A Complete Planner for Temporal Answer Set Programming*

Pedro Cabalar, Manuel Rey and Concepción Vidal

Department of Computer Science
University of Corunna, SPAIN
{cabalar,j.manuel.rey,concepcion.vidalm}@udc.es

Abstract. In this paper we present `tasplan`, a complete planner for temporal logic programs. The planner receives a planning specification as input, having the form of a temporal ASP program, and obtains as output one or several alternative (shortest) plans, if the problem is solvable, or answers that no solution exists, otherwise. The tool allows different search strategies, including informed search algorithms if the user defines a domain-dependent heuristics with additional program rules.

1 Introduction

The paradigm *Answer Set Programming* (ASP [12, 11]) has become a popular approach for practical Knowledge Representation and problem solving thanks to its simple semantics based on stable models [9], the availability of efficient solvers and their application in a wide spectrum of diverse domains [7]. Although the ASP modelling language and its associated solvers are designed for static combinatorial search problems, many ASP applications require handling a dynamic component, normally, dealing with transition systems over discrete time. An extension of ASP for temporal reasoning was proposed with the introduction of *Temporal Equilibrium Logic* (TEL) [4, 1], a combination of *Equilibrium Logic* [13] (the logical characterisation of stable models) with the usual modal temporal operators from *Linear-time Temporal Logic* (LTL [10, 14]). The first tools for TEL inference were based on model checking [2] and automata transformation methods [5]. Although these implementations were convenient for studying system properties or strong equivalence under the assumption of infinite traces, they were highly inefficient for solving planning problems. A more practical orientation came with the recent definition of TEL for finite traces [6]. This led to the implementation of system `telingo` [3], a temporal extension of the popular ASP solver `clingo` that relies on the same incremental solving strategy for finding the shortest plan in an efficient way.

The usual strategy in ASP planning is based on an iterative deepening search, incrementally increasing the length n of the searched plan until a solution is found [8]. This kind of planners are obviously *incomplete*: when the problem

* This work was partially supported by MINECO, Spain, grant TIC2017-84453-P, Xunta de Galicia, Spain (GPC ED431B 2019/03 and 2016-2019 ED431G/01, CITIC)

has no solution, the planner is trapped in an infinite loop, trying new values for n indefinitely[1]. On the other hand, the model checking tools [2, 5] mentioned before can detect non-existence of a plan, but they are not tailored for planning problems and result rather inefficient.

In this paper, we describe `tasplan`[2], a complete planner for temporal logic programs. `tasplan` receives a temporal logic program as an input. This temporal program describes the fluents that configure the states of the system, the actions that can be executed and the restrictions that have to be fulfilled. It will also determine the behaviour of the state transition system, the initial state and the goal state to be reached. The output will consist of one or several plans (sequences of actions) or the answer that there is no possible plan, so `tasplan` is the first complete planner for ASP. The strategy followed by `tasplan` consists in a classical search algorithm (we follow the pseudocode in [15]) maintaining a hash table and a fringe containing the next states to be explored (reaching an empty fringe is a guarantee that there exists no solution). The main distinctive feature in our search algorithm is that the generation of successor states relies on multiple calls to `clingo`[3] that is kept running in the background and is used through its Python API. Finally, `tasplan` can use different (uninformed and informed) search algorithms that can be selected through the command line call. Informed search incorporates the possibility of specifying a heuristic function that can be used to prune the search, when the planning problem is solvable. This heuristic is included in the specification of the problem as one more regular rule, something that provides a natural and declarative method for defining different heuristics.

