Gearing up for Effective ASP Planning

Martin Gebser, Roland Kaminski, and Torsten Schaub*

Universität Potsdam

Abstract. We elaborate upon incremental modeling techniques for ASP Planning, a term coined by Vladimir Lifschitz at the end of the nineties. Taking up this line of research, we argue that ASP needs both a dedicated modeling methodology and sophisticated solving technology in view of the high practical relevance of dynamic systems in real-world applications.

1 Introduction

The stable models semantics was born more than two decades ago, fathered by Michael Gelfond and Vladimir Lifschitz in [1]. Since then, it has seen a pretty rough childhood. Initially facing the greatly dominating elder brother Prolog, it made its way despite many fights with first and second grade cousins in the area of Logic Programming and Nonmonotonic Reasoning. Being now in its early adulthood, under the pseudonym of Answer Set Programming (ASP; [2]), it entertains a competitive yet extremely fruitful relationship with Satisfiability Testing (SAT; [3]), an offspring of a house with a certain veil of antique nobility, viz. classical logic.

However, the rivalry between ASP and SAT has turned out to be extremely productive for ASP. And in fact ASP often followed in the footsteps of SAT. This is particularly true as regards computational issues, where the Davis-Putman-Logemann-Loveland procedure [4, 5] led the way for the first effective implementation of ASP, namely the *smodels* system [6]. Similarly, current ASP solvers like *clasp* [7] largely benefit from the technology of advanced Boolean constraint solving boosted to great success in the area of SAT (cf. [3]).

Looking at the success stories of SAT, among which the most shiny ones are arguably Automated Planning [8] and Model Checking [9], we notice that both deal with dynamic applications, whose complexity seems to be a priori out of reach of SAT. In both cases the key idea was to reduce the complexity from PSPACE to NP by treating these problems in a bounded way and to consider in turn one problem instance after another by gradually increasing the bound on the solution size. This idea can be seen as the major driving force behind the extension of SAT solvers with incremental interfaces (cf. [10, 11]) which constitute nowadays a key technology in many SAT-based real-world applications.

Similarly, there were early attempts to apply ASP to Automated Planning in [12, 13] based upon which Vladimir Lifschitz put forward ASP Planning in [14, 2] as a knowledge-intense alternative to SAT Planning. In fact, ASP's rich modeling language

* Affiliated with the School of Computing Science at Simon Fraser University, Canada, and the Institute for Integrated and Intelligent Systems at Griffith University, Australia.

offers an attractive alternative to the encoding of planning problems via imperative programming languages, which is common and actually unavoidable in SAT. So far, however, ASP Planning is no real match for SAT Planning. For one thing, ASP modeling techniques for dynamic domains focus on knowledge representation issues but neglect the development of design patterns aiming at search space reductions, as found in the SAT Planning literature [15] (eg. forward expansion, mutex analysis, or operator splitting). A first yet incomplete attempt to address this problem was done in [16], where a selection of such SAT Planning techniques was modeled and studied in the context of ASP. This approach is summarized in Section 6.

Another and more general reason why ASP is lagging behind SAT in terms of dynamic applications is that incremental grounding and solving techniques have not yet found the same proliferation as in SAT. On the one hand, SAT has its focus on solving, while ASP is additionally concerned with grounding in view of its modeling language. Hence, it is a more complex endeavor to come up with an ASP system addressing both incremental grounding and solving. On the other hand, now that a first incremental ASP system, viz. *iclingo* [17], is available since a couple of years, we find it important to elaborate upon the differences in modeling and employment with respect to a static setting in order to foster its usage and thus to open up dynamic applications to ASP.

This is also our essay's topic. After laying some formal foundations in Section 2, we start from a blocks world planning example stemming from Vladimir Lifschitz' work on ASP Planning and transfer it into an incremental setting in Section 3 and 4. For a complement, we address in Section 5 the "Towers of Hanoi" problem in order to deepen the introduction to modeling in incremental ASP. Finally, we return to the initial motivation of Vladimir Lifschitz' work and sketch a first prototype for PDDL-based ASP Planning.

In what follows, we presuppose some familiarity with the syntax and semantics of logic programs in the framework of ASP. A general introduction to ASP can be found in [18]; one focusing on the theme of this essay is given in [17].

2 Incremental logic programs

For capturing dynamic systems, we take advantage of *incremental logic programs* [17], consisting of triples (B, P, Q) of logic programs, among which P and Q contain a (single) parameter t ranging over the natural numbers. In view of this, we sometimes denote P and Q by $P[t]$ and $Q[t]$. The base program B is meant to describe static knowledge, independent of parameter t. The role of P is to capture knowledge accumulating with increasing t, whereas Q is specific for each value of t. Provided all programs are "modularly composable" (cf. [17]), we are interested in finding an answer set of the program

$$B \cup \left(\bigcup_{1 \leq j \leq i} P[k/j] \right) \cup Q[k/i] \tag{1}$$

for some (minimum) natural number $i \geq 1$.

