13/13	13/13	13/13	11/13	11/13	11/13	11/13	10/13	10/13	7/13	10/13	11/13
all domains	76/76	76/76	76/76	76/76	69/76	68/76	61/76	61/76	57/76	52/76	52/76	50/76	36/76

Table 1. Solved instances without preprocessing by *Fast Downward*

domain	M_p	M	B_p^G	$B_p^\forall$	B_p^G	$B_p^\forall$	B^E	$B^\exists$	B^R	$B^\forall$	B^s	$A^\forall$	$S^\forall$
gripper (1998)	0.29	0.35	2.81	8.55	540.72	567.59	17.40	31.89	132.78	487.46	575.73	566.90	756.10
logistics (1998)	0.29	0.32	1.96	1.36	58.32	44.56	46.65	46.66	45.14	44.56	62.95	45.03	44.57
blocks (2000)	0.82	0.44	2.48	2.08	39.51	32.99	454.07	576.42	650.82	450.42	398.94	488.80	527.45
elevator (2000)	0.31	0.31	2.02	2.04	2.64	2.44	46.44	7.82	45.12	317.15	28.16	405.54	809.92
depots (2002)	0.37	0.42	5.08	6.08	106.12	103.12	337.93	281.14	486.31	259.26	632.16	283.20	241.04
driverlog (2002)	5.00	6.72	5.84	4.33	140.78	140.40	139.80	142.65	250.32	210.24	440.72	296.59	184.78
all domains	1.24	1.46	3.33	3.86	123.46	124.27	202.69	217.29	307.04	335.45	352.27	397.01	512.74

Table 2. Average runtimes without preprocessing by *Fast Downward*

allowing for a single additional state only, as this accelerated increase led to generally better performance. For example, algorithm A with $n = 16$ initially runs the lengths $0, 5, 10, \ldots, 75$ simultaneously. Each of the resulting algorithm/encoding combinations can optionally be augmented with the planning heuristic described in Section 3, which is denoted by an additional subscript $_p$, like in $B_p^\forall$. Moreover, the superscript ^{G} stands for the guess-and-check strategy that switches from the program G in Section 3 to the encoding of $\forall$-step plans in case a sequence of sets of actions obtained with G turns out to be not $\exists$-step serializable, and we below investigate the setting B_p^G that emerged as the overall most successful combination of techniques for our benchmarks.

Tables 1 and 2 show the numbers of solved instances and average runtimes, for individual domains of PDDL specifications and in total, for the two aforementioned *Madagascar* variants and *plasp* settings that do not use preprocessing by *Fast Downward*. However, for comparison we also include the two *plasp* configurations indicated in blue, which are based on preprocessing by *Fast Downward* and obtain their ASP facts from SAS format. First, comparing different planning algorithms, we observe that the sequential approach of $S^\forall$ falls significantly behind the other strategies that consider several plan lengths simultaneously. Unlike that, the gap between $A^\forall$ and $B^\forall$, where the latter serves as our baseline for varying the planning algorithm, encoding, or heuristic, amounts to two more solved instances only, showing that both algorithms likewise help to overcome costly unsatisfiable parts of the plan search. Regarding different encodings, aiming at sequential plans with B^s works well in the inherently sequential *blocks (2000)* domain as well as in the *elevator (2000)* domain, although parallel representations manage to reduce plan length here. In the *depots (2002)* and *driverlog (2002)* domains, however, the performance of B^s does not match parallel representations, so that it ends up last among the encodings run with planning algorithm B. The next better *plasp* setting, $B^\forall$, improves in the latter domains by referring to $\forall$-step plans, while it also exhibits particular difficulties in the *elevator (2000)* domain and does thus not solve more instances in total than B^s. The encodings of (relaxed) $\exists$-step plans, utilized by $B^\exists$, B^E, and B^R, have