2 Architecture

The proposed architecture (see Figure 1) has the form of a pipeline: `tasplan` receives as input a file with the specification of the problem to solve, which must be written in ASP. The problem formalisation must be fragmented in five `#program` blocks named as `types(t)`, `static`, `initial`, `dynamic` and `final(t)`. The input file is grounded using tool `gringo` to obtain a ground program. No further grounding will be required afterwards. In order to avoid that the grounder makes assumptions on dynamic atoms (representing actions and fluents), for instance, assuming that non-occurring actions are false, we precede those atoms by the grounder directive `#external`. This will let us manipulate the ground program on the fly, adding and removing (ground) dynamic atoms without the need of grounding again. After that, the solving module implements the planning algorithm that explores the state space and decides the existence of a plan. Each time a non-goal state S is picked from the fringe, the algorithm computes its possible successor states by a call to `clasp` on the ground program extended

[1] An interesting topic for future study would be determining integer bounds for the required number of steps n to obtain a plan.

[2] https://github.com/jmanuelrey/T-ASPlan/

[3] Systems `clingo`, `gringo` and `clasp` are available at `https://potassco.org`.

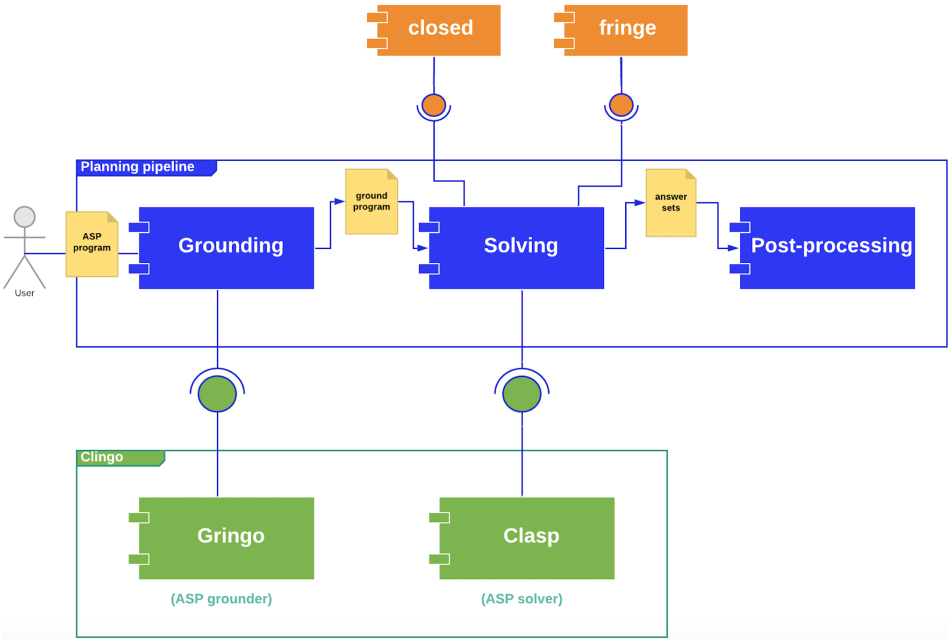

Fig. 1. `tasplan` architecture based on "Pipe and filter".

with the facts corresponding to S. The possible successor states are obtained as the multiple stable models. This process is repeated as many times as transition computations needed by the search algorithm. If any successor state matches with the goal specified in the problem formalisation, the solver ends, and the goal is passed to the last module. Otherwise, if `tasplan` explores the complete state space and no new state is generated, the tool determines the non existence of plan. The post-processing module prepares the data so it can be printed in a comfortable way for the user.