Such an answer is traditionally found by appeal to iterative deepening search. That is, one first checks whether $B \cup P[1] \cup Q[1]$ has an answer set, if not, the same is done for $B \cup P[1] \cup P[2] \cup Q[2]$ and so on. For a given i, this approach re-processes B

for i times and $(i{-}j{+}1)$ times each $P[j]$, where $1 \leq j \leq i$, while each $Q[j]$ is dealt with only once. Unlike this, incremental ASP solving computes these answers sets in an incremental fashion, starting from B but then gradually dealing only with the program slices $P[i]$ and $Q[i]$ rather than the entire program in (1). However, B and the previously processed slices $P[j]$ and $Q[j]$, $1 \leq j < i$, must be taken into account when dealing with $P[i]$ and $Q[i]$: while the rules in $P[j]$ are accumulated, the ones in $Q[j]$ must be discarded. For accomplishing this, an ASP system has to operate in a "stateful way." That is, it has to maintain its previous state for processing the current program slices. In this way, all components, B, $P[j]$, and $Q[i]$, of (1) are dealt with only once, and duplicated work is avoided when increasing i.

However, it is important to note that an incremental proceeding leads to a slightly different semantics than obtained in a static setting. Foremost, we must realize that we deal with an infinite set of terms containing all natural numbers. Unlike this, an incremental proceeding aims at providing a finite grounding at each step. On the one hand, we may thus never obtain a complete finite representation of the overall program. And on the other hand, each incremental step can only produce a grounding relative to the (finite) set of terms that were up to that point encountered by the grounder. The stable models semantics must thus falsify all atoms that have so far not been derived, although they might become true at future steps. We refer the interested reader to [17] for a formal elaboration of this phenomenon along with its formal semantics.

However, given that an ASP system is composed of a grounder and a solver, an incremental ASP solver gains on both ends. As regards grounding, it reduces efforts by avoiding reproducing previous ground rules. Regarding solving, it reduces redundancy, in particular, if a learning ASP solver is used, given that previously gathered information on heuristics, conflicts, or loops (cf. [7]), respectively, remains available and can thus be continuously exploited.

For illustration, consider the following example enumerating even and odd natural numbers.

$$B = \{\ even(0)\ \}$$

$$P[k] = \left\{ \begin{array}{l} odd(k) \leftarrow even(k-1) \\ even(k) \leftarrow odd(k-1) \\ opooo(k) \leftarrow odd(k), k = M * N * O, odd(M), odd(N), odd(O) \end{array} \right\}$$

$$Q[k] = \{\ \leftarrow \{\ opooo(K) \mid K = 1..k\ \} \leq\ 2\ \}$$

The goal of this program is to find the first triple *opooo* number, that is, the smallest number k such that there are 3 odd numbers $k' \leq k$ that equal the product of 3 odd numbers.

This program is represented in the language of the incremental ASP system *iclingo* in Listing 1. The partition of rules into B, $P[k]$, and $Q[k]$ is done by appeal to the directives `#base`, `#cumulative`, and `#volatile` (all of which may appear multiple times in a program), respectively.

Passing the program in Listing 1 to *iclingo* yields an answer set at Step 27:

```
nix>  iclingo opooo.lp
Answer: 1
even(0) odd(1) even(2) odd(3) even(4) odd(5) even(6) odd(7) even(8)   \
```

Listing 1. An incremental program computing the triple *opooo* number (`opooo.lp`)

```
1  #base.

3  even(0).

5  #cumulative k.

7   odd(k) :- even(k-1).
8  even(k) :-  odd(k-1).

10 opooo(k) :- odd(k), k==M*N*O, odd(M), odd(N), odd(O), M<N, N<O.

12 #volatile k.

14 :- { opooo(K) : K=1..k } 2.
```

```
odd(9) even(10) odd(11) even(12) odd(13) even(14) odd(15) opooo(15)   \
even(16) odd(17) even(18) odd(19) even(20) odd(21) opooo(21) even(22) \
odd(23) even(24) odd(25) even(26) odd(27) opooo(27)
SATISFIABLE

Models      : 1
Total Steps : 27
Time        : 0.000
  Prepare   : 0.000
  Prepro.   : 0.000
  Solving   : 0.000
```

Observe that *iclingo* launched 26 solving processes before the above answer set was found in the 27th step. We get three opooo numbers, viz. 15, 21, and 27. Rather than re-grounding and re-solving each time from scratch, *iclingo* only grounded each time the necessary program slice. For instance, at Step 25 only 2 rules are sent to the solver:

```
odd(25).
:-.
```

The first rule is the instantiation of Line 7 in Listing 1 in view of the fact that `even(24)` was obtained at Step 24. And the second rule is the instantiation of the volatile integrity constraint in Line 14 in Listing 1. This is because the cardinality constraint is instantiated as '`{opooo(15),opooo(21)} 2`' and then evaluated to true and thus removed from the body of the integrity constraint.

