plasp: A Prototype for PDDL-Based Planning in ASP

Martin Gebser, Roland Kaminski, Murat Knecht, and Torsten Schaub*

Institut für Informatik, Universität Potsdam

Abstract. We present a prototypical system, *plasp*, implementing Planning by compilation to Answer Set Programming (ASP). Our approach is inspired by Planning as Satisfiability, yet it aims at keeping the actual compilation simple in favor of modeling planning techniques by meta-programming in ASP. This has several advantages. First, ASP modelings are easily modifiable and can be studied in a transparent setting. Second, we can take advantage of available ASP grounders to obtain propositional representations. Third, we can harness ASP solvers providing incremental solving mechanisms. Finally, the ASP community gains access to a wide range of planning problems, and the planning community benefits from the knowledge representation and reasoning capacities of ASP.

1 Introduction

Boolean Satisfiability (SAT; [1]) checking provides a major implementation technique for Automated Planning [2]. In fact, a lot of efforts have been made to develop compilations mapping planning problems to propositional formulas. However, the underlying techniques are usually hard-wired within the compilers, so that further combinations and experiments with different features are hard to implement.

We address this situation and propose a more elaboration-tolerant platform to Planning by using Answer Set Programming (ASP; [3]) rather than SAT as target formalism. The idea is to keep the actual compilation small and model as many techniques as possible in ASP. This approach has several advantages. First, planning techniques modeled in ASP are easily modifiable and can be studied in a transparent setting. Second, we can utilize available ASP grounders to obtain propositional representations. Third, we can harness ASP solvers providing incremental solving mechanisms. Finally, the ASP community gains access to a wide range of planning problems, and the planning community benefits from the knowledge representation and reasoning capacities of ASP.

Our prototypical system, *plasp*, follows the approach of *SATPlan* [4, 5] in translating a planning problem from the Planning Domain Definition Language (PDDL; [6]) into Boolean constraints. Unlike *SATPlan*, however, we aim 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 particular, we make use of the incremental ASP system *iClingo* [7], supporting the step-wise unrolling of problem horizons. Our case studies demonstrate the impact of alternative compilations and ASP modelings on the performance of *iClingo*.

* Affiliated with Simon Fraser University, Canada, and Griffith University, Australia.

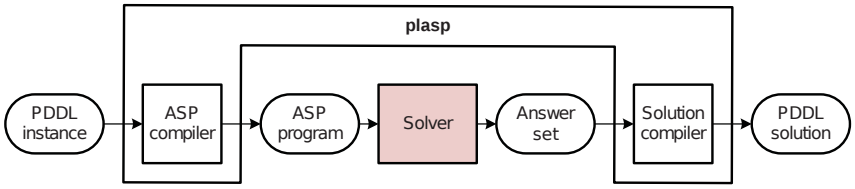

Fig. 1. Architecture of the *plasp* system.

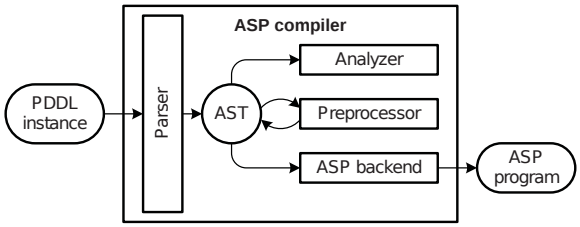

Fig. 2. Architecture of the *ASP compiler*.

2 Architecture

As illustrated in Figure 1, *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 in PDDL syntax. *plasp* thus consists of two modules, viz., the ASP and Solution compilers. The *ASP compiler* is illustrated in Figure 2. First, a parser reads the PDDL description as input and builds an internal representation, also known as Abstract Syntax Tree (AST). Then, the *Analyzer* gathers information on the particular problem instance; e.g., 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.

3 Compilations and Meta-Programs

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 3. 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) are processed just once, 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

$$
\begin{array}{ll}
(1) & \#base. \\
(2) & holds(F,0) \leftarrow init(F). \\
(3) & \#cumulative\ t. \\
(4) & 1\ \{apply(A,t) : action(A)\}\ 1. \\
(5) & \leftarrow apply(A,t), demands(A,F,true), not\ holds(F,t{-}1). \\
(6) & \leftarrow apply(A,t), demands(A,F,false), holds(F,t{-}1). \\
(7) & holds(F,t) \leftarrow apply(A,t), adds(A,F). \\
(8) & del(F,t) \leftarrow apply(A,t), deletes(A,F). \\
(9) & holds(F,t) \leftarrow holds(F,t{-}1), not\ del(F,t). \\
(10) & \#volatile\ t. \\
(11) & \leftarrow goal(F,true), not\ holds(F,t). \\
(12) & \leftarrow goal(F,false), holds(F,t).
\end{array}
$$

Fig. 3. Basic ASP encoding of STRIPS planning.

$$
\begin{array}{ll}
(4') & 1\ \{apply(A,t) : action(A)\}. \\
(4'a) & \leftarrow apply(A_1,t), apply(A_2,t), A_1 \neq A_2, demands(A_1,F,true), deletes(A_2,F). \\
(4'b) & \leftarrow apply(A_1,t), apply(A_2,t), A_1 \neq A_2, demands(A_1,F,false), adds(A_2,F). \\
(4'c) & \leftarrow apply(A_1,t), apply(A_2,t), A_1 \neq A_2, adds(A_1,F), deletes(A_2,F).
\end{array}
$$

Fig. 4. Adaptation of the basic ASP encoding to parallel STRIPS planning.

volatile rules, contributing ground rules of (11) and (12) only for the current step t. See [7] for further details on incremental ASP solving. 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 false by default, that is, unless they are explicitly derived to be true. Otherwise, the specification follows closely the semantics of STRIPS [2].