To conclude this section, we show in Figure 2 an example of input specification for `tasplan`. The program represents the well-known 8-puzzle problem, where a grid of 3×3 cells contains 8 tiles (from 1 to 8) and a hole (represented by 0). The possible actions are moving an adjacent tile to the hole, creating a new hole in its previous position. We assume some familiarity with `clingo` language. The time instant is represented here by the constant terms `t` and `t-1`. Perhaps the most relevant feature is the special predicate `heuristics(N,t)` that is used to specify a domain-specific heuristic function `N` for each state at time point `t`. In this particular case, the heuristics corresponds to the Manhattan distance of each tile to its goal position, and uses a `#sum` aggregate for all tiles.

```
#program types(t).
action(move(X,t)) :- dir(X).
fluent(pos(X,Y,Z,t)) :- tile(X), row(Y), col(Z).
fluent(goal(t)).

#program static.
row(1..3).
col(1..3).
tile(0..8).
cell(X,Y) :- row(X), col(Y).
% goal state
goalpos(1,1,1). goalpos(2,1,2). goalpos(3,1,3).
goalpos(4,2,1). goalpos(5,2,2). goalpos(6,2,3).
goalpos(7,3,1). goalpos(8,3,2). goalpos(0,3,3).
dir(up;down;right;left).
adj(X,Y,up,  X-1,Y) :- row(X),row(X-1),col(Y).
adj(X,Y,down, X+1,Y) :- row(X),row(X+1),col(Y).
adj(X,Y,left, X,Y-1) :- row(X),col(Y-1),col(Y).
adj(X,Y,right,X,Y+1) :- row(X),col(Y+1),col(Y).

#program initial.
pos(8,1,1,0). pos(6,1,2,0). pos(7,1,3,0).
pos(2,2,1,0). pos(5,2,2,0). pos(4,2,3,0).
pos(3,3,1,0). pos(0,3,2,0). pos(1,3,3,0).

#program dynamic(t).
% Manhattan heuristic
heuristics(N,t) :- N = #sum{ M,F : tile(F), F!=0, pos(F,X,Y,t),
    goalpos(F,X2,Y2), M=|X-X2|+|Y-Y2|}.
1 {move(D,t) : dir(D)} 1.
pos(0,Z,T,t) :- move(D,t), pos(0,X,Y,t-1), adj(X,Y,D,Z,T).
pos(P,X,Y,t) :- move(D,t), pos(0,X,Y,t-1), adj(X,Y,D,Z,T),
    pos(P,Z,T,t-1).
% Inertia
pos(P,Z,T,t) :- pos(P,Z,T,t-1), not -pos(P,Z,T,t).
-pos(P,Z,T,t) :- pos(P,X,Y,t), (X,Y)!=(Z,T), row(Z), col(T).

#program final(t).
goal(t) :- #count{X,Y: pos(P,X,Y,t), not goalpos(P,X,Y)} = 0.
```

Fig. 2. Encoding of an instance of the 8-puzzle problem in `tasplan` input language.

Table 1. Time measurements for different scenarios of 8-puzzle.

Plan length	`tasplan` (A*)	`tasplan` (breadth)	`telingo`
5	0.044s	0.409s	0.024s
9	0.068s	2.575s	0.036s
12	0.108s	13.809s	0.104s
15	0.634s	75.154s	0.282s
17	1.259s	143.978s	0.865s
20	3.615s	289.900s	0.868s
25	17.732s	551.520s	12.341s
28	47.271s	860.446s	44.203s
30	49.485s	951.516s	64.537s
31	155.139s	1326.927s	187.962s
∞	1438.047s	2294.307s	∞

3 Evaluation

We have performed a preliminary evaluation of `tasplan` efficiency in comparison to the incremental solving strategy. Table 1 shows the time results obtained for different instances of the classical 8-puzzle problem. The measurements have been taken on a computer with seven Intel© Xeon© E5504 processors with 4M cache, 2 GHz processor frequency, 4.80 GT/s, 4 Intel© QPI Core CPUs, 16 GB RAM, and 3TB of external storage. Each instance has been classified by the length (number of steps) of the shortest plan. This is represented in the leftmost column: the last line, with value ∞, corresponds to a scenario with no solution. The time values are organised in three columns: the first one corresponds to `tasplan` with an A* algorithm (using Manhattan distance as heuristics), the second is `tasplan` using an uninformed breadth search and the third one is the result of `telingo` with an analogous encoding (`telingo` does not allow heuristics specification). As we can see, `telingo` does not (and cannot) provide an answer to the last (unsolvable) scenario, whereas both versions of `tasplan` allow eventually deciding that there is no possible solution. In the rest of cases, `telingo` is clearly faster than breadth search in `tasplan`. When we allow domain specific heuristics, however, `tasplan` with A* has a similar performance to `telingo` and is even slightly better for longer plans. This comparison is not completely fair, in the sense that `telingo` does not use domain specific heuristics like the Manhattan distance. However, the truth is that the addition of such information to `telingo` could not be exploited anyway due to its iterative deepening strategy. This preliminary experiment shows the potential interest of exploiting user-defined domain heuristics for ASP planning.

