Linear Temporal Logic as an Executable Semantics for Planning Languages

Marta Cialdea Mayer Carla Limongelli
Andrea Orlandini Valentina Poggioni

Università degli Studi di Roma TRE

Abstract

This paper presents an approach to artificial intelligence planning based on linear temporal logic (LTL). A simple and easy-to-use planning language is described, PDDL-K (Planning Domain Description Language with control Knowledge), which allows one to specify a planning problem together with heuristic information that can be of help for both pruning the search space and finding better quality plans. The semantics of the language is given in terms of a translation into a set of LTL formulae. Planning is then reduced to "executing" the LTL encoding, i.e. to model search in LTL. The feasibility of the approach has been successfully tested by means of the system PDK, an implementation of the proposed method.

1 Introduction

Automated planning is a field of Artificial Intelligence that studies methods and algorithms to find action sequences (*plans*) achieving some given goals. If the concurrent execution of different actions is allowed, then a (parallel) plan is a sequence of *sets* of actions. In general, a *planning problem* is specified by describing all the executable actions, some goal, and what is known about the initial state of the world. A *planner* is a piece of software that takes as input the description of a planning problem and outputs a sequence of actions that, if executed, transforms the initial state into a state satisfying the goals. In order to make the planning task more precise, various *planning models* can be considered. Each of them formalizes different assumptions on the class of problems that may be dealt with, in particular on the nature of actions (see for instance [21]). In *classical planning*, to which most work has been devoted so far, actions are assumed to be deterministic and the world is finite, discrete, fully observable, and with no exogenous events.

"Intuitively, planning is logical reasoning of some kind" [9], and in fact, beyond planners based on specialised algorithms, different logical approaches to planning have been proposed. In such approaches, a planning problem is

encoded by a logical theory and plans are synthesised by means of some general logical procedure. Representative of the great variety of approaches and logics used to this aim are [2], [8], [9], [11], [18], [25], [29], [30], [31], [32], [35], [37], [38], [39], [41].

Traditionally, planning has been formalised as deduction: plans are generated by constructive proofs of so-called *plan specification formulae*, stating that there exists a plan leading from the initial state to a state satisfying the goal. The best-known logical formalisation of planning in the deductive view is the Situation Calculus [33]. [37] (Chapter 10) presents some Situation Calculus planning systems, written in GOLOG, a high-level logic programming language for agents, based on the Situation Calculus and implemented in Prolog. The GOLOG planners show one of the main strong points of the logical approaches to planning: thanks to the expressive power of logical languages, it is possible to enrich the description of planning problems with problem dependent information, that can be of great help both in reducing the search space and finding better quality plans.

Dually to the "planning as derivability" approach, the "planning as satisfiability" paradigm was introduced by [25], and carried on in [26] and [28]. According to this paradigm, a planning problem is encoded by a logical theory, modelling the rules governing the world evolution, in such a way that any model of the theory corresponds to a valid plan. Based on the above cited works is the planner SATPLAN,[1] the winner for optimal deterministic planning in the last two international planning competitions (IPC 2004 and IPC 2006).

In SATPLAN, the target logic of the encoding is classical propositional logic. This planner can thus be considered as an application to planning of the approach based on "executing *propositional* formulae" – where executing a formula means building a model of it [19]. In order to represent time in classical propositional logic, the language of SATPLAN uses indexed propositional letters, the index playing the role of a time stamp. This forces the planner to fix a time limit (*layer*) in advance, in which goals have to be achieved. In order to be complete, plan search proceeds by iteratively deepening such a limit. When solution plans are fairly long, this method can be quite hard.

Moreover, SATPLAN does not decide, in general, the problem of existence of a solution. In fact, when a problem has no solution, the system may not terminate, unless a maximal layer or a timeout is fixed in advance. Note that the same can be said of planners based on predicate logic, such as the GOLOG planners and those based on the so called "Answer Set Planning" approach [42, 31].

The work presented in this paper conforms to the "planning as satisfiability" paradigm but, differently from [25], the logic used to encode planning problems is propositional Linear Temporal Logic (LTL). This approach carries on the idea of *executing temporal logics* [7] and its application to planning (already sketchily proposed in [8]).

The choice of LTL is due to two main reasons. First of all, it allows a simple and natural representation of a world that changes over time. Secondly, it is decidable. These two features together free planning in LTL from the above mentioned inconveniences of planning in (either propositional or first-order) classical logic.

[1] SATPLAN is available at `http://www.cs.washington.edu/homes/kautz/ satplan/`.

Moreover, domain dependent knowledge can be expressed in LTL (see [4], [15]), as well as domain restrictions in the style of [13] and intermediate tasks, like in [3].

A prototype system that encodes the whole planning problem into LTL and reduces planning to model search was presented in [14]. The first experiments with that system showed however that stating domain dependent and control knowledge correctly as LTL formulae can be quite a delicate task, since the domain expert cannot be assumed to be a logician. Moreover, with the addition of restrictions on the searched plans, completeness may be lost and it may happen that the planner finds no solution even if one exists.

Such problems are quite general and affect any planner that allows for the specification of extra problem-dependent knowledge. They can be addressed by providing tools supporting the domain expert in the specification task, in particular

(i) debugging tools, and

(ii) a simple, compact and easy-to-use planning language (similar to the special purpose formalisms widely adopted in the planning community), which allows the user to include different forms of domain specific knowledge.

A proposal to achieve the second aim above is represented by the planning language PDDL-K (Planning Domain Description Language with control Knowledge), which is described in this paper. PDDL-K guides the user in the specification of heuristic knowledge, providing a set of *control schemata*. The language is given an *executable* semantics by means of its translation into LTL.

The system PDK (Planning with Domain Knowledge)[2] implements the semantics of PDDL-K: it accepts PDDL-K as input language, translates the problem description into its LTL representation and reduces planning to model search. Moreover, in order to achieve the first of the two above mentioned aims, PDK provides tools that can help the user in the debugging phase. Such tools exploit the fact that the planning problem is entirely encoded as a logical theory. In fact, different parts of the encoding are subjected to logical tests, allowing one to identify where possible bugs come from.