domain	M_p	M	B_p^G	$B_p^\forall$	$B^\forall$	$A^\forall$	B^E	$B^\exists$	$S^\forall$	B^R	B^s	B_p^G	$B_p^\forall$
grid (1998)	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	0/2	0/2
gripper (1998)	17/17	17/17	17/17	17/17	17/17	15/17	17/17	17/17	1/17	17/17	4/17	4/17	3/17
logistics (1998)	20/20	20/20	20/20	20/20	20/20	20/20	20/20	20/20	19/20	19/20	7/20	0/20	0/20
mystery (1998)	14/14	14/14	14/14	14/14	14/14	14/14	10/14	10/14	14/14	9/14	14/14	0/14	0/14
blocks (2000)	20/20	20/20	20/20	20/20	20/20	20/20	20/20	20/20	20/20	20/20	20/20	15/20	14/20
elevator (2000)	20/20	20/20	20/20	20/20	20/20	20/20	20/20	20/20	19/20	20/20	20/20	20/20	20/20
freecell (2000)	16/16	16/16	16/16	16/16	16/16	16/16	5/16	5/16	16/16	5/16	16/16	0/16	0/16
depots (2002)	18/18	18/18	18/18	18/18	15/18	15/18	18/18	18/18	16/18	14/18	7/18	12/18	13/18
driverlog (2002)	9/9	9/9	9/9	9/9	9/9	9/9	9/9	9/9	8/9	7/9	5/9	7/9	7/9
all domains	136/136	136/136	136/136	136/136	133/136	131/136	121/136	121/136	115/136	113/136	95/136	58/136	57/136

Table 3. Solved instances with preprocessing by *Fast Downward*

domain	M_p	M	B_p^G	$B_p^\forall$	$B^\forall$	$A^\forall$	B^E	$B^\exists$	$S^\forall$	B^R	B^s	B_p^G	$B_p^\forall$
grid (1998)	0.97	1.69	9.53	9.05	15.29	15.18	47.39	55.11	14.64	370.15	100.52	900.00	900.00
gripper (1998)	0.31	0.61	6.00	22.30	83.39	466.75	21.42	21.78	847.34	37.04	738.88	741.50	757.34
logistics (1998)	13.42	13.61	11.75	10.67	10.59	10.63	35.62	37.38	54.35	99.97	663.05	900.00	900.00
mystery (1998)	4.38	33.96	20.45	16.69	26.55	68.71	284.00	294.57	19.49	367.34	21.38	900.00	900.00
blocks (2000)	2.21	1.40	5.59	5.21	4.81	4.82	8.43	22.66	4.79	33.33	4.05	383.18	366.12
elevator (2000)	0.32	0.31	2.69	4.82	5.12	4.35	2.95	3.05	128.51	8.94	7.61	20.82	20.65
freecell (2000)	0.88	19.15	17.55	14.27	14.53	14.54	655.81	657.26	14.47	669.84	123.03	900.00	900.00
depots (2002)	0.51	7.62	8.18	12.60	204.55	221.76	22.91	54.70	187.64	333.34	603.41	367.43	389.62
driverlog (2002)	0.50	3.44	10.02	8.33	26.80	27.73	19.77	31.89	113.14	281.05	492.90	203.55	203.21
all domains	3.05	9.34	9.75	11.58	46.96	101.45	121.02	129.81	169.76	210.33	322.22	558.32	560.68

Table 4. Average runtimes with preprocessing by *Fast Downward*

noticeable advantages and work especially well in the *gripper (1998)* domain, where they prove to be more effective than the encodings of sequential and ∀-step plans. While the different implementation techniques of $B^\exists$ and B^E yield some performance variance, yet without a clear trend in favor of either encoding, the extra efforts for enabling relaxed ∃-step plans with B^R do not pay off and are particularly counterproductive in the *blocks (2000)* domain, given that parallel encodings do not lead to reduced plan length here. The planning heuristic applied by $B_p^\forall$ as well as B_p^G constitutes an orthogonal approach to boost plan search, which turns out to be advantageous in all but the *gripper (1998)* domain in which the encodings of (relaxed) ∃-step plans remain more successful. However, as the two *plasp* settings in blue that are included for comparison show, the preprocessing by *Fast Downward* is the by far most effective way to improve plan search, and these two settings also come close to *Madagascar*, whose lead in terms of time is explained by its streamlined yet planning-specific implementation of grounding.