3 Blocks world planning

Let us begin with addressing the problem of blocks world planning following the approach taken by Vladimir Lifschitz in [14, 2].

A planning problem consists of three parts. An initial situation, a set of actions, and a goal situation (or formula characterizing goal situations). Given such a problem description, a solution is given by a sequence of actions leading from the initial situation to a goal situation.

The initial and desired final situation of a simple blocks world problem is given in Figure 1. In each situation, we consider six blocks on a table, yet in different

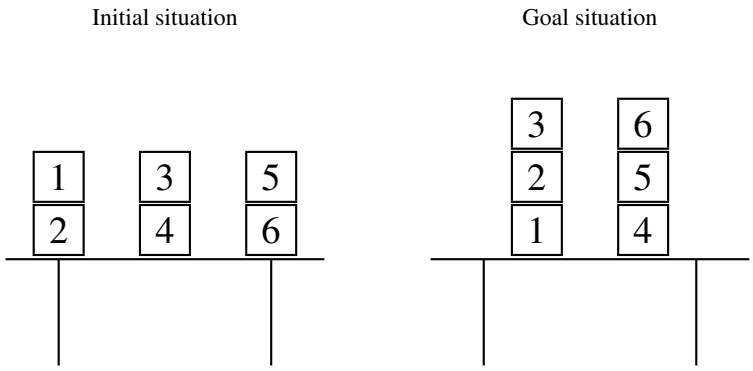

Fig. 1. The initial and goal situation for blocks world planning

Listing 2. Initial and goal situation (`blocks.lp`)

```
3  on(1,2,0).
4  on(2,table,0).
5  on(3,4,0).
6  on(4,table,0).
7  on(5,6,0).
8  on(6,table,0).


12 :- not on(3,2,lasttime).
13 :- not on(2,1,lasttime).
14 :- not on(1,table,lasttime).
15 :- not on(6,5,lasttime).
16 :- not on(5,4,lasttime).
17 :- not on(4,table,lasttime).
```

arrangements. Listing 2 gives the representation provided in [14, 2]. An atom like `on(2,table,0)` expresses that Block `2` is on the table at timepoint `0`. Note that the initial and goal situation are represented differently. While the initial situation is given and thus represented as facts, the goal situation is represented as conditions on the final state (at `lasttime`) that must be generated by a successful plan. In this simple example, we consider a single action, `move`, that allows us to move a block on a location at a certain timepoint. A location can be a block or the table.

Listing 3 gives the encoding of blocks world planning given by Vladimir Lifschitz in [14, 2], yet adapted to the current language of the ASP grounder *gringo*. For this purpose, we eliminated (nowadays) obsolete domain predicates, added the symbol `#` in front of directives, and finally provide a simpler display expression in terms of `#hide` and `#show` directives. (The formatting of Listing 2 and 3 is done in accord with the incremental ones in Listing 4 and 5, respectively.)

Listing 3. Blocks world planning: Vladimir Lifschitz' encoding (`planning.lp`)

```
1  #const lasttime=3.
2  #const grippers=2.


5  time(0..lasttime).
6  block(1..6).

8  location(B) :- block(B).
9  location(table).



13 { move(B,L,T) : block(B) : location(L) } grippers :- time(T), T<lasttime.

15 on(B,L,T+1) :- move(B,L,T), T<lasttime.
16 on(B,L,T+1) :- on(B,L,T), not onp(B,L,T+1), T<lasttime.

18 onp(B,L1,T) :- on(B,L,T), L!=L1, location(L1).

20 :- on(B,L,T), onp(B,L,T).
21 :- 2 { on(B1,B,T) : block(B1) }, block(B), time(T).
22 :- move(B,L ,T), on(B1,B,T),   T<lasttime.
23 :- move(B,B1,T), move(B1,L,T), T<lasttime.

25 #hide.
26 #show move/3.
```

Vladimir Lifschitz' encoding consists of five parts. Line 1 and 2 are directives fixing default values for the last time step as well as the number of grippers. The very first part of the actual program begins in Line 5 and ends in Line 9 and provides the basic *data*. The second part is concentrated in Line 13 and deals with the *generation* of all possible sequences of move actions. The third part furnishes the *definition* of the successor states in terms of the fluents *on* and its negation *onp*.[1] Line 15 specifies the effect of moving a block. Line 16 is a frame axiom for fluent *on*. And Line 18 addresses the uniqueness of locations, stating that once a block is on a location it cannot be on any other location. The following integrity constraints provide a *test* series, eliminating invalid solution candidates. Line 20 makes sure that fluent *onp* is the negation of *on*. Line 21 ensures that two blocks cannot be on top of the same block. Line 22 makes sure that a block cannot be moved unless it is clear. And finally Line 23 forbids that a block is moved onto a block that is also being moved. Line 25 and 26 can be regarded as the *display* part, directing the solver to project answer sets onto the instances of *move*/3.