The paper is organised as follows. Section 2 briefly recalls the syntax and semantics of LTL and shows how plans can be characterised in terms of temporal models. Section 3 formally defines classical planning problems, by directly using the language PDDL-K, and shows how the specification of a planning problem is encoded into LTL. Some examples of problems that are not easily expressible in classical planning formalisms are also given. Since the correct formulation of domain knowledge is a delicate issue, Section 4 points out some guidelines that can help in this task and describes how different forms of heuristic information can be specified in PDDL-K, along with their LTL semantics. The tools offered by the system PADOK to perform an off-line check of the specification are described in Section 5. In Section 6 the language PDDL-K is compared with the languages of other planners allowing for the specification of control knowledge

[2] PDK is a new, more efficient implementation of the prototype presented in [14], offering also new features and tools. It is implemented in Objective Caml (available at `http://caml.inria.fr/`) and C. The planner can be downloaded from `http://pdk.dia.uniroma3.it/`, together with a set of sample domains and a user manual.

and some experimental results are presented, comparing the performances of PDK with other planning systems. Section 7 concludes this work.

2 Linear Temporal Logic, temporal interpretations and plans

The language of linear temporal logic considered in this paper extends classical propositional logic by means of the unary modal operators $\Box$ (always), $\Diamond$ (eventually) and $\bigcirc$ (next).

A temporal structure is a countably infinite sequence of elements called *states* or *time points*. A temporal interpretation $\mathcal{M}$ consists of a temporal structure $\langle s_0, s_1, s_2, \ldots\rangle$ and a mapping $v_{\mathcal{M}} : \mathbb{N} \to 2^P$, where P is the set of propositional letters of the language. For any $k \in \mathbb{N}$, $v_{\mathcal{M}}(k) \subseteq P$ is the set of propositional letters holding at state s_k. The satisfiability relation $\mathcal{M}_k \models A$, for $k \in \mathbb{N}$, is inductively defined as follows:

1. $\mathcal{M}_k \models p$ iff $p \in v_{\mathcal{M}}(k)$, for any propositional letter $p \in P$.
2. $\mathcal{M}_k \models \neg A$ iff $\mathcal{M}_k \not\models A$.
3. $\mathcal{M}_k \models A \wedge B$ iff $\mathcal{M}_k \models A$ and $\mathcal{M}_k \models B$.
4. $\mathcal{M}_k \models A \vee B$ iff either $\mathcal{M}_k \models A$ or $\mathcal{M}_k \models B$.
5. $\mathcal{M}_k \models A \to B$ iff either $\mathcal{M}_k \not\models A$ or $\mathcal{M}_k \models B$.
6. $\mathcal{M}_k \models \Box A$ iff for all $j \geq k$, $\mathcal{M}_j \models A$.
7. $\mathcal{M}_k \models \Diamond A$ iff there exists $j \geq k$ such that $\mathcal{M}_j \models A$.
8. $\mathcal{M}_k \models \bigcirc A$ iff $\mathcal{M}_{k+1} \models A$.

Truth is satisfiability in the initial state: a formula A is true in $\mathcal{M}$ (and $\mathcal{M}$ is a model of A) iff $\mathcal{M}_0 \models A$. Truth of sets of formulae is defined as usual.

In order to define the correspondence between temporal interpretations and plans, we recall that a classical planning problem is described distinguishing *fluents* (propositions about the world) and *actions*. Accordingly, we assume that a planning problem is described in a temporal language containing two disjoint sets of atomic propositions: *Actions*, whose elements denote actions, and *Fluents*, denoting facts about the world. If $a \in \mathit{Actions}$, the intended meaning of a holding at state s_k of a temporal model is that the action denoted by a is performed at state s_k.

Under the assumptions of classical planning, the interpretations of the language are candidates for representing solution plans. Since a plan is a *finite* sequence of (sets of) actions, we are actually interested only in finite initial fragments of temporal interpretations (up to the achievement of the goal), in particular interpretations $\mathcal{M}$ falsifying all $a \in \mathit{Actions}$ from some state onwards. We shall abuse language and call such interpretations *finite*. If $\mathcal{M}$ is a finite temporal interpretation, then it determines the (parallel) plan $P = \langle A_0, ..., A_n\rangle$ such that for every $i = 0, ..., n$ and action $a \in \mathit{Actions}$, $\mathcal{M}_i \models a$ iff $a \in A_i$, and for all $k > n$ and all $a \in \mathit{Actions}$, $\mathcal{M}_k \not\models a$ (i.e. no action is performed after

state n). If the plan $P = \langle A_0, ..., A_n \rangle$ determined by the finite interpretation $\mathcal{M}$ is a solution plan of the problem with final goal G, then $\mathcal{M}_{n+1} \models G$.

In Section 3 we show how to build a temporal theory T_Π, for each classical planning problem Π, that is a "correct and complete" encoding of the problem Π, i.e. such that every finite model of T_Π determines a plan solving Π, and every solution plan for Π is determined by some model of T_Π.

The basics of the LTL encoding of a planning problem can be sketchily illustrated here by the following small example: 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, or from B to A. The problem can be described in the syntax of PDDL, the present standard Planning Domain Description Language [20], as follows:

```
(:predicates (at ?x))              (:objects A B)
(:init (at A))                     (:goal (at B))
(:action go :parameters (?from ?to)
     :precondition (at ?from)
     :effect (and (at ?to) (not (at ?from))))
```

The *declarations* in the first line above describe the signature of the language (the unary predicate *at* and the two constants A and B); the second line declares the initial state and goal, the last lines describe the parameters, precondition and effect of the operator *go*. An "operator" is a (parametrized) *action schema* representing the whole set of its ground instances. From now on, only ground instances of operators will be called "actions".

The signature of the target language contains a propositional letter for each ground instance of the fluent and operator, i.e. at_A, at_B, go_A_B, go_B_A (going from x to x is excluded, since such actions have contradictory effects). The problem itself is represented by the following set of formulae:

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

The first formula represents the initial state; the second the 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 robot can move from a place only if it is there). The last two formulae are an LTL reformulation of Reiter's "successor state axioms" [37]: at any non-initial state, the robot is at place x if and only if either it was already there and did not go away, or it has just arrived at x. One of the models 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 $\langle \{go\_A\_B\} \rangle$, consisting of a single action set.

Once a planning problem is encoded by a set of temporal formulae, planning is reduced to model construction. To this aim, the planner PDK uses the system PTL, an efficient implementation of proof search in LTL by means of tableaux techniques [43], developed in C by G. Janssen at Eindhoven University [23]. PTL is called in "satisfiability mode" on the encoding S of the planning problem and, if this is satisfiable, it outputs a representation of a complete and open tableau for S. PDK extracts a plan from the first open branch of the tableau: node labels

(sets of literals) are cleaned up by keeping only (positive) atoms representing actions.