Beyond the meta-program in Figure 3, *plasp* offers planning with concurrent actions. The corresponding modification of the rule in (4) is shown in Figure 4. 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 [8]. Furthermore, *plasp* offers operator splitting as well as forward expansion. The goal of operator splitting [9] is to reduce the number of propositions in the representation of a planning problem by decomposing action predicates; e.g., 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 [10]) 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. 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 [11].

Benchmark	basic	concur	split	expand	concur+expand	split+expand	SATPlan	SGPlan
Blocks-4-0	0.16	0.21	0.43	0.21	0.22	0.20	0.34	0.10
Blocks-6-0	0.30	0.63	0.93	0.44	0.56	1.40	0.27	0.04
Blocks-8-0	1.58	6.53	12.78	98.60	317.53	47.57	1.24	0.09
Elevator-3-0	0.27	0.56	0.92	0.30	0.46	0.89	0.10	0.02
Elevator-4-0	11.72	264.11	20.30	14.69	324.18	28.88	0.30	0.02
Elevator-5-0	—	—	320.58	—	—	467.98	0.61	0.04
FreeCell-2-1	93.42	mem	64.28	60.52	51.33	56.94	2.44	0.12
FreeCell-3-1	—	mem	—	—	175.03	—	10.44	0.14
Logistics-4-0	7.85	0.38	79.15	8.81	0.39	70.56	0.34	0.05
Logistics-7-0	—	0.99	—	—	0.61	—	0.31	0.04
Logistics-9-0	—	0.89	—	—	0.57	—	0.27	0.04
Satellite-1	0.23	0.87	0.23	0.29	0.74	0.26	0.10	0.03
Satellite-2	4.56	638.08	2.19	5.43	448.60	2.69	0.41	0.03
Satellite-3	8.76	3.52	4.00	7.54	3.29	3.70	0.21	0.04
Schedule-2-0	mem	mem	mem	1.03	3.37	mem	mem	mem
Schedule-3-0	mem	mem	mem	1.63	12.89	mem	mem	mem

Table 1. Experiments comparing different compilations.

4 Experiments

We conducted experiments comparing the different compilation techniques furnished by *plasp*[1] (1.0): the meta-program in Figure 3 (column "basic" in Table 1), its adaptation to concurrent actions in Figure 4 ("concur"), operator splitting ("split"), forward expansion ("expand"), and two combinations thereof ("concur+expand" and "split+expand"). To compute answer sets of compilations, representing shortest plans, *plasp* uses (a modified version of) the incremental ASP system *iClingo*[1] (2.0.5). Although we mainly study the effect of different compilations on the performance of *iClingo*, for comparison, we also include *SATPlan*[2] (2006) and *SGPlan*[3] (5.2.2). While *SGPlan* [12] does not guarantee shortest plan lengths, the approach of *SATPlan*, based on compilation and the use of a SAT solver as search backend, leads to shortest plans. In fact, its compilation is closely related to the "concur+expand" setting of *plasp*, where *SATPlan* in addition applies mutex analysis. The benchmarks, formulated in the STRIPS subset[4] of PDDL, stem from the Second International Planning Competition[4], except for the three Satellite instances taken from the fourth competition[5]. All experiments were run on a Linux PC equipped with 2 GHz CPU and 2 GB RAM, imposing 900 seconds as time and 1.5 GB as memory limit.

Runtime results in seconds are shown in Table 1; an entry "—" indicates a timeout, and "mem" stands for memory exhaustion. On all benchmarks but Schedule, we observe that *SGPlan* has an edge on the other, less specialized (yet guaranteeing shortest

[1] `http://potassco.sourceforge.net`
[2] `http://www.cs.rochester.edu/~kautz/satplan`
[3] `http://manip.crhc.uiuc.edu/programs/SGPlan`
[4] `http://www.cs.toronto.edu/aips2000`
[5] `http://www.tzi.de/~edelkamp/ipc-4`

plans) systems. The fact that *SATPlan* is usually faster than *plasp* can be explained by the fact that compilations of *plasp* are instantiated by a general-purpose ASP grounder, while *SATPlan* utilizes a planning-specific frontend [10]. Moreover, mutex analysis as in *SATPlan* is currently not included in (encodings of) *plasp*. However, we observe that different compilation techniques of *plasp* pay off on particular benchmarks. On the Blocks and small Elevator instances, the simplest meta-program ("basic") is superior because concurrency and expansion are barely applicable to them and may even deteriorate performance. On Elevator-5-0, splitting ("split") helps to reduce the size of the problem representation. Furthermore, we observe that allowing for concurrent actions without explicit mutexes ("concur" and "concur+expand") dramatically decreases search efficiency on the Elevator domain. However, concurrent actions in combination with forward expansion ("concur+expand") are valuable on FreeCell and Logistics instances, given that they involve non-interfering actions. Splitting ("split" and "split+expand") appears to be useful on Satellite instances, where Satellite-2 again yields the phenomenon of concurrent actions deteriorating search. Finally, forward expansion ("expand") enables *plasp* to successfully deal with the Schedule domain, where even *SATPlan* and *SGPlan* exceed the memory limit. We conjecture that (too) exhaustive preprocessing, e.g., mutex analysis, could be responsible for this.