In the same manner as above, Tables 3 and 4 report numbers of solved instances and average runtimes for *plasp* settings that make use of preprocessing by *Fast Downward*, where the $B_p^\forall$ and B_p^G configurations given in black take PDDL inputs directly and serve for comparison. Note that the tables refer to different instances than before, as the instances to include with or without preprocessing by *Fast Downward* were selected independently such that none of the *plasp* settings considered in (Dimopoulos et al. 2017) solves an instance in less than 5 seconds, while some of these configurations finds a plan within the given resource limits.[8] The first apparent observation is that the two

---

[8] The requirement that some of the *plasp* settings available in (Dimopoulos et al. 2017) had to finish an instance along with high computational efforts in view of frequent timeouts are responsible for the

comparison configurations indicated in black are outperformed by *plasp* settings based on preprocessing by *Fast Downward* and translation from SAS format, given that the introduced multivalued fluents make ground instantiations much more compact than with Boolean fluents in ASP facts obtained directly from the original PDDL specifications. The B^s configuration that aims at sequential plans is last among the *plasp* settings using preprocessing by *Fast Downward*, as it suffers from greater plan lengths than needed with parallel encodings in the *gripper (1998)*, *logistics (1998)*, *depots (2002)*, and *driverlog (2002)* domains. The next setting, B^R, solves 18 instances more by referring to relaxed $\exists$-step plans, while it also has particular difficulties in the *mystery (1998)* and *freecell (2000)* domains, where the large number of actions goes along with an expensive ground representation of serialization conditions. Although the same bottleneck applies to the $\mathsf{B}^\exists$ and B^E configurations, utilizing different encodings of $\exists$-step plans, they manage to remain ahead of the sequential algorithm of $\mathsf{S}^\forall$, which is outperformed by the other two planning algorithms, applied by $\mathsf{A}^\forall$ and $\mathsf{B}^\forall$, in the *gripper (1998)* domain. In fact, the gap between $\mathsf{A}^\forall$ and $\mathsf{B}^\forall$ is again small, and the additional incorporation of a planning heuristic in $\mathsf{B}^\forall_p$ and B^G_p lets these two settings solve all instances under consideration. Similarly, the heuristic of M_p brings about a time advantage in comparison to the plain version M of *Madagascar*, and the time difference between M_p and the $\mathsf{B}^\forall_p$ and B^G_p configurations of *plasp* evolves primarily from grounding.

For a broad comparison of the *Madagascar* version with planning heuristic, M_p, and the best-performing *plasp* setting, B^G_p, integrating preprocessing by *Fast Downward*, planning algorithm B along with our guess-and-check strategy to delay serialization conditions, and a planning heuristic, we ran both planners on the full collection of STRIPS planning tasks used in (Rintanen 2012). The corresponding numbers of solved instances and average runtimes are shown in Table 5. In 15 out of the 37 domains, M_p and B^G_p solve the same number of instances, M_p is ahead in 15 domains, and B^G_p has an advantage in 7 domains. While the performance of both planning systems generally tends to be close, the observed differences stem from specific implementations of grounding and heuristics, as well as the use of an encoding of $\exists$-step plans by M_p, where B^G_p switches to $\forall$-step plans instead if a plan that is not $\exists$-step serializable is found. In fact, in the *elevator (2011)* domain, we checked that the gap of 18 solved instances between M_p and B^G_p is due to the (implementation of) planning heuristic, as M_p is also able to solve 16 instances more than the *Madagascar* variant M without such a heuristic. Vice versa, B^G_p is ahead of M_p by 32 solved instances in the *blocks (2000)* domain, where the planning heuristic is likely to be responsible again, given that this domain is inherently sequential so that plan length remains unaffected by encoding differences. Unlike that, the advantages of M_p in the *parking (2011)* and *tidybot (2011)* domains, amounting to 19 or 14 instances solved more than with B^G_p, cannot be explained by the planning heuristic alone, but are rather related to encodings, i.e., the $\exists$-step plans obtained with M_p lead to reduced plan lengths in comparison to the $\forall$-step plans of B^G_p. Let us stress again that efficiency considerations regarding the ground representation of interfering pairs of actions lead us to the guess-and-check strategy switching to $\forall$-step rather than $\exists$-step plans. The same bottleneck has also been noted in (Rintanen 2012), and *Madagascar* features a streamlined