3 Planning problems and their LTL encoding

A planning problem is usually described specifying what holds in the initial state, what the goals are and which actions can be performed to change the world. In classical planning, it is assumed that the initial state can be represented by a (classical) formula I that completely describes the world, i.e. it is assumed that for any fluent R, it is known whether R is initially true or false. There is a single final goal, that can be described by a (classical) formula G built up from fluents. Each action is described specifying its *preconditions* and *effects* on fluents; when describing the effects of an action, only the action changes are specified, assuming that all the rest stays unchanged.

In classical planning formalisms, like STRIPS [17] and PDDL [20], important restrictions are usually imposed on the syntactic form of each of the above components. For instance, actions cannot be mentioned in action preconditions or effects.

In what follows we describe the kernel of the language PDDL-K, that can be viewed as an extension of the ADL-subset [34] of classical PDDL [20], and give its semantics in terms of a translation into a set of LTL formulae (the features of PDDL-K devoted to the representation of heuristic knowledge are presented in Section 4).

A planning problem can be represented by a set of LTL formulae according to different encodings, as shown in [12]. The semantics of PDDL-K consists of a simple form of progression encoding, that recalls the encoding of planning problems in the Situation Calculus [36] and the linear encoding of [24]. Such an encoding schema is provably correct and complete.

The description that follows of the language PDDL-K and its semantics will be illustrated by use of an example, the *teatime domain*, one of the classical problems in artificial intelligence planning. It consists of the class of problems where a robot has to deliver tea to the inhabitants of a given number of rooms, where one of the rooms contains a cup-stack and another a tea-machine. Each room is connected to the hallway and possibly to other rooms.[3] Though easy to understand and describe, problems in the teatime domain have fairly long solution plans, and are not so easy to solve automatically.

3.1 Signature

Like in PDDL, the PDDL-K description of a planning problem follows a multi-sorted first-order syntax, where however each domain is finite and fixed. The specification of the problem contains the definition of the signature: type, constant and predicate declarations. In particular, wrt type declarations, subtyping is allowed. For each type, a finite set of constants is specified, naming the objects of the domain.

[3] This domain is a simplification of the teatime domain in the repository of the European Planning Network of Excellence (PLANET), that can be found at `http://scom.hud.ac.uk/planet/repository/`. The original domain is a multi-robot scenario: two robots cooperate to serve tea; each robot is only allowed in some rooms and they meet in the hallway to exchange cups.

For instance, in our description of the teatime domain, types consist of locations and rooms, where *location* is a supertype of *room*:

```
(:types room - location /* room is a subtype of
                           location */
        location)
```

In the problem with four rooms, constants may be declared as follows:

```
(:objects hallway - location
          room1 room2 room3 room4 - room)
```

From such declarations, it follows that `room1, room2, room3, room4` are also locations.

In PDDL-K predicates are distinguished into *fluents*, denoting properties of the world that may change over time, and *static predicates*, whose interpretation is fixed. The declaration of the predicate symbols of the language associates a type to each of them.

In the teatime domain, fluents (`:predicates`) and static predicates (`:static`) can be defined as follows:

```
(:predicates (at ?x - location)
             (hascup)
             (fullcup)
             (ordered ?x - room))
(:static  (connected ?x ?y - location)
          (cupstack ?x - room)
          (teamachine ?x - room))
```

(identifiers beginning with a question mark are variables).

A fluent $at(x)$ is true when the robot is at location x; *hascup* is true if the robot is holding a cup; $fullcup$ is true when the robot is holding a cup full of tea; $ordered(x)$ holds when the inhabitant of room x has ordered tea and it has not been served yet; $teamachine(x)$ means that x is the room where the teamachine is, $cupstack(x)$ that the cupstack is in room x, and $connected(x, y)$ that the two locations x and y are directly connected by a door.[4]

Each ground atom in the language of the PDDL-K-specification is mapped to a propositional letter. We will continue using a first order syntax also for LTL formulae, in order to enhance readability. Typed quantification will be used as 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.

PDDL-K also accepts simple arithmetical formulae, built up from integers, (quantified) variables, arithmetical functions and predicates. Their semantics is operational: the truth value of an arithmetical atomic formula is computed just by evaluating it, like in TLPLAN [4]. Equality can also be applied to non-numeric arguments and is treated according to the assumption that the objects in each domain are pairwise distinct: if t and u are non-arithmetical terms (i.e. they are constants), then $t = u$ holds iff t and u are identical.

[4]Here and in the following, we shall use the LISP-like syntax of PDDL when showing pieces of PDDL-K code, and go on using the ordinary logical notation for formulae in the text.

3.2 Background theory

A specific section contains the *background theory*, i.e. knowledge about static predicates. It contains formulae without temporal operators, that are meant to be true throughout time, i.e. they represent facts that do not vary over time (*state invariants*). The background theory is completed with respect to static predicates, 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. This set 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. As will be better explained later on, the theory is also used to filter out operator instances, by elimination of those actions whose preconditions or effects are inconsistent.

It is worth pointing out that, since the completed theory is a set of literals, in order to carry out simplification there is no need to test derivability, but only set membership.

> In our example, the background theory contains knowledge about the topology of the rooms and the location of the cupstack and the teamachine:
>
> ```
> (:theory  (forall (?x - room) (connected ?x hallway))
>           (teamachine room1) (cupstack room2)
>           (connected room1 room3)
>           (connected room2 room4))
> ```

3.3 Initial state and goal

The other basic declarations in the description of the problem are the specification of the initial state, goal and operators. Knowledge about the *initial state* is specified by means of a set of formulae.

> For instance, if in the initial state the robot is in the first room and all the rooms have ordered tea, we can declare:
>
> ```
> (:init (at room1)
>        (forall (?x - room) (ordered ?x)))
> ```

Negative information need not be stated explicitly. Since knowledge about the initial state is assumed to be complete (like in classical planning), the initial state is completed wrt fluents. The LTL encoding of the initial state consists then of the set S_0 of literals built up from fluents as follows. Let $Init$ be the set of formulae declared in the `:init` section of the specification, and K the completed background theory. For any fluent R, if $Init \cup K \models R$ then $R \in S_0$, otherwise $\neg R \in S_0$.

