Pdk: the System and its Language

Marta Cialdea Mayer, Carla Limongelli,
Andrea Orlandini, and Valentina Poggioni

Università di Roma Tre, Dipartimento di Informatica e Automazione
{cialdea,limongel,orlandin,poggioni}@dia.uniroma3.it

Abstract. This paper presents the planning system Pdk (Planning with Domain Knowledge), based on the translation of planning problems into Linear Time Logic theories, in such a way that finding solution plans is reduced to model search. The model search mechanism is based on temporal tableaux. The planning language accepted by the system allows one to specify extra problem dependent information, that can be of help both in reducing the search space and finding plans of better quality.

1 Introduction

Artificial Intelligence planning is concerned with the automatic synthesis of sequences of actions that, when executed, lead from a given initial state to a state satisfying some logics have been used in different perspectives in this context, either in the deductive view [10, 12], or by model checking [5], or as a language to add control knowledge to a specialised planner [1, 6]. The system presented in this work is a planner fully based on Linear Time Logic (LTL), planning problem into LTL in such a way that planning is reduced to model search. This corresponds to the idea of *executing temporal logics*, where executing a formula means building a model of it [7] (the application to planning was already sketchily proposed in [2]).

The system Pdk (Planning with Domain Knowledge) accepts the description of a planning problem, given in the planning language PDDL-K (Planning Domain Description Language with control Knowledge), that offers the possibility of specifying extra problem dependent and control information, that can be of help both in reducing the search space and finding plans of better quality. The specification of the planning problem is translated into a set S of LTL formulae in such a way that any model of S represents a plan solving the problem. The reduction consists of a "linear encoding" [3], that recalls the classical Situation Calculus representation of planning [11].

The small example that follows illustrates how a planning problem is encoded into LTL. In the *initial state*, a robot is in room A, its *goal* is to be in room B and the only *actions* it can perform is going from a location to another, specifically either from A to B (go_A_B), or from B to A (go_B_A). This problem is represented by the following set of LTL formulae:

$$
\begin{aligned}
S = \{\ & at\_A \wedge \neg at\_B, \Diamond at\_B, \\
& \Box(go\_A\_B \rightarrow at\_A),\ \Box(go\_B\_A \rightarrow at\_B), \\
& \Box(\bigcirc at\_A \equiv (at\_A \wedge \neg go\_A\_B) \vee go\_B\_A), \\
& \Box(\bigcirc at\_B \equiv (at\_B \wedge \neg go\_B\_A) \vee go\_A\_B)\}
\end{aligned}
$$

The first two formulae represent the initial state and goal of the problem (sometimes in the future the goal will be achieved). The two formulae in the second line above represent the preconditions for the executability of the two actions (the agent can move from a place only if it is there). The last two formulae are the LTL reformulation of Reiter's "successor state axioms" [11]. A model of S is the sequence of states $s_0, s_1, ...$ such that the only true atoms at s_0 are at_A and go_A_B, and the only true atom at s_i, for $i > 0$, is at_B. The plan corresponding to such a model is the sequence of actions $\langle go\_A\_B \rangle$.

Pdk, which is available at `http://pdk.dia.uniroma3.it/`, is implemented in Objective Caml (`http://caml.inria.fr/`) and its model search mechanism relies on the system `ptl`, an efficient implementation of proof search in LTL by means of tableaux techniques, developed in C by G. Janssen at Eindhoven University [9].

A first version of the planner was already presented in [4]. The experiments with that system raised the need for tools supporting the domain expert in the specification task, especially when stating domain specific and control knowledge. In fact, the domain expert charged of the description of the problem cannot be assumed to be a logician, and it is therefore important to have a simple, compact and easy-to-use planning language, similar to the special purpose formalisms widely adopted in the planning community.

The new release of the planner, briefly described in this paper, beyond being a more efficient implementation, accepts problem specifications written in a planning language where heuristic knowledge can not only be provided explicitly as a set of LTL formulae, but also as instances of a set of specific control schemata. Moreover, the system provides meta-level tools that can be of help in debugging the problem specification (synthetically described at the end of the next section). Such tools can be developed thanks to the fact that the whole planning domain is encoded into a logical theory.