Although there are no answer sets for `lasttime=1,2`, that is, no plans of length one or two, we get a plan of length three as shown next.

```
nix>  clingo -c lasttime=3 planning.lp blocks.lp
Answer: 1
move(6,5,2) move(3,2,2)     move(5,4,1)     \
move(2,1,1) move(3,table,0) move(1,table,0)
SATISFIABLE

Models      : 1
Time        : 0.000
  Prepare   : 0.000
```

[1] *onp* stands for *on′*.

Listing 4. Initial and goal situation: Incremental encoding (`blocksInc.lp`)

```
1  #base.

3  on(1,2,0).
4  on(2,table,0).
5  on(3,4,0).
6  on(4,table,0).
7  on(5,6,0).
8  on(6,table,0).

10 #volatile lasttime.

12 :- not on(3,2,lasttime).
13 :- not on(2,1,lasttime).
14 :- not on(1,table,lasttime).
15 :- not on(6,5,lasttime).
16 :- not on(5,4,lasttime).
17 :- not on(4,table,lasttime).
```

```
Prepro.   : 0.000
Solving   : 0.000
```

Looking at the statistics we note that 2492 rules were grounded for `lasttime=3`. Adding the 687 and 1577 ground rules obtained for the two unsatisfiable programs obtained for `lasttime=1,2`, respectively, we needed to ground in total 4731 rules in order to find the above plan. In addition, we had to re-launch *clingo* three times and no (learned) information could be passed from one attempt to the next.

4 Incremental blocks world planning

Let us now turn to an incremental setting. To begin with, let us adapt the logic program giving the initial and goal situation in Listing 2. The result is shown in Listing 4. We note that the actual program is unaffected. The only change concerns the addition of two directives. The first one, `#base`, declares the initial situation (in Line 3-8) as static information that is grounded only once and stays within the solver. In contrast to this, the statement `#volatile` directs the grounder to reground the integrity constraints (in Line 12-17) expressing the goal situation at each step, while withdrawing the previous instantiation of the constraints from the solver.

Listing 5 provides an incremental version of the encoding in Listing 3. The `#base` part is almost identical to the one in Listing 3. In fact, Line 6, 8, and 9 are identical in both listings. However, the unary `time`/1 predicate (along with the declaration of the default value of the constant `lasttime`) in Line 5 (and 1) have vanished in Listing 3. This actually applies to all occurrences of the predicate `time`/1 in Listing 3 because the instantiation of time steps is now handled via the incremental parameters and the corresponding directives.

The adaption of the remaining non-incremental encoding in Listing 3 is less straightforward. In fact, looking at the idealized program in (1), we observe that the unfolding of the cumulative program part starts with 1. The idea is that static knowledge, often attached with time step 0, belongs to the static case (that is, the `#base`

Listing 5. Blocks world planning: Incremental encoding (`planningInc.lp`)

```
2  #const grippers=2.

4  #base.

6  block(1..6).

8  location(B) :- block(B).
9  location(table).

11 #cumulative t.

13 { move(B,L,t-1) : block(B) : location(L) } grippers.

15 on(B,L,t) :- move(B,L,t-1).
16 on(B,L,t) :- on(B,L,t-1), not onp(B,L,t).

18 onp(B,L1,t) :- on(B,L,t), L!=L1, location(L1).

20 :- on(B,L,t), onp(B,L,t).
21 :- 2 { on(B1,B,t) : block(B1) }, block(B).
22 :- move(B,L ,t-1), on(B1,B,t-1).
23 :- move(B,B1,t-1), move(B1,L,t-1).

25 #hide.
26 #show move/3.
```

part). This semantics is also accounted for in the incremental solver *iclingo*, where the parameters like `t` declared by '`#cumulative t.`' or '`#volatile t.`' are instantiated beginning with 1. Unlike this, the time steps in the original encoding — bound by `time(T)` — range from `0` to `lasttime` (yet often limited to `T<lasttime`) in Listing 3. This difference has major consequences. First of all, the facts of the initial situation are implicitly verified by the constraints in Listing 3, while this is not the case in Listing 5. To do so, the corresponding constraints had to be replicated with time stamp `0`. Moreover, the respective time stamps have to be adapted to the shift by one. This can be accomplished as follows. For each rule in Listing 3 having body literal '`T<lasttime`', decrement the terms including `T` by one and substitute `T` by `t`. Otherwise, simply replace `T` by `t`. As a consequence, the application of `move`/3 actions is aligned in both encodings, although the generation in Line 13 refers to different relative time steps, viz `T` and `t-1`.

As a side-effect, our proceeding has also resolved a major problem that had been obtained by a straightforward replacement of `T` by `t` in Listing 3. Recall that an incremental solver unfolds the cumulative part stepwisely. Now inspecting Rule 16 in Listing 3, we observe that the literal `onp(B,L,T+1)` refers to time stamp `T+1`. The only rule deriving instances of `onp`/3 is given in Line 18 of Listing 3. However, this rule's head atom `onp(B,L1,T)` refers to time stamp `T`. Hence, when the nth program slice is grounded, the overall program may only contain instances of `onp(B,L1,T)` for `T` $= 1..n$ and none for `T` $= n+1$. As a consequence, all instances of the body literal `onp(B,L,T+1)` would be false when producing the nth program slice, simply because they refer to the yet unavailable future. Such phenomena cannot arise in a non-incremental setting given that all program slices are grounded at once. See [17] for a

formal elaboration of this and a module based account of incremental grounding and solving.