> In our example, the encoding of the initial state is
>
> $$\begin{aligned} S_0 = \{ \quad & at(room1),\ \forall x : room\ ordered(x), \\ & \neg at(hallway), \neg at(room2), \neg at(room3), \neg at(room4), \\ & \neg hascup, \neg fullcup\ \} \end{aligned}$$

Attention must be payed to disjunctive information in the `:init` section. For instance, if the signature contains only the unary predicate p and constants a and b, the background theory is empty and the declaration of the initial state is `(:init (or (p a)(p b)))`, then the encoding of the knowledge about the initial state is $\{\neg p(a), \neg p(b)\}$, since neither $p(a)$ nor $p(b)$ is derivable from $p(a) \vee p(b)$.

The description of the initial state may also contain temporal operators. Non-classical formulae in the `:init` declaration, that will just be added to the encoding, may be used to specify intermediate goals that have to be achieved or actions that have to be performed. For instance, in a domain containing a *go* operator ($go(x, y)$: the agent moves from x to y), the description of the initial state may 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 to the bank and afterwards to the post office.

The specification of the final goal is given by any first-order formula G, and its encoding is the formula $\Diamond G$ (the goal will eventually be true).

For instance:

```
(:goal (forall (?x - room) (not (ordered ?x))))
```

is translated into:

$$\Diamond \forall x : room\ \neg ordered(x)$$

Before dealing with the representation of actions, we observe that the theoretical computational complexity of the encoding of a PDDL-K problem into LTL is at least exponential in the length of the propositional translation of the problem specification. This is because of the derivability tests needed to complete the background theory and the description of the initial state. Therefore, the presence of a complex background theory can have a heavy influence on the encoding time. However, in practice the encoding time is often a small percentage of the total execution time: the average encoding time in solving about 100 problems in 7 different domains is less than 8% of the total execution time.

3.4 Operators

The kernel description of each operator specifies its name, parameters (with associated type), preconditions and effects.

The actions allowed in the teatime domain can be described as follows:

```
(:action getcup
     :parameters (?x - room)
     :precondition (at ?x) (cupstack ?x) (not (hascup))
     :effect (hascup))
(:action fillcup
     :parameters (?x - room)
     :precondition (at  ?x) (teamachine ?x)
                   (hascup) (not (fullcup))
     :effect (fullcup))
```

```
(:action deliver
     :parameters (?x - room)
     :precondition (at ?x) (ordered ?x) (fullcup)
     :effect (not (ordered ?x)) (not (fullcup))
             (not (hascup)))
(:action go
     :parameters  (?from ?to - location)
     :precondition (or (connected ?from ?to)
                       (connected ?to ?from))
                   (at ?from)
     :effect  (at ?to) (not (at ?from)))
```

Preconditions can have any form. For instance, the first formula in the precondition of the *go* operator above is a disjunction. Such a precondition dispenses us to explicitly declare, in the background theory, that *connected* is a symmetric relation. Conditional and universally quantified effects are also allowed. For instance, a *go* operator with a single parameter can be declared as follows:

```
(:action go :parameters (?to - location)
   :precondition (forsome (?from - location)
                      (and (at ?from)
                           (or (connected ?from ?to)
                               (connected ?to ?from))))
   :effect (at ?to)
           (forall (?from - location)
                (when (at ?from) (not (at ?from)))))
```

Here, `forsome` is the existential quantifier and `when` is used for conditional effects:

```
(forall (?x - location) (when (at ?x) (not (at ?x))))
```

means that for every location x, if the robot is at x when performing a *go* action, it will no longer be at x in the next state.

As already said in Section 2, each ground operator instance, called "action", is mapped to a propositional letter, just like atoms representing facts about the world. However, every action with contradictory preconditions or effects is eliminated from the encoding (and replaced everywhere by $false$). For example, any instance of $go(x, y)$ where $x = y$ is contradictory: its post-condition, in fact, can never be satisfied; such actions are automatically ruled out. Actions are eliminated also when they are inconsistent with the background knowledge about the domain. For example, if room 2 is not connected to room 3 (and it will never be), then the actions $go(room2, room3)$ and $go(room3, room2)$ could never be executed. Action filtering by use of the background theory often dramatically reduces the search space.

> In the teatime problem with four rooms, more than 50% of the total number of operator instances are contradictory, and the percentage increases with the dimension of the problem: in the 20 rooms problem, it becomes more than 80%.

Knowledge about actions and their effects on the world is encoded by means of several groups of formulae.

Action preconditions. For every action a, the encoding contains a formula of the form $\Box(a \rightarrow \pi_a)$, where π_a represents the preconditions for the executability of a: "at any time, a is performed only if its preconditions π_a hold".

Some of the action precondition axioms in the teatime domain are:

$\Box(getcup(room2) \rightarrow at(room2) \wedge \neg hascup)$,
$\Box(fillcup(room1) \rightarrow at(room1) \wedge \neg fullcup \wedge hascup)$,
$\forall x : location\ \Box(deliver(x) \rightarrow at(x) \wedge ordered(x) \wedge fullcup)$,
$\Box(go(room1, hallway) \rightarrow at(room1))$,
$\Box(go(room1, room3) \rightarrow at(room1))$

Note that, thanks to the background knowledge and the consequent simplifications, static predicates never appear in the encoding: since $cupstack(x)$ is true if and only if $x = room2$, $cupstack(room2)$ is replaced by $true$ and it does not appear among the preconditions of $getcup(room2)$. All the other instances of $getcup(x)$ with $x \neq room2$ are eliminated because $cupstack(x)$ is in this case replaced by $false$.

Incompatibility among actions. A set of formulae describes incompatibility relations between actions. Obviously, if two actions a and b are incompatible because they have conflicting preconditions or effects, this need not be represented explicitly. In fact, no model of a complete encoding can have a and b true at the same time point. But if a deletes a precondition of b (or *vice-versa*), then their incompatibility has to be explicitly represented. For every action a, the encoding contains a formula of the form $\Box(a \rightarrow \neg a_1 \wedge ... \wedge \neg a_n)$, where $a_1, ..., a_n$ are all the actions whose incompatibility with a must be made explicit.

For example:

$$\Box(go(room2, room4) \rightarrow \neg go(room2, hallway) \wedge \neg deliver(room2) \wedge \neg fillcup(room2))$$
$$\Box(go(hallway, room1) \rightarrow \neg go(hallway, room2) \wedge \neg go(hallway, room3) \wedge \neg go(hallway, room4) \wedge \neg deliver(hallway))$$

Action effects. For every ground instance R of a fluent, two formulae are computed from the operators descriptions: G_R^+ specifies all the conditions that can lead to change the truth value of R from false to true, and G_R^- specifies all the conditions that can lead to change the truth value of R from true to false. The encoding includes the following formula, for any fluent R:

$$\Box(\bigcirc R \equiv G_R^+ \vee (R \wedge \neg G_R^-))$$

"At any non-initial time point (say s_{n+1}), R holds iff at the previous state (s_n) either some action having R as effect is performed or else R holds and nothing is done that causes $\neg R$". This is a paraphrase of Reiter's *successor state axiom* [36] into LTL.