---

limited numbers of domains considered in Tables 1–4. Unlike that, Table 5 below focuses on the novel guess-and-check strategy pursued by the B^G_p configuration and covers substantially more domains.

	solved instances		average runtime	
domain	M_p	B_p^G	M_p	B_p^G
grid (1998)	5/5	4/5	73.01	263.53
gripper (1998)	20/20	20/20	0.31	5.35
logistics (1998)	27/30	30/30	127.27	34.76
movie (1998)	30/30	30/30	0.32	1.67
mystery-prime (1998)	20/20	20/20	10.07	42.13
mystery (1998)	12/19	14/19	344.29	257.26
blocks (2000)	61/102	93/102	381.89	214.97
freecell (2000)	43/60	38/60	280.59	363.27
depots (2002)	22/22	22/22	0.75	11.23
driverlog (2002)	20/20	20/20	5.59	12.83
zenotravel (2002)	20/20	20/20	0.69	14.75
airport (2004)	49/50	50/50	59.87	135.41
pipesworld-no-tankage (2004)	42/50	35/50	173.70	323.59
pipesworld-tankage (2004)	27/50	17/50	488.12	648.56
promela-dining-philosophers (2004)	29/29	29/29	0.42	258.30
promela-optical-telegraph (2004)	14/14	9/14	8.50	612.00
psr-small (2004)	50/50	50/50	0.34	12.98
satellite (2004)	32/36	19/36	104.99	431.10
pathways (2006)	30/30	30/30	0.52	275.99
rovers (2006)	40/40	32/40	30.32	190.18
storage (2006)	30/30	16/30	6.13	448.71
tpp (2006)	30/30	17/30	8.14	453.66
trucks (2006)	15/30	16/30	462.98	556.72
barman (2011)	7/20	0/20	612.40	900.00
elevator (2011)	20/20	2/20	36.48	839.21
floor-tile (2011)	20/20	20/20	0.36	3.16
no-mystery (2011)	18/20	20/20	253.69	29.79
openstacks (2011)	0/20	0/20	900.00	900.00
parc-printer (2011)	20/20	20/20	0.38	3.62
parking (2011)	19/20	0/20	148.56	900.00
peg-solitaire (2011)	20/20	15/20	18.45	268.93
scanalyzer-3d (2011)	17/20	17/20	150.95	175.08
sokoban (2011)	3/20	2/20	836.80	863.30
tidybot (2011)	17/20	3/20	176.94	822.16
transport (2011)	4/20	6/20	730.82	792.01
visit-all (2011)	0/20	0/20	900.00	900.00
woodworking (2011)	20/20	20/20	4.92	26.31
all domains	853/1067	756/1067	204.17	326.62

Table 5. Solved instances and average runtimes on the benchmark set by (Rintanen 2012)

planning-specific implementation of grounding to tame the complexity of instantiating an encoding of ∃-step plans. While *plasp* cannot compete with *Madagascar* regarding low-level efficiency, its high-level approach brings the advantage that first-order ASP encodings and general grounding facilities make it easier to prototype and experiment with different planning algorithms and encodings. Such flexibility has, e.g., been exploited in (Miura and Fukunaga 2017) to implement axiom-enhanced planning, and in (Thielscher 2009) to solve single-player games specified in the Game Description Language (GDL; (Love et al. 2008)) by means of ASP planning.