Finally, launching *iclingo* on the incremental programs in Listing 4 and 5, yields the following result. Note that we take advantage of *iclingo*'s option *--istats* in order to get some insight into the intermediate steps as well.

```
nix>  iclingo planningInc.lp blocksInc.lp --istats
=============== step 1 ===============

Models   : 0
Time     : 0.000 (g: 0.000, p: 0.000, s: 0.000)
Rules    : 656
Choices  : 0
Conflicts: 0
=============== step 2 ===============

Models   : 0
Time     : 0.000 (g: 0.000, p: 0.000, s: 0.000)
Rules    : 904
Choices  : 0
Conflicts: 0
=============== step 3 ===============
Answer: 1
move(3,table,0) move(1,table,0) move(5,4,1) \
move(2,1,1)     move(6,5,2)     move(3,2,2)

Models   : 1
Time     : 0.000 (g: 0.000, p: 0.000, s: 0.000)
Rules    : 904
Choices  : 7
Conflicts: 5
=============== Summary ===============
SATISFIABLE

Models      : 1+
Total Steps : 3
Time        : 0.000
  Prepare   : 0.000
  Prepro.   : 0.000
  Solving   : 0.000
```

In total, *iclingo* grounds 2464 rules, 656 in the first step and 904 in the second and third step. Also, the solver is initiated only once and updated twice with new information. Whenever a solving process is engaged it thus benefits from the information gathered during the previous solving attempt.

5 Towers of Hanoi

For further illustration, we now discuss a rather compact encoding of a related planning problem that was trimmed to produce a linear number of ground rules (during the 2011 ASP competition).

The towers of Hanoi problem is a simple puzzle game very similar to the blocks world planning problem presented above. The differences are that instead of a table there are pegs on which discs are put rather than blocks. In addition, discs are of different sizes and can only be put on either a peg or a disk of larger size. Given an initial placement of discs on pegs, the goal is to find a plan that establishes another such placement.

Listing 6. Towers of Hanoi instance

```
1  peg(a;b;c).

3  disk(1..6).

5  on(1,a,0).
6  on(2,b,0).
7  on(3..6,c,0).

9  goal_on(3;4,a).
10 goal_on(6,b).
11 goal_on(1;2;5,c).
```

A towers of Hanoi instance consists of a set of pegs given by predicate `peg/1`, a set of discs given by predicate `disk/1`, the initial situation specifying which disk is on which peg at time step zero via predicate `on/3`, and finally the predicate `goal/2` specifying which disk has to be on which peg in the goal situation. Note that in contrast to the description of blocks world instances, we are just specifying on which peg a disk is because the ordering of discs on a peg is implicitly given by the discs' sizes. Given that there are typically only three pegs and a much larger number of discs, this allows for representing a state with a linear number of fluents instead of the quadratic representation chosen in the blocks world setting. Listing 6 gives an instance with six discs and three pegs, which is depicted in Figure 2.

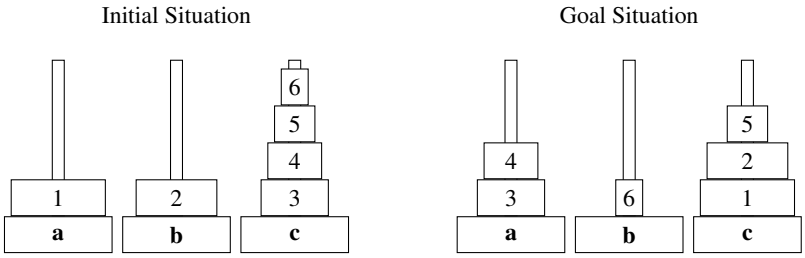

Fig. 2. The initial and goal situation for the towers of Hanoi problem

Listing 7 shows the towers of Hanoi encoding. It incorporates the same ideas as the encoding used in the last ASP competition. Hence, it is optimized for use with ASP solvers. The rule in Line 3 of the cumulative part of the encoding guesses a move. As in the instance, we just select the target peg. This way we reduce the number of atoms used to represent moves and keep the branching factor low. Note that we do not allow for parallel moves here. In principal this would easily be possible but we just have instances with three pegs. Hence, parallel moves are impossible anyway. In the consecutive line, the target is projected out to just capture which disc has been moved. The idea here is to avoid a general frame axiom as in the blocks world encoding and rather to use the moves directly to specify the state transition in lines 6 and 7. This way the grounding is kept more compact because we do not write rules involving the cross-