In summary, we conclude that the different compilation techniques of *plasp* can be advantageous. The automatic, domain-specific choice of an appropriate compilation, required in view of varying characteristics [12], is an intrinsic subject to future work.

5 Discussion

We have presented a prototypical approach to Automated Planning by means of compilation to ASP. In order to close the gap to established planning systems, more background knowledge (e.g., mutexes) would need to be included. If such knowledge can be encoded in meta-programs, it fosters elaboration tolerance and flexibility of planning implementations. In fact, the recent version transition of *iClingo* from 2 to 3 gives inherent support of forward expansion, generating the possibly applicable instances of actions (and fluents) on-the-fly during grounding. Importantly, regardless of additional features that might boost performance (cf. [13]), the compilation capacities of *plasp* are already useful as they make various planning problems, formulated in PDDL, accessible as benchmarks for ASP systems. The range could be further extended by generalizing the compilations supported by *plasp* beyond the STRIPS subset of PDDL.

Given the proximity of Planning and General Game Playing (GGP; [14]), the latter can also (partially) be implemented by compilation to ASP. An approach to solve single-player games in ASP is provided in [15], and [16] presents ASP-based methods to prove properties of games, which can then be exploited for playing. Automatically proving properties of interest to steer the selection of solving techniques may also be useful for Planning. Another line of future work could be Conformant Planning [17], whose elevated complexity could be addressed by compilation to disjunctive ASP. In fact, the $dlv^{\mathcal{K}}$ system [18] supports Conformant Planning wrt action language $\mathcal{K}$.

Acknowledgments. This work was partly funded by DFG grant SCHA 550/8-2.

References

1. Biere, A., Heule, M., van Maaren, H., Walsh, T.: Handbook of Satisfiability. IOS Press (2009)
2. Nau, D., Ghallab, M., Traverso, P.: Automated Planning: Theory and Practice. Morgan Kaufmann (2004)
3. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
4. Kautz, H., Selman, B.: Planning as satisfiability. Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI'92), Wiley (1992) 359–363
5. Kautz, H., Selman, B.: Pushing the envelope: Planning, propositional logic, and stochastic search. Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI'96), AAAI/MIT Press (1996) 1194–1201
6. 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)
7. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: Engineering an incremental ASP solver. Proceedings of the Twenty-fourth International Conference on Logic Programming (ICLP'08), Springer (2008) 190–205
8. Rintanen, J., Heljanko, K., Niemelä, I.: Parallel encodings of classical planning as satisfiability. Proceedings of the Ninth European Conference on Logics in Artificial Intelligence (JELIA'04), Springer (2004) 307–319
9. Kautz, H., McAllester, D., Selman, B.: Encoding plans in propositional logic. Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR'96), Morgan Kaufmann (1996) 374–384
10. Blum, A., Furst, M.: Fast planning through planning graph analysis. Artificial Intelligence **90**(1-2) (1997) 279–298
11. Robinson, N., Gretton, C., Pham, D., Sattar, A.: SAT-based parallel planning using a split representation of actions. Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS'09), AAAI Press (2009) 281–288
12. Hsu, C., Wah, B., Huang, R., Chen, Y.: Constraint partitioning for solving planning problems with trajectory constraints and goal preferences. Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07), AAAI/MIT Press (2007) 1924–1929
13. Sideris, A., Dimopoulos, Y.: Constraint propagation in propositional planning. Proceedings of the Twentieth International Conference on Automated Planning and Scheduling (ICAPS'10), AAAI Press (2010) 153–160
14. Genesereth, M., Love, N., Pell, B.: General game playing: Overview of the AAAI competition. AI Magazine **26**(2) (2005) 62–72
15. Thielscher, M.: Answer set programming for single-player games in general game playing. Proceedings of the Twenty-fifth International Conference on Logic Programming (ICLP'09), Springer (2009) 327–341
16. Thielscher, M., Voigt, S.: A temporal proof system for general game playing. Proceedings of the Twenty-fourth National Conference on Artificial Intelligence (AAAI'10), AAAI Press (2010) 1000–1005
17. Smith, D., Weld, D.: Conformant Graphplan. In: Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI'98), AAAI/MIT Press (1998) 889–896
18. Eiter, T., Faber, W., Leone, N., Pfeifer, G., Polleres, A.: A logic programming approach to knowledge-state planning. Artificial Intelligence **144**(1-2) (2003) 157–211