In addition to the above experiments on PDDL domains, we evaluated the planning algorithms S, A, and B on ASP planning benchmarks, including the *hanoi-tower*, *labyrinth*, *no-mystery*, *ricochet-robots*, *sokoban*, and *visit-all* domains from recent ASP competition editions (Alviano et al. 2013; Calimeri et al. 2016; Gebser et al. 2017b). While the

domain	A_1	A_2	A_4	A_8	A_{16}	$B_{0.5}$	$B_{0.75}$	$B_{0.875}$	$B_{0.9}$	$B_{0.9375}$
hanoi-tower	15/20	12/20	12/20	12/20	12/20	13/20	17/20	16/20	17/20	16/20
labyrinth	6/20	11/20	13/20	11/20	11/20	10/20	8/20	11/20	11/20	11/20
no-mystery	14/20	14/20	14/20	11/20	12/20	14/20	12/20	13/20	9/20	10/20
ricochet-robots	9/20	10/20	12/20	11/20	9/20	11/20	12/20	9/20	10/20	10/20
sokoban	4/20	4/20	6/20	9/20	10/20	9/20	9/20	9/20	8/20	8/20
visit-all	1/20	6/20	8/20	8/20	8/20	7/20	8/20	8/20	8/20	8/20
all domains	49/120	57/120	65/120	62/120	62/120	64/120	66/120	66/120	63/120	63/120

Table 6. Solved instances on ASP planning benchmarks

domain	A_1	A_2	A_4	A_8	A_{16}	$B_{0.5}$	$B_{0.75}$	$B_{0.875}$	$B_{0.9}$	$B_{0.9375}$
hanoi-tower	314.41	376.18	397.74	409.12	488.05	384.41	277.92	311.14	296.30	308.41
labyrinth	633.08	475.39	375.81	422.48	424.33	466.45	546.50	422.43	418.29	420.42
no-mystery	310.71	340.11	384.71	477.92	523.36	334.81	413.55	499.99	561.19	554.93
ricochet-robots	525.16	512.58	477.50	512.31	591.89	476.86	432.70	604.89	623.16	558.48
sokoban	720.48	720.22	679.87	523.46	476.59	587.15	525.97	517.96	563.00	560.44
visit-all	881.89	808.13	737.42	625.79	608.43	723.75	646.64	588.56	598.90	591.03
all domains	564.29	538.77	508.84	495.18	518.78	495.57	473.88	490.83	510.14	498.95

Table 7. Average runtimes on ASP planning benchmarks

original competition benchmarks utilize non-incremental ASP encodings along with a given maximum plan length, we furnished incremental versions of these encodings and let the planner, as presented in Section 3, look for a sufficient plan length. We varied the parameters n of A and γ of B by using 1, 2, 4, 8, and 16 for n as well as 0.5, 0.75, 0.875, 0.9, and 0.9375 for γ, which are the same values as taken in (Rintanen et al. 2006). The resulting algorithm variants are denoted by subscripts, viz. A_n for $n \in \{1, 2, 4, 8, 16\}$ and B_γ for $\gamma \in \{0.5, 0.75, 0.875, 0.9, 0.9375\}$, where A_1 represents the sequential algorithm S. The experiments on ASP planning benchmarks were performed sequentially on a Linux machine equipped with Intel Core i7-6700 processor at 3.4 GHz and 32 GB RAM, as before limiting time and memory per run to 900 seconds and 8 GB, while taking aborted runs as 900 seconds within average runtimes.