Listing 7. Towers of Hanoi encoding

```
1  #cumulative t.

3  1 { move(D,P,t) : disk(D) : peg(P) } 1.
4  move(D,t) :- move(D,_,t).

6  on(D,P,t) :- on(D,P,t-1), not move(D,t).
7  on(D,P,t) :- move(D,P,t).

9  blocked(D-1,P,t) :- on(D,P,t-1), D > 0.
10 blocked(D-1,P,t) :- blocked(D,P,t), D > 0.

12 :- move(D,t), on(D,P,t-1), blocked(D,P,t).
13 :- move(D,P,t), blocked(D-1,P,t).
14 :- not 1 { on(D,P,t) : peg(P) } 1, disk(D).

16 #volatile t.
17 :- goal_on(D,P), not on(D,P,t).

19 #hide.
20 #show move/3.
```

product of all discs. The number of ground rules per step here are directly proportional to number of discs (assuming a constant number of pegs). The rules in lines 9 and 10 specify which positions on a peg are blocked. Again the number of ground rules is directly proportional to the number of discs. The last block of rules in the cumulative part eliminates incorrect moves and adds some domain knowledge in Line 14 to speed up solving. The number of resulting ground rules is again proportional to the number of discs. Finally, the goal situation is checked in the volatile block in Line 17 and the answer set is projected onto the moves in lines 19 and 20.

All in all, the number of rules per time step is directly proportional to the number of discs. Thus we get a very compact encoding that can be used to solve instances requiring large plan lengths. For example a plan for the instance given in Figure 2, which requires 34 moves, can be found in less than a second with *iclingo*.

6 Towards PDDL-based ASP planning

Finally, let us sketch how incremental solving can be used for PDDL-based ASP planning. The prototypical system, *plasp*, follows the approach of *SATPlan* [8, 19] in translating a planning problem from the Planning Domain Definition Language (PDDL; [20]) into Boolean constraints. Unlike *SATPlan*, however, *plasp* aims at keeping the actual compilation simple in favor of modeling planning techniques by meta-programming in ASP. Although the compilations and meta-programs made available by *plasp* do not (yet) match the sophisticated approaches of dedicated planning systems, they allow for applying ASP systems to available planning problems. In analogy to the previous sections, *plasp* also makes use of the incremental ASP system *iclingo* [17], supporting the step-wise unrolling of problem horizons.

As illustrated in Figure 3, *plasp* translates a PDDL problem instance to ASP and runs it through a solver producing answer sets. The latter represent solutions to the initial planning problem. To this end, a plan is extracted from an answer set and output

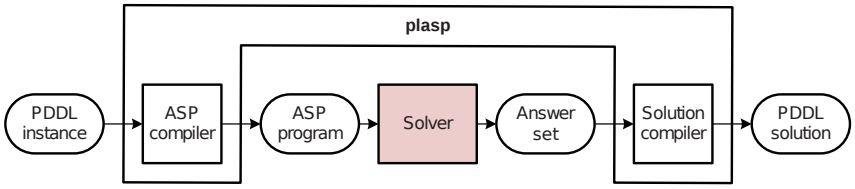

Fig. 3. Architecture of the *plasp* system

in PDDL syntax. *plasp* thus consists of two modules, viz., the ASP and Solution compilers. The *ASP compiler* is illustrated in Figure 4. First, a parser reads the PDDL de-

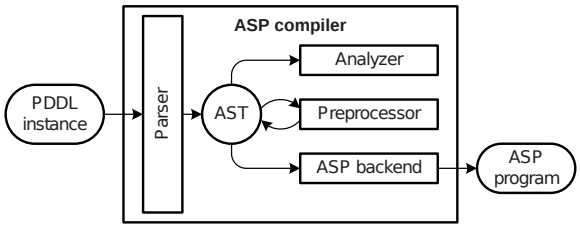

Fig. 4. Architecture of the *ASP compiler*.

scription as input and builds an internal representation, also known as Abstract Syntax Tree (AST). Then, the *Analyzer* gathers information on the particular problem instance. For example, it determines predicates representing fluents. Afterwards, the *Preprocessor* modifies the instance and enhances it for the translation process. Finally, the *ASP backend* produces an ASP program using the data gathered before. The *Solution compiler* constructs a plan from an answer set output by the solver. This is usually just a syntactic matter, but it becomes more involved in the case of parallel planning where an order among the actions must be re-established. Afterwards, the plan is verified and output in PDDL syntax.