For instance:

$$\forall x : location\ \Box(\bigcirc at(x) \equiv \exists y : location\ go(y,x) \vee (at(x) \wedge \neg\exists y : location\ go(x,y)))$$

$$\Box(\bigcirc fullcup \equiv fillcup(room2) \vee (fullcup \wedge \neg\exists x : location\ deliver(x)))$$

3.5 Actions as formulae

The fact that actions are explicitly represented by formulae is one of the main features of the language. In order to appreciate the flexibility deriving from this fact, let us consider the following example, borrowed from [1], that is not easily and naturally representable in most planning languages. In order to open the door to the Computer Science Building at Rochester, both hands must be used: a spring lock must be held open with one hand, while the door is pulled open with the other hand. Unless the lock is held open, it snaps shut. This is an example where the effect of two actions performed together is different from the sum of their effects. Let us consider the (propositional) language with the single fluent *open* (the door is open) and the two operators *pull_door* and *hold_lock* (with no parameters). The following is a correct PDDL-K specification of a problem that can only be solved by executing the two actions together:

```
(:predicates open)
(:goal open)
(:action pull_door
         :effect (when hold_lock open))
(:action hold_lock
         :effect (when pull_door open)))
```

This is an example with conditional effects, where conditions are actions. In general, any formula can be used to express conditions, with no syntactical restrictions.

4 Control Knowledge

The kernel specification of a planning problem, i.e. what necessarily has to be known in order to solve the problem, can be enriched with knowledge about *how* to solve the problem, in order to find a solution faster, as well as to obtain a better quality solution. For instance, the domain expert can know that some actions are useless under certain circumstances or that they should be preferred in other cases. Information of this kind can be added to the LTL encoding of a planning problem in the form of a set $K = \{\Box A_1, ..., \Box A_k\}$ of temporal formulae, where the A_i's are called *control formulae*. The addition of a set of control formulae to the encoding of a problem in general reduces the set of models of the resulting theory, and, consequently, the search space.

As a matter of fact, the addition of correct and effective control knowledge to a problem specification is essential in order for the system PDK to behave well, both in terms of execution time and in terms of plan quality. In fact, since it relies on a depth-first model search mechanism, the solutions PDK can find without control knowledge are quite disappointing in most problems. For instance, the solution found by the planner PDK to the teatime problem with four rooms, with no heuristic knowledge, begins with:

```
1) go_room1_room3
2) go_room3_room1
3) go_room1_hallway
4) go_hallway_room4
5) go_room4_room2
6) getcup_room2
...
```

The following table compares the execution times (in seconds, columns 2 and 3) and plan lengths (i.e. number of actions, columns 4 and 5) in the teatime domain, when PDK is called with control knowledge (columns 2 and 4) and without any heuristic information (columns 3 and 5).

number of rooms	*Execution time*		*Plan length*	
	with control knowledge	*without knowledge*	*with control knowledge*	*without knowledge*
4	0.02	0.02	32	50
8	0.29	0.44	68	168
12	0.65	2.76	104	355
16	1.44	5.03	140	610
20	2.29	7.50	176	933
22	3.22	9.77	194	1120

The rest of this section is devoted to illustrate how control knowledge can be stated in PDDL-K. Beyond control formulae, in 4.1 the main predefined control schemata provided by the language are illustrated.[5] Subsections 4.2 and 4.3 present other extensions of PDDL that are often useful to provide heuristic information. Finally, in 4.4 we observe how the use of control knowledge allows one to define new classes of problems that cannot be handled by classical planning formalisms.

4.1 Control formulae and control schemata

The language PDDL-K allows one to add control formulae explicitly, in a specific section.

> For instance, the constraint that tea must be delivered to a room as soon as possible can be stated as follows:
>
> ```
> (:control
>   (forall (?x - room)
>     (implies (and (ordered ?x) (at ?x) (fullcup))
>              (deliver ?x))))
> ```
>
> As a consequence, the formula
>
> $$\Box\forall x : room(ordered(x) \wedge at(x) \wedge fullcup \rightarrow deliver(x))$$
>
> is added to the encoding

[5] A complete description can be found at `http://pdk.dia.uniroma3.it/`

However, heuristic knowledge is more easily described by means of specific *control schemata*. The use of predefined schemata reduces the risk of introducing occasional errors.

At present, PDDL-K accepts two kinds of control schemata: *fluent-oriented* and *action-oriented*. Fluent-oriented control information is provided in specific sections of the problem description. It consists essentially of knowledge about *bad* and *good situations*: situations that should never be caused by the agent's actions (bad situations) and subgoals that, once achieved, must never – and never need to – be undone (good situations) [37]. Good situations are special cases of the "next-state" control formulae in [4]. The encoding of bad and good situations consists of formulae of the form $\Box(A \rightarrow \bigcirc A)$, where either A is a good situation or $A = \neg B$, where B is a bad situation.

Knowledge about good and bad situations can in most cases be stated more easily in an action oriented way (for instance: "do not perform action a in case it destroys a good situation"). *Action oriented control schemata* are given by means of additional fields in the definition of an operator (besides the `:parameters`, `:precondition` and `:effect` fields). They can be broadly classified into two main categories: *reject* and *select* schemata.

4.1.1 Reject schemata

Reject schemata translate into formulae preventing the addition of some operator instance to the plan, under given conditions. Such formulae are equivalent to formulae of the form

$$\forall x_1 : t_1 ... \forall x_n : t_n \Box(F \rightarrow \neg name(x_1, ..., x_n))$$

where F is a formula, *name* is the name of an operator and $x_1, ..., x_n$ its parameters.

A simple reject schema allows one to specify conditions that should hold when an action is executed in a state. Such conditions are specified in the `:only-if` field of the operator description, in the form:

`(:action` *name* `:parameters` $x_1 – t_1 \ldots, x_k – t_k$
...
`:only-if` $F_1 \ldots F_n$
... `)`

where $F_1, \ldots, F_n$ are formulae, whose free variables are among $x_1 \ldots x_k$. The semantics of the `:only-if` field is

$$\forall x_1 : t_1 \ldots \forall x_k : t_k \Box(name(x_1, \ldots, x_k) \rightarrow F_1 \wedge \ldots \wedge F_n)$$

The fact that the semantics of the `:only-if` field is the same as that of action preconditions is not surprising (see [5]). It is however important that control information is kept separate from knowledge inherent to the domain.