The results are summarized in Tables 6 and 7, showing the numbers of solved instances and average runtimes. The overall best-performing configuration is $B_{0.75}$, but others like $B_{0.875}$, A_4, and $B_{0.5}$ come very close, and no setting strictly dominates over all domains. However, it is apparent that the sequential approach of A_1 or S, respectively, leads to significantly fewer solved instances than the other algorithms that consider several plan lengths simultaneously, which particularly applies to the *labyrinth*, *sokoban*, and *visit-all* domains. Comparing the length of plans found between the best-performing configuration, $B_{0.75}$, and A_1 yields that the former include about 35 states on average, while A_1 leads to slightly more than 20 states only. That is, moving on to greater plan lengths before finishing costly unsatisfiable parts of the plan search is helpful on ASP planning benchmarks as well, especially in order to solve instances requiring a substantial number of states to become satisfiable. Although we do not show the detailed results here, let us note that successively incrementing the plan length by five states at once, rather than adding a single state only, likewise leads to better performance on inputs obtained by translation from PDDL and direct ASP encodings of planning problems. Given that similar solving strategies turn out to be advantageous in both cases, developing metaencoding approaches that are applicable to incremental ASP encodings, in order to

condense stable models in the same fashion as parallel representations do for sequential plans, may be a promising way to further speed up multishot ASP solving in the future.

6 Conclusion

We presented the key features of the new *plasp* 3 system, providing a translator from PDDL specifications to ASP facts along with a multishot ASP planner based on *clingo*. While the ASP metaencodings in Section 2 focus on STRIPS-like planning tasks, *plasp*'s translator component also supports a range of advanced features from PDDL 3.1 as well as the intermediate SAS format. Moreover, its planner can be applied to any incremental ASP encoding and thus to dynamic domains at large. As the experiments in Section 5 show, the resulting general-purpose approach comes close in performance to the state-of-the-art SAT planning system *Madagascar*, where differences are mainly due to the specific implementations of grounding and heuristics. In particular, general grounding techniques can constitute a bottleneck on PDDL domains involving a large number of actions or fluents, while dedicated preprocessing as provided by *Fast Downward* helps to make the propositional representation of planning tasks more compact. The major benefit of the high-level approach of *plasp* is that first-order ASP encodings and general grounding means facilitate prototyping and experimenting with different planning algorithms and encodings. As future work, we intend to generalize our ASP encodings of parallel plans beyond the STRIPS fragment of PDDL, in order to support the advanced PDDL and SAS features mentioned in Section 4, e.g., conditional effects and derived predicates or axiom rules, respectively. Whether parallel representations of plans can be further adopted to express the stable models of arbitrary incremental ASP encodings more compactly is also an interesting open question that may be addressed in the future.

References

ALVIANO, M., CALIMERI, F., CHARWAT, G., DAO-TRAN, M., DODARO, C., IANNI, G., KRENNWALLNER, T., KRONEGGER, M., OETSCH, J., PFANDLER, A., PÜHRER, J., REDL, C., RICCA, F., SCHNEIDER, P., SCHWENGERER, M., SPENDIER, L., WALLNER, J., AND XIAO, G. 2013. The fourth answer set programming competition: Preliminary report. In *Proceedings of the Twelfth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'13)*, P. Cabalar and T. Son, Eds. Lecture Notes in Artificial Intelligence, vol. 8148. Springer-Verlag, 42–53.

APT, K., BLAIR, H., AND WALKER, A. 1987. Towards a theory of declarative knowledge. In *Foundations of Deductive Databases and Logic Programming*, J. Minker, Ed. Morgan Kaufmann Publishers, 89–148.

BARAL, C. AND GELFOND, M. 2000. Reasoning agents in dynamic domains. In *Logic-Based Artificial Intelligence*, J. Minker, Ed. Kluwer Academic Publishers, 257–279.

BIERE, A., HEULE, M., VAN MAAREN, H., AND WALSH, T., Eds. 2009. *Handbook of Satisfiability.* Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press.