2 The Planning Language and Off-line Checks

The planning language PDDL-K can be viewed as an extension of the ADL-subset of classical PDDL [8]. A full description of the language and its semantics, given by means of the translation into LTL, can be found at the above cited URL, together with a set of sample domains. Due to space restrictions, only a brief overview can be given here. The PDDL-K description of a planning problem follows a multi-sorted first-order syntax. However, as usual in classical planning, domains are finite and fixed, and (typed) quantification is actually an abbreviation for propositional formulae. For instance, $\forall x : t\, A(x)$ stands for $A(c_1) \wedge ... \wedge A(c_n)$, where $c_1, ..., c_n$ are all the constants of type t.

The specification of the problem contains the definition of the signature: type declarations, with associated sets of constants, and predicate declarations. In PDDL-K predicates are distinguished into *static* predicates (denoting properties that do not change over time) and *fluents*. The value of static predicates is declared as a *background theory*, consisting of a set of classical formulae. The background theory is completed with respect to static atoms, according to the closed world assumption: what is not classically derivable from the background theory is false. Therefore, each ground static atom is either true or false at each time point. All literals built up from fluents which are derivable from the background theory are also added to its completion. After completion, the background theory consists of a set of literals. It is used to simplify the encoding of the planning problem: each static atom is replaced by either True or False, and the same happens with fluent literals occurring in the completed theory. The theory is also used to filter out operator instances, by elimination of those actions whose preconditions or effects are inconsistent.

The other fundamental declarations in the description of the problem are the specification of the initial state, goal and operators. Since, like in classical planning, the knowledge about the initial state is assumed to be complete, the initial state is completed wrt fluents. The description of the initial state may also contain temporal operators. Non-classical formulae, that will just be added to the encoding, may be used to specify intermediate goals that must be achieved or actions that must be performed. For instance, the description of the initial state can contain $\Diamond(\exists x : location\, go(x, bank) \wedge \Diamond \exists x : location\, go(x, post\_office))$: during plan execution, the agent must sooner or later go (from some place x) to the bank and afterwards to the post office.

Control knowledge can be specified either explicitly, as a set of temporal formulae, or according to a predefined set of *control schemata* in the description of operators. Such schemata allow one, for instance, to specify that a given action should be performed only if given conditions hold, or that it should be performed whenever possible (see the example given below).

A peculiarity of the language is that it is possible to refer to actions in formulae. Beyond fluents and static predicates, in fact, atoms can be build out of predicates denoting actions. This allows one to specify for instance that the execution of a given action must be either accompanied or followed by others, or that the effect of performing two actions together is different from what can be obtained by executing each of them separately.

In the specification of control information, the possibility to refer to the goal to be achieved, as well as to what is true in the initial state can be of use, especially to eliminate useless actions. To this aim, the syntax of PDDL-K formulae is extended by means of the unary modal operators *goal* (which can dominate only literals) and *initially* (which can dominate only atoms): *goal* ℓ means that ℓ is a goal of the problem, *initially* p means that p is true in the initial state.

In order to give the flavour of the language, let us consider the simple domain where a robotic agent has to move some objects from/to different locations by

means of a briefcase. Constants are either of sort *location* or *object*. The fluents are $atRobby(x)$ (the robot is at location x), $in(x)$ (the object x is in the robot's briefcase), $at(x, y)$ (the object x is at location y). The complete specification of this class of problems can be found at the above cited URL. Here, we only show how the *take* action can be specified using some of the available control schemata.

```
(:action take :parameters (x - object y - location)
              :precondition (atRobby y) (at x y)
              :effect (in x) (not (at x y))
              :only-if (not (goal (at x y))) (initially (at x y))
              :s-asap )
```

The parameters of the action are the object x and the place y where both the object and the robot are (this is specified in the action precondition). After the execution of the action, the object is no more at y, but inside the briefcase (the effect of the action). The `:only-if` field specifies additional (heuristic) restrictions on the action execution: take x from place y only if x is not already at its destination and x is at y in the initial state. The `:s-asap` field (*strong* asap) requires all instances of the action to be performed "as soon as possible". If $O = \neg goal(at(x, y)) \wedge initially(at(x, y))$ and $P = atRobby(y) \wedge at(x, y)$, the encoding of the two control fields above (`:only-if` and `:s-asap`) are (the propositional counterparts of) the following formulae, respectively:

$$\Box\, \forall x : object\, \forall y : location(take(x, y) \rightarrow O)$$
$$\Box\, \forall x : object\, \forall y : location(P \wedge O \rightarrow take(x, y))$$