> As an example, in the teatime domain we can specify that an action of the form $go(x, y)$ should not be executed if the robot has nothing to do in y, unless y is the hallway:

```
(:action go :parameters  (?from ?to - location)
     .........
```

```
:only-if (or (= ?to hallway)
             (and (fullcup) (ordered ?to))
             (and (hascup) (not (fullcup))
                  (teamachine ?to))
             (and (not (hascup)) (cupstack ?to))))
```

The LTL encoding of the problem is then added the following formula, suitably simplified according to the background theory:

$$\forall x : location\, \forall y : location\, \Box\,(go(x,y) \rightarrow \\ y = hallway \vee (fullcup \wedge ordered(y)) \vee \\ (hascup \wedge \neg fullcup \wedge teamachine(y)) \vee (\neg hascup \wedge cupstack(y)))$$

Another form of reject schema can be specified in the `:next` field of an operator description, specifying conditions that should hold immediately after having performed the corresponding action.

(`:action` *name* `:parameters` $x_1{-}t_1, \ldots, x_k{-}t_k$
...
`:next` $F_1, \ldots, F_n$
...)

where $F_1, \ldots, F_n$ are formulae, whose free variables are among $x_1, \ldots, x_k$. The semantics of the `:next` field is

$$\forall x_1 : t_1 ... \forall x_k : t_k \Box(name(x_1, ..., x_k) \rightarrow \bigcirc(F_1 \wedge ... \wedge F_n))$$

Typically, the `:next` field is used to force a sequence of actions.

For instance, in the teatime domain the random movement of the robot can be avoided by requiring that, after entering a room, the robot actually does something there:

```
(:action go :parameters  (?from ?to - location)
     ...........
     :next (or (= ?to hallway)
               (deliver ?to)(fillcup ?to)(getcup ?to)))
```

This causes the addition of (the simplification of) the following formula to the encoding:

$$\forall x : location\, \forall y : location\, \Box(go(x,y) \rightarrow \\ \bigcirc(y = hallway \vee deliver(y) \vee fillcup(y) \vee getcup(y)))$$

Note that this is a stronger requirement with respect to the sample `:only-if` constraint above (allowing the robot to enter a room where it could do something and yet exiting it immediately), and however it is much more compact and simple. This is possible thanks to the fact that actions can be mentioned explicitly in operators descriptions.

4.1.2 Select schemata

Select schemata translate into formulae forcing the addition of some operator instance to the plan, under given conditions. Such formulae are equivalent to formulae of the form

$$\forall x_{j_1} : t_{j_1} \ldots \forall x_{j_m} : t_{j_m} \Box(F \rightarrow \exists x_{k_1} : t_{k_1} \ldots \exists x_{k_p} : t_{k_p} name(x_1, ..., x_n))$$

where F is a formula whose free variables are among $x_{j_1}, \ldots, x_{j_m}$, *name* is the name of an operator and $\{x_1, \ldots, x_n\} = \{x_{j_1}, \ldots, x_{j_m}, x_{k_1}, \ldots, x_{k_p}\}$ the set of its parameters.

PDDL-K provides select schemata representing suggestions of the kind: perform a given action as soon as possible, possibly under other conditions. Such schemata are called "`asap`" (As Soon As Possible) schemata. The weaker form of `asap` field expresses the fact that, whenever the preconditions for the applications of the operator on *some* values of its parameters hold, and the `:only-if` conditions also hold for the same values of the parameters, then the operator has to be applied, for *some* values of the parameters:

`(:action` *name* `:parameters` $x_1{-}t_1, ..., x_k{-}t_k$
...
`:asap` $F_1, \ldots, F_n$
... `)`

where $F_1, ..., F_n$ are formulae, whose free variables are among $x_1, ..., x_k$. The semantics of such an `:asap` field is the following formula:

$$\Box(\exists x_1 : t_1 ... \exists x_k : t_k \, (G \wedge F_1 \wedge ... \wedge F_n) \rightarrow \\ \exists x_1 : t_1 ... \exists x_k : t_k \, name(x_1, ..., x_k))$$

Here, G is the conjunction of the formulae in the `:precondition` and `:only-if` fields in the definition of the operator.

> As an example, the definition of the `getcup` operator in the teatime domain can be enriched with:
>
> ```
> (:action getcup :parameters (?x - room)
>      :precondition (at ?x) (cupstack ?x) (not (hascup))
>       ...
>      :asap (exists (?y - location) (ordered ?y)))
> ```
>
> The LTL encoding of the problems is then added the simplification of:
>
> $$\Box(\exists x : location \, (at(x) \wedge cupstack(x) \wedge \neg hascup \wedge \\ \exists y : location \; ordered(y)) \rightarrow \\ \exists x : location \; getcup(x)))$$
>
> I.e.:
>
> $$\Box(at(room2) \wedge \neg hascup \wedge \exists y : location \; ordered(y) \rightarrow getcup(room2))$$

The `:s-asap` ("strong asap") field expresses a stronger form of "`asap`" field: it forces the (concurrent) application of the operator to *all* the values of the parameters to which it can be applied.

> One can see how simple it is now to add the information that tea has to be delivered as soon as possible (already presented at page 13 as an LTL formula to be included in the `:control` section). In fact, it amounts to an unrestricted `:s-asap` field in the definition of `deliver`:
>
> ```
> (:action deliver :parameters (?x - room)
>        ....
>        :s-asap)
> ```

Existential and universal quantification in "as soon as possible" restrictions can also be mixed.

4.2 Reference to the goal and initial state

In the specification of problem-dependent control information, the possibility to refer to the goal to be achieved, as well as to what holds in the initial state can often be useful. 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. A formula of the form *goal* ℓ or *initially* p is either always true or always false; consequently it is treated like static predicates and simplified out in the final encoding. As a consequence, some operator instances can also be eliminated.

> For example, let us consider the problem where a one-arm robot has to move objects from the locations where they initially are to given destinations, stated in the goal. We may then want to specify that an object should be taken away from a given location only if it is not already at its destination place. Moreover, since objects have to be dropped only at their destinations, the robot needs to take up an object only from its initial location. This can be specified by defining the `take` operator as follows:

```
(:action take
    :parameters (?x - object ?y - location)
    :precondition (atRobby ?y) (at ?x ?y)
              (forall (?z - object) (not(holding ?z)))
    :effect (holding ?x)
    :only-if (not (goal(at ?x ?y)))
             (initially (at ?x ?y)))
```

> As a consequence, all the instances of the `take` operator whose arguments do not satisfy the `:only-if` requirement are eliminated from the language of the encoding. For instance, if the destination of object $obj1$ is $room2$, the atom $take(obj1, room2)$ is replaced by $false$ in the encoding.