BOMANSON, J., GEBSER, M., JANHUNEN, T., KAUFMANN, B., AND SCHAUB, T. 2016. Answer set programming modulo acyclicity. *Fundamenta Informaticae 147,* 1, 63–91.

CALIMERI, F., GEBSER, M., MARATEA, M., AND RICCA, F. 2016. Design and results of the fifth answer set programming competition. *Artificial Intelligence 231*, 151–181.

CIMATTI, A., PISTORE, M., AND TRAVERSO, P. 2008. Automated planning. See Lifschitz et al. (2008), 841–867.

DIMOPOULOS, Y., GEBSER, M., LÜHNE, P., ROMERO, J., AND SCHAUB, T. 2017. plasp 3: Towards effective ASP planning. In *Proceedings of the Fourteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'17)*, M. Balduccini and T. Janhunen, Eds. Lecture Notes in Artificial Intelligence, vol. 10377. Springer-Verlag, 286–300.

DIMOPOULOS, Y., NEBEL, B., AND KÖHLER, J. 1997. Encoding planning problems in nonmonotonic logic programs. In *Proceedings of the Fourth European Conference on Planning (ECP'97)*, S. Steel and R. Alami, Eds. Lecture Notes in Artificial Intelligence, vol. 1348. Springer-Verlag, 169–181.

EDELKAMP, S. AND HOFFMANN, J. 2004. PDDL2.2: The language for the classical part of the 4th international planning competition. Tech. Rep. 195, Institute of Informatics, University of Freiburg.

EITER, T. AND POLLERES, A. 2006. Towards automated integration of guess and check programs in answer set programming: A meta-interpreter and applications. *Theory and Practice of Logic Programming 6,* 1-2, 23–60.

FISHER, M. 2008. Temporal representation and reasoning. See Lifschitz et al. (2008), 513–550.

GEBSER, M., KAMINSKI, R., KAUFMANN, B., LINDAUER, M., OSTROWSKI, M., ROMERO, J., SCHAUB, T., AND THIELE, S. 2017a. Potassco user guide, second ed. University of Potsdam. Available at https://github.com/potassco/guide/releases/tag/v2.1.0.

GEBSER, M., KAMINSKI, R., KAUFMANN, B., OSTROWSKI, M., SCHAUB, T., AND WANKO, P. 2016. Theory solving made easy with clingo 5. In *Technical Communications of the Thirty-second International Conference on Logic Programming (ICLP'16)*, M. Carro, A. King, N. Saeedloei, and M. De Vos, Eds. Open Access Series in Informatics, vol. 52. Dagstuhl Publishing, 2:1–2:15.

GEBSER, M., KAMINSKI, R., KAUFMANN, B., AND SCHAUB, T. 2014. Clingo = ASP + control: Preliminary report. In *Technical Communications of the Thirtieth International Conference on Logic Programming (ICLP'14)*, M. Leuschel and T. Schrijvers, Eds. *Theory and Practice of Logic Programming 14,* 4-5, online supplement. Available at https://arxiv.org/abs/1405.3694v1.

GEBSER, M., KAMINSKI, R., KNECHT, M., AND SCHAUB, T. 2011a. plasp: A prototype for PDDL-based planning in ASP. In *Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*, J. Delgrande and W. Faber, Eds. Lecture Notes in Artificial Intelligence, vol. 6645. Springer-Verlag, 358–363.

GEBSER, M., KAUFMANN, B., OTERO, R., ROMERO, J., SCHAUB, T., AND WANKO, P. 2013. Domain-specific heuristics in answer set programming. In *Proceedings of the Twenty-Seventh National Conference on Artificial Intelligence (AAAI'13)*, M. desJardins and M. Littman, Eds. AAAI Press, 350–356.