4 Final conclusions

We have presented `tasplan`, a complete planner for temporal Answer Set Programming. For a temporal logic program specifying a planning scenario, `tasplan`

is capable of deciding the non-existence of a plan when the problem is unsolvable, outperforming previous tools based on model checking. When the problem has a plan, `tasplan` is still slower than current ASP incremental solvers, but its performance is comparable if we allow the introduction of domain specific heuristics, that can be exploited using standard informed search algorithms such as A*. For future work, we plan to modify the `tasplan` input language to accept `telingo` logic programs, so that both planners can be applied on the same specification. We also plan to study automated computation of domain-independent heuristics.

References

1. Aguado, F., Cabalar, P., Diéguez, M., Pérez, G., Vidal, C.: Temporal equilibrium logic: a survey. Journal of Applied Non-Classical Logics **23**(1-2), 2–24 (2013)
2. Cabalar, P., Diéguez, M.: STELP - a tool for temporal answer set programming. In: LPNMR'11. Lecture Notes in Computer Science, vol. 6645, pp. 370–375 (2011)
3. Cabalar, P., Kaminski, R., Morkisch, P., Schaub, T.: `telingo` = ASP + Time. In: Proc. of the 15th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'19) (2019)
4. Cabalar, P., Perez, G.: Temporal Equilibrium Logic: A First Approach. In: Proceedings of the 11$^{\text{th}}$ International Conference on Computer Aided Systems Theory (EUROCAST'07). pp. 241–248 (2007)
5. Cabalar, P., Diéguez, M.: Strong equivalence of non-monotonic temporal theories. In: Proceedings of the 14th International Conference on Principles of Knowledge Representation and Reasoning (KR'14). Vienna, Austria (2014)
6. Cabalar, P., Kaminski, R., Schaub, T., Schuhmann, A.: Temporal answer set programming on finite traces. TPLP **18**(3-4), 406–420 (2018)
7. Erdem, E., Gelfond, M., Leone, N.: Applications of answer set programming. AI Magazine **37**(3), 53–68 (2016)
8. Gebser, M., Kaufmann, B., Kaminski, R., Ostrowski, M., Schaub, T., Schneider, M.T.: Potassco: The potsdam answer set solving collection. AI Commun. **24**(2), 107–124 (2011)
9. Gelfond, M., Lifschitz, V.: The Stable Model Semantics For Logic Programming. In: Proc. of the 5$^{\text{th}}$ International Conference on Logic Programming (ICLP'88). pp. 1070–1080. Seattle, Washington (1988)
10. Kamp, H.: Tense Logic and the Theory of Linear Order. Ph.D. thesis, UCLA (1968)
11. Marek, V., Truszczyński, M.: Stable models and an alternative logic programming paradigm, pp. 169–181. Springer-Verlag (1999)
12. Niemelä, I.: Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm. Annals of Mathematics and Artificial Intelligence **25**(3-4), 241–273 (1999)
13. Pearce, D.: A New Logical Characterisation of Stable Models and Answer Sets. In: Proc. of Non-Monotonic Extensions of Logic Programming (NMELP'96). pp. 57–70. Bad Honnef, Germany (1996)
14. Pnueli, A.: The temporal logic of programs. In: Proceedings of the 18th Annual Symposium on Foundations of Computer Science. pp. 46–57. SFCS '77, IEEE Computer Society, Washington, DC, USA (1977)
15. Russell, S.J., Norvig, P.: Artificial Intelligence - A Modern Approach (3. internat. ed.). Pearson Education (2010)