4.3 Definitions

In some cases relevant domain dependent information can be expressed in a general form only by recourse to new defined predicates. The language PDDL-K provides this possibility. In the encoding, every instance of a defined predicate is replaced by (the value of) its definition. Recursion is also allowed in definitions.

> For instance, in the description of the well known *Blocks World*,[6] "good towers" can be defined as follows:

```
(:define goodTower (?x - block)
    (or (and (goal(onTable ?x)) (onTable ?x))
        (forsome (?y - block) (and (goal(on ?x ?y))
                                   (on ?x ?y)
                                   (goodTower ?y)))))
```

> In a problem where the goal is to build a tower where block B is on block A which, in turn, is on the table, every occurrence of $goodTower(B)$ is replaced, in the encoding, by $on(B, A) \wedge onTable(A)$.

[6] In the blocks domain, a one-arm robot has to re-arrange a set of blocks on a table, making towers with them.

4.4 Control knowledge and new classes of problems

As a final observation, it is worth pointing out that the possibility to include heuristic information in a problem description enlarges the classes of problems that can be dealt with. For instance, there are problems whose solutions have to respect some requirements on the order in which actions are executed, that could not be stated in classical planning languages. As an example, we can consider a refined version of the classical *briefcase domain*, where a robot has to take some objects from some places to others, by use of a briefcase. In the new version of the problem, objects are of two distinguished types: normal and perishable ones, that should not be kept in the briefcase longer than necessary. The robot should therefore take perishable objects last, still trying to minimise its own movements. This means that a normal object can be put into the briefcase that already contains some perishables only if it is in the same location where another perishable object is to be taken. This additional restriction, that cannot be represented in classical planning languages, can be stated in PDDL-K by saying that $take(x, y)$ (take object x from location y and put it into the briefcase) can be executed `:only-if` either x is perishable or there are no perishable objects in the briefcase or x is put in the briefcase at the same time as a perishable object:

$$is\_perishable(x) \vee \forall z : perishable\ \neg in\_briefcase(z) \\ \vee \exists z : perishable\ take(z, y)$$

See `http://pdk.dia.uniroma3.it` for the complete specification of the problem.

5 Meta-level tools for off-line check

A problem specification may suffer from different forms of incorrectness, consequently making the planner incomplete. Moreover, the addition of control knowledge sometimes risks making the search harder, instead of helping, because of the overhead caused by the processing of the control theory itself (see [27]). It is therefore important that the encoding is kept as compact as possible, and redundancies are recognised and carefully evaluated. Representing the whole planning problem in a logical language provides the possibility to perform some important off-line consistency and redundancy checks with little extra effort.

A special utility in the system PDK allows one to check the specification and warn the user, with respect to some metalevel properties. Such tools analyse the set of formulae obtained from the specification of the problem, always excluding the description of the goal. The description of the properties, that follows, will be illustrated in some points with examples from a simple planning problem.

> A one-arm robot has to move a ball and a book from room A to room B. The problem can be specified as follows.

```
(:types object location)
(:objects A B - location
          ball book - object)
(:predicates (at ?x - object ?y - location)
          (atRobby ?x - location) (free)
```

```
          (carry ?x - object))
(:init (atRobby A) (forall (?x - object) (at ?x A))
       (free))
(:goal (forall (?x - object)(at ?x B)))
(:action pick :parameters (?x - object ?y - location)
        :precondition (atRobby ?y)(at ?x ?y)(free)
        :effect (not (at ?x ?y))(not (free))(carry ?x))
(:action drop :parameters (?x - object ?y - location)
        :precondition (atRobby ?y)(carry ?x)
        :effect (at ?x ?y) (free) (not (carry ?x)))
(:action go :parameters (?x ?y - location)
        :precondition (atRobby ?x)
        :effect (atRobby ?y)(not (atRobby ?x)))
```

Consistency of the kernel of the specification: the system tests the logical consistency of the set of LTL formulae obtained from the initial state, the background theory and the kernel of the operators description, excluding control knowledge. This is a minimal requirement for the specification to be sound.

Consistency of control knowledge: the system checks whether the set of formulae obtained from the specification of control knowledge can be safely added to the kernel of the specification.

It is reasonable to require that the robot should pick up an object whenever possible and when it is not in its goal destination. But, if a "strong `asap`" requirement is added to the specification of the *pick* action:

```
(:action pick
        :parameters (?x - object ?y - location)
        ........
        :s-asap (not (goal (at ?x ?y))))
```

then no plan is found and the system detects an inconsistency in control knowledge. In fact the encoding of the `:s-asap` field is:

$$\Box(atRobby\_A \wedge at\_ball\_A \wedge free \rightarrow pick\_ball\_A)$$
$$\Box(atRobby\_A \wedge at\_book\_A \wedge free \rightarrow pick\_book\_A)$$

Since in the initial state at_ball_A, at_book_A, $atRobby\_A$ and $free$ are all true, then $pick\_ball\_A$ and $pick\_book\_A$ should also hold in the initial state. But the incompatibility axioms include

$$\Box(pick\_ball\_A \rightarrow \neg pick\_book\_A)$$

So, the addition of control knowledge causes the encoding to be contradictory, as detected by this metalevel tool.

Action executability: each operator instance (which has not been already filtered out because of its direct inconsistency with the background theory) is considered, in turn, in order to check whether it can ever be applied. In order to do this, the formula $\Diamond A$, where A is the considered action, is added to the set of formulae consisting of the kernel of the specification and control knowledge (i.e. deriving from the initial state, background theory, operator descriptions and control formulae), and the resulting set of formulae is tested for satisfiability.

Let us modify our simple problem as follows: there is a third location, *home*, where the robot is initially and has to go back at the end:

```
(:objects A B home - location
          ball book - object)
(:init (atRobby home) (free)
       (forall (?x - object) (at ?x A)))
(:goal (atRobby home)
       (forall (?x - object)(at ?x B)))
```

Moreover, the following control fields are added to the operator descriptions:

```
(:action drop
     :parameters (?x - object ?y - location)
     ......
     :only-if (goal (at ?x ?y)))
(:action go
     :parameters (?x ?y - location)
     ......
     :next (exists (?z - object)
                (or (drop ?z ?y)(pick ?z ?y))))
```

This problem has no solution. The reason is that no action of the form $go(x, home)$ can ever be executed. In fact, no object can be dropped at home, and nothing can therefore be picked up there (there is nothing at home initially, and there will never be).