Following the guidelines given by action oriented control schemata, the specification task becomes easier, but surely not free from errors. It often happens in fact that the addition of too strong control requirements makes the problem unsolvable. In order to help the domain expert, the system Pdk provides some meta-level tools that can be of help in debugging the specification. The system allows one to check for the consistency of the theory encoding the kernel problem (i.e. excluding control knowledge and goal), and to check whether such a theory becomes inconsistent with the addition of control knowledge. Sometimes, moreover, it happens that, although the theory is consistent, the goal cannot be achieved because some important actions can never be performed, since they would violate some too strong control requirement. The system allows one to check actions, one by one, to verify their executability.

3 Conclusion

Some experiments have been carried out in order to verify whether the expressive power of the language compensates for the loss in time efficiency that often derives from the use of general logical procedures (especially if based on exhaustive search) and/or allows the planner to find solutions of better quality. The performances of Pdk have been compared with some planners that showed best

performances in the last International Planning Competitions (IPC 2002 and IPC 2004). The experiments – whose results cannot be reported in detail here for space reasons – show that Pdk outperforms optimal planners, such as SATPLAN_2004. With respect to suboptimal planners, such as LPG, the relative performances of the planners in terms of execution times vary in dependence of the planning domains: in some cases LPG is much faster, in other domains the execution times are comparable and sometimes Pdk is faster than LPG. In nearly all experimented cases, however, Pdk finds shorter plans.

Comparing Pdk with other planners that allow for the employment of heuristic knowledge, such as TLPLAN [1], involves, beyond efficiency, also expressiveness considerations. Although TLPLAN is faster than Pdk, the specification of heuristic knowledge is often quite long and cumbersome. One of the reasons is that all statements must be made in terms of fluents. The fact that, in PDDL-K, actions are represented by atoms allows for much simpler control formulae, that, in turn, can often be reduced to the addition of even simpler control fields in operators specifications. We believe that this is not a secondary issue, since the statement of correct control knowledge is often a subtle and difficult task.

References

1. F. Bacchus and F. Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116:123–191, 2000.
2. H. Barringer, M. Fisher, D. Gabbay, and A. Hunter. Meta-reasoning in executable temporal logic. In *Proc. of KR'91*, pages 40–49, 1991.
3. S. Cerrito and M. Cialdea Mayer. Using linear temporal logic to model and solve planning problems. In *Proc. of AIMSA'98*, pages 141–152, 1998.
4. M. Cialdea Mayer, A. Orlandini, G. Balestreri, and C. Limongelli. A planner fully based on linear time logic. In *Proc. of AIPS-2000*, pages 347–354, 2000.
5. A. Cimatti, E. Giunchiglia, F. Giunchiglia, and P. Traverso. Planning via model checking: a decision procedure for $\mathcal{AR}$. In *Proc. ECP-97*, pages 130–142, 1997.
6. P. Doherty and J. Kvarnström. TALplanner: A temporal logic based planner. *AI Magazine*, 22:95–102, 2001.
7. M. Fisher and R. Owens. An introduction to executable modal and temporal logics. In *Executable modal and temporal logics (Proc. of the IJCAI'93 Workshop)*, pages 1–20, 1995.
8. M. Fox and D. Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:61–124, 2003.
9. G. L. J. M. Janssen. *Logics for Digital Circuit Verification. Theory, Algorithms and Applications*. CIP-DATA Library Technische Universiteit Eindhoven, 1999.
10. J. Koehler and R. Treinen. Constraint deduction in an interval-based temporal logic. In *Executable Modal and Temporal Logics, (Proc. of the IJCAI'93 Workshop)*, pages 103–117, 1995.
11. R. Reiter. *Knowledge in Action: logical foundations for describing and implementing dynamical systems*. MIT Press, 2001.
12. B. Stephan and S. Biundo. Deduction based refinement planning. In *Proc. of AIPS-96*, pages 213–220, 1996.