In order to give an idea of the resulting ASP programs, let us sketch the most basic planning encoding relying on meta-programming. To this end, a PDDL domain description is mapped onto a set of facts built from predicates $init$, $goal$, $action$, $demands$, $adds$, and $deletes$ along with their obvious meanings. Such facts are then combined with the meta-program in Figure 5. Note that this meta-program is treated incrementally by the ASP system *iclingo*, as indicated in lines (1), (3), and (10). While the facts resulting from the initial PDDL description along with the ground rules of (2) in Figure 5 are processed just once (and passed to the ASP solver), the rules in (4)–(9) are successively grounded for increasing values of t and accumulated in *iclingo*'s solving component. Finally, goal conditions are expressed by volatile rules, contributing ground rules of (11) and (12) only for the current step t. From a representational perspective, it is interesting to observe that ASP allows for omitting a frame axiom (like the one in line (9)) for negative information, making use of the fact that instances of $holds$ are

```
(1)   #base.
(2)   holds(F,0) ← init(F).
(3)   #cumulative t.
(4)   1 {apply(A,t) : action(A)} 1.
(5)   ← apply(A,t), demands(A,F,true), not holds(F,t−1).
(6)   ← apply(A,t), demands(A,F,false), holds(F,t−1).
(7)   holds(F,t) ← apply(A,t), adds(A,F).
(8)   del(F,t) ← apply(A,t), deletes(A,F).
(9)   holds(F,t) ← holds(F,t−1), not del(F,t).
(10)  #volatile t.
(11)  ← goal(F,true), not holds(F,t).
(12)  ← goal(F,false), holds(F,t).
```

Fig. 5. Basic ASP encoding of STRIPS planning.

```
(4′)   1 {apply(A,t) : action(A)}.
(4′a)  ← apply(A1,t), apply(A2,t), A1 ≠ A2, demands(A1,F,true), deletes(A2,F).
(4′b)  ← apply(A1,t), apply(A2,t), A1 ≠ A2, demands(A1,F,false), adds(A2,F).
(4′c)  ← apply(A1,t), apply(A2,t), A1 ≠ A2, adds(A1,F), deletes(A2,F).
```

Fig. 6. Adaptation of the basic ASP encoding to parallel STRIPS planning.

false by default, that is, unless they are explicitly derived to be true. Otherwise, the specification follows closely the semantics of STRIPS [21].

Beyond the meta-program in Figure 5, *plasp* offers planning with concurrent actions. The corresponding modification of the rule in (4) is shown in Figure 6. While $(4')$ drops the uniqueness condition on applied actions, the additional integrity constraints stipulate that concurrent actions must not undo their preconditions, nor have conflicting effects. The resulting meta-program complies with the $\forall$-step semantics in [22]. Furthermore, *plasp* offers operator splitting as well as forward expansion. The goal of operator splitting [23] is to reduce the number of propositions in the representation of a planning problem by decomposing action predicates. For instance, an action $a(X, Y, Z)$ can be represented in terms of $a_1(X), a_2(Y), a_3(Z)$. Forward expansion (without mutex analysis [24]) instantiates schematic actions by need, viz., if their preconditions have been determined as feasible at a time step, instead of referring to statically given instances of the $action$ predicate. This can be useful if initially many instances of a schematic action are inapplicable, yet it requires a domain-specific compilation; meta-programming is difficult to apply because $action$ instances are not represented as facts. For further details on the compilation techniques supported by *plasp*, we refer the interested reader to [21, 25]. Finally, *plasp* supports combinations of forward expansion with either concurrent actions or operator splitting. Regardless of whether forward expansion is used, concurrent actions and operator splitting can currently not be combined; generally, both techniques are in opposition, although possible solutions have recently been proposed [26].

7 Discussion

ASP Planning was put forward by Vladimir Lifschitz in [14, 2] as a knowledge-intense alternative to SAT Planning. Although ASP's modeling language offers an attractive alternative to the encoding of planning problems via imperative programming languages in SAT, so far, ASP Planning is no real match for SAT Planning in terms of performance. On the one hand, ASP lacks modeling techniques aiming at search space reductions in dynamic domains. First attempts to take advantage of incremental grounding and solving ASP techniques were conducted in the areas of Automated Planning [16], Action Languages [27], Finite Model Generation [28], and Stream Reasoning [29, 30]. And on the other hand, we take too little advantage of incremental ASP solving when addressing dynamic domains. We discussed these issues and sketched first attempts to rectify this situation. However, dynamic systems are omnipresent in real-world applications. Hence, it is important to equip ASP with an adequate methodology and technology for addressing such highly demanding applications. This is not only important in offline settings, like ASP Planning, but moreover in online settings in view of the emergence of pervasive and ubiquitous computing. A first step in this direction is done in [29, 30].

Acknowledgments This work was partially funded by the German Science Foundation (DFG) under grant SCHA 550/8-1/2. We are grateful to Son Cao Tran and Oliver Ray for comments on an earlier draft of this paper.