GEBSER, M., MARATEA, M., AND RICCA, F. 2017b. The sixth answer set programming competition. *Journal of Artificial Intelligence Research 60*, 41–95.

GEBSER, M., SABUNCU, O., AND SCHAUB, T. 2011b. An incremental answer set programming based system for finite model computation. *AI Communications 24,* 2, 195–212.

GELFOND, M. AND INCLEZAN, D. 2013. Some properties of system descriptions of $\mathcal{AL}_d$. *Journal of Applied Non-Classical Logics 23,* 1-2, 259–285.

GELFOND, M. AND LIFSCHITZ, V. 1998. Action languages. *Electronic Transactions on Artificial Intelligence 3,* 6, 193–210.

GIUNCHIGLIA, E., LEE, J., LIFSCHITZ, V., MCCAIN, N., AND TURNER, H. 2004. Nonmonotonic causal theories. *Artificial Intelligence 153,* 1-2, 49–104.

Helmert, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research 26*, 191–246.

IPC. 2014. Homepage of the eighth international planning competition. `https://helios.hud.ac.uk/scommv/IPC-14/`.

Kautz, H., McAllester, D., and Selman, B. 1996. Encoding plans in propositional logic. In *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR'96)*, L. Aiello, J. Doyle, and S. Shapiro, Eds. Morgan Kaufmann Publishers, 374–384.

Lifschitz, V. 2002. Answer set programming and plan generation. *Artificial Intelligence 138,* 1-2, 39–54.

Lifschitz, V., van Harmelen, F., and Porter, B., Eds. 2008. *Handbook of Knowledge Representation.* Elsevier Science.

Love, N., Hinrichs, T., Haley, D., Schkufza, E., and Genesereth, M. 2008. General game playing: Game description language specification. Tech. Rep. LG-2006-01, Stanford University.

McDermott, D. 1998. PDDL — the planning domain definition language. Tech. Rep. CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

Miura, S. and Fukunaga, A. 2017. Automatic extraction of axioms for planning. In *Proceedings of the Twenty-seventh International Conference on Automated Planning and Scheduling (ICAPS'17)*, L. Barbulescu, J. Frank, Mausam, and S. Smith, Eds. AAAI Press, 218–227.

Rintanen, J. 2012. Planning as satisfiability: Heuristics. *Artificial Intelligence 193*, 45–86.

Rintanen, J. 2014. Madagascar: Scalable planning with SAT. In *Proceedings of the Eighth International Planning Competition (IPC'14)*, M. Vallati, L. Chrpa, and T. McCluskey, Eds. University of Huddersfield, 66–70.

Rintanen, J., Heljanko, K., and Niemelä, I. 2006. Planning as satisfiability: Parallel plans and algorithms for plan search. *Artificial Intelligence 170,* 12-13, 1031–1080.

Son, T., Baral, C., Nam, T., and McIlraith, S. 2006. Domain-dependent knowledge in answer set planning. *ACM Transactions on Computational Logic 7,* 4, 613–657.

Thielscher, M. 2009. Answer set programming for single-player games in general game playing. In *Proceedings of the Twenty-fifth International Conference on Logic Programming (ICLP'09)*, P. Hill and D. Warren, Eds. Lecture Notes in Computer Science, vol. 5649. Springer-Verlag, 327–341.

Tseitin, G. 1968. On the complexity of derivation in the propositional calculus. *Zapiski Nauchnykh Seminarov LOMI 8*, 234–259.

Van Gelder, A., Ross, K., and Schlipf, J. 1991. The well-founded semantics for general logic programs. *Journal of the ACM 38,* 3, 620–650.

Wehrle, M. and Rintanen, J. 2007. Planning as satisfiability with relaxed ∃-step plans. In *Proceedings of the Twentieth Australian Joint Conference on Artificial Intelligence (AI'07)*, M. Orgun and J. Thornton, Eds. Lecture Notes in Computer Science, vol. 4830. Springer-Verlag, 244–253.