References

1. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In Kowalski, R., Bowen, K., eds.: Proceedings of the Fifth International Conference and Symposium of Logic Programming (ICLP'88), MIT Press (1988) 1070–1080
2. Lifschitz, V.: Answer set programming and plan generation. Artificial Intelligence **138**(1-2) (2002) 39–54
3. Biere, A., Heule, M., van Maaren, H., Walsh, T., eds.: Handbook of Satisfiability. Volume 185 of Frontiers in Artificial Intelligence and Applications. IOS Press (2009)
4. Davis, M., Putnam, H.: A computing procedure for quantification theory. Journal of the ACM **7** (1960) 201–215
5. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Communications of the ACM **5** (1962) 394–397
6. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138**(1-2) (2002) 181–234
7. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In Veloso, M., ed.: Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07), AAAI Press/The MIT Press (2007) 386–392
8. Kautz, H., Selman, B.: Planning as satisfiability. In Neumann, B., ed.: Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI'92), John Wiley & sons (1992) 359–363
9. Clarke, E., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Formal Methods in System Design **19**(1) (2001) 7–34
10. Whittemore, J., Kim, J., Sakallah, K.: SATIRE: a new incremental satisfiability engine. In: Proceedings of the Thirty-eighth Conference on Design Automation (DAC'01), ACM Press (2001) 542–545

11. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electronic Notes in Theoretical Computer Science **89**(4) (2003)
12. Subrahmanian, V., Zaniolo, C.: Relating stable models and AI planning domains. In: Proceedings of the Twelfth International Conference on Logic Programming, MIT Press (1995) 233–247
13. Dimopoulos, Y., Nebel, B., Köhler, J.: Encoding planning problems in nonmonotonic logic programs. In Steel, S., Alami, R., eds.: Proceedings of the Fourth European Conference on Planning. Volume 1348 of Lecture Notes in Artificial Intelligence., Springer-Verlag (1997) 169–181
14. Lifschitz, V.: Answer set planning. In de Schreye, D., ed.: Proceedings of the International Conference on Logic Programming (ICLP'99), MIT Press (1999) 23–37
15. Rintanen, J.: Planning and SAT. [3] chapter 15 483–504
16. Gebser, M., Kaminski, R., Knecht, M., Schaub, T.: plasp: A prototype for PDDL-based planning in ASP. [31] 358–363
17. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: Engineering an incremental ASP solver. In Garcia de la Banda, M., Pontelli, E., eds.: Proceedings of the Twenty-fourth International Conference on Logic Programming (ICLP'08). Volume 5366 of Lecture Notes in Computer Science., Springer-Verlag (2008) 190–205
18. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
19. Kautz, H., Selman, B.: Pushing the envelope: Planning, propositional logic, and stochastic search. In Shrobe, H., Senator, T., eds.: Proceedings of the National Conference on Artificial Intelligence (AAAI), AAAI Press (1996) 1194–1201
20. McDermott, D.: PDDL — the planning domain definition language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control (1998)
21. Nau, D., Ghallab, M., Traverso, P.: Automated Planning: Theory and Practice. Morgan Kaufmann Publishers (2004)
22. Rintanen, J., Heljanko, K., Niemelä, I.: Parallel encodings of classical planning as satisfiability. In Alferes, J., Leite, J., eds.: Proceedings of the Ninth European Conference on Logics in Artificial Intelligence (JELIA'04). Volume 3229 of Lecture Notes in Computer Science., Springer-Verlag (2004) 307–319
23. Kautz, H., McAllester, D., Selman, B.: Encoding plans in propositional logic. In Aiello, L., Doyle, J., Shapiro, S., eds.: Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR'96), Morgan Kaufmann Publishers (1996) 374–384
24. Blum, A., Furst, M.: Fast planning through planning graph analysis. Artificial Intelligence **90**(1-2) (1997) 279–298
25. Knecht, M.: Efficient domain-independent planning using declarative programming. M.Sc. thesis, Institute for Informatics, University of Potsdam (2009)
26. Robinson, N., Gretton, C., Pham, D., Sattar, A.: SAT-based parallel planning using a split representation of actions. In Gerevini, A., Howe, A., Cesta, A., Refanidis, I., eds.: Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS'09), AAAI Press (2009) 281–288
27. Gebser, M., Grote, T., Schaub, T.: Coala: A compiler from action languages to ASP. In Janhunen, T., Niemelä, I., eds.: Proceedings of the Twelfth European Conference on Logics in Artificial Intelligence (JELIA'10). Volume 6341 of Lecture Notes in Artificial Intelligence., Springer-Verlag (2010) 360–364
28. Gebser, M., Sabuncu, O., Schaub, T.: An incremental answer set programming based system for finite model computation. AI Communications **24**(2) (2011) 195–212
29. Gebser, M., Grote, T., Kaminski, R., Schaub, T.: Reactive answer set programming. [31] 54–66

30. Gebser, M., Grote, T., Kaminski, R., Obermeier, P., O.Sabuncu, Schaub, T.: Stream reasoning with answer set programming: Preliminary report. In Eiter, T., McIlraith, S., eds.: Proceedings of the Thirteenth International Conference on Principles of Knowledge Representation and Reasoning (KR'12), AAAI Press (2012) To appear.
31. Delgrande, J., Faber, W., eds.: In Delgrande, J., Faber, W., eds.: Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11). Volume 6645 of Lecture Notes in Artificial Intelligence., Springer-Verlag (2011)
32. oclingo. http://www.cs.uni-potsdam.de/oclingo