Another frequent reason for plan search failure is that some important action has been removed from the very beginning, because its precondition or effects are inconsistent. The system allows one also to obtain a list of all the eliminated actions, that can therefore be examined to check whether they are actually unnecessary.

Let us consider the same problem as above, where also the *pick* action is added a control field:

```
(:action pick
     :parameters (?x - object ?y - location)
     ......
     :only-if (initially (at ?x ?y)))
```

Again, this problem has no solution because the robot cannot go back home. Differently from before, however, all actions of the form $go(x, home)$ are contradictory and therefore are eliminated, because all actions of the form $pick(y, home)$ and $drop(y, home)$ are also removed. This fact can be recognised by examining the list of eliminated actions.

Redundancy check: each control formula is tested for provability from the rest of the specification. In case of a positive answer, such a formula is pointed out as redundant. Note that redundancies are not always to be avoided. In fact, there are cases where plan search becomes faster, notwithstanding the overhead due to a larger encoding. It is however important that they are pointed out, so that the user can carefully evaluate each case.

Let us consider our simple example with the only two locations A and B, described at page 18, with the addition of the following control fields in the operators description:

```
(:action pick
     :parameters (?b - object ?r - location)
     ........
     :only-if (not (goal (at ?b ?r)))
     :asap )
(:action drop
     :parameters (?b - object ?r - location)
     ........
     :only-if (goal (at ?b ?r))
     :s-asap )
(:action move
     :parameters (?from ?to - location)
     ........
     :next (exists (?x - object)
              (or (pick ?x ?to)(drop ?x ?to)))))
```

Then the system's metalevel tool will point out that the encoding of the `:s-asap` field of the operator *drop* is redundant (for all its instances).

6 Experiments and Comparisons

In this section we are going to compare PDK and its language with other existing planners. The comparison is made with respect to three main features: expressive power, execution time and plan quality.

6.1 Expressive power

When describing a planning domain, the identification and statement of correct and effective control knowledge is often a subtle and difficult task. Therefore, when comparing planners allowing one to specify problem dependent information, it is important to evaluate the ease of use of the underlying planning languages. In this section, we briefly compare PDDL-K with the languages accepted by other planning systems.

The planners written in GOLOG [37][7] do not accept a true planning language, but the specification of each problem is rather a piece of Prolog code, that defines action preconditions and successor state axioms, in the Situation Calculus style. Control knowledge can be stated in terms of "bad situations" (which are used to control the planners in a way similar to our reject schemata): "it is in bad situations where all the planner's intelligence resides" [37]. The equivalent of our select schemata (called "opportunistic rules" by Reiter) are to be stated in negative terms. For instance, in the teatime domain, in order to say that tea should be served as soon as possible, a bad situation has to be defined: the situation generated by any action destroying the existing preconditions for delivering tea (in this domain, going away). As a further example, let us consider the blocks world and the following "opportunistic rule": *if an action can*

[7]The implementation of GOLOG in ECLIPSE Prolog and the planners are available at `http://www.cs.toronto.edu/cogrobo/kia/`.

create a good tower, don't do a bad-tower-creating moveToTable action. Such a rule is encoded as follows [37]:

```
badSituation(do(moveToTable(X),S)) :-
            not goodTower(X,do(moveToTable(X),S)),
            existsActionThatCreatesGoodTower(S).

existsActionThatCreatesGoodTower(S) :-
            (A = move(Y,X) ; A = moveToTable(Y)),
            poss(A,S), goodTower(Y,do(A,S)).
```

It is apparent that, at present, writing (and debugging) a planning domain in GOLOG requires good Prolog programming skills.

The same remark applies to the planner based on Answer Set Planning,[8] described by [40]. Planning domains and control knowledge are in fact described in AnsProlog* [6], a logic programming language with answer set semantics. Although also in this case writing a planning domain requires specific programming skills, new forms of control knowledge can be specified, including knowledge inspired by the partial-ordering constructs used in *Hierarchical Task Networks* [16, 44].

Among the logic-based planners, SATPLAN, that has already been described in Section 1, is the approach that shares more features with PDK. [27] examine different forms of control knowledge that can be encoded in the "planning as satisfiability" approach, and [22] report experiments with an extension of PDDL that allows additional control constraints. They observe that the empirical results show that the addition of control knowledge speeds the system up to an order of magnitude. The style and the encoding of the language are similar to PDDL-K, but, to our knowledge, there is no available implementation of SATPLAN accepting this extension of PDDL.

Also planners based on specialized algorithms can benefit from the use of a logical language to express control knowledge. TLPLAN [4] and TALplanner [15] are based on specialized forward-chaining search algorithms, exploring a tree-like structure of states, where each path is expanded only if control specifications, encoded as Linear Temporal Logic formulae, are satisfied in the path. In particular, TLPLAN is known as the best planner exploiting control knowledge. Its behaviour has been very effective in past planning competitions and, in terms of execution time, it largely outperforms the perfomances of PDK. Nevertheless, giving an effective description of a planning domain in the language of TLPLAN (that also extends PDDL) requires a deep analisys of the domain and a considerable effort. In fact, in order to obtain a good behaviour of the planner, new features have to be extracted from the original domain and the description has to be consequently enriched with definitions of predicates and functions, often in the style of a true programming language. Furthermore, the specification of correct and effective control knowledge is often quite cumbersome, because all statements must be made in terms of fluents. For instance, the fact that, after going to a place y, the agent has to do something at y has to be paraphrased into: "if the agent is at a place x at time t and it is at $y \neq x$ at time $t+1$, then at time $t+2$ it stays at y".

[8] Answer Set Planning is a planning approach that translates a planning problem into a problem of model finding in logic programming with answer set semantics. It was originally proposed by [42] and [31].

6.2 Time efficiency and plan quality

In this section we compare and discuss the performances of PDK, in terms of time efficiency and plan quality, w.r.t. other existing planners. The experiments were run on a PC with P4, 3.00GHz and 1GB RAM, running under Linux. Beyond the planners written in GOLOG and SATPLAN, we have considered two planners that recently showed best performances in the International Planning Competitions, YAHSP and LPG. LPG[9] is a planner based on a stochastic local search. YAHSP[10] is based on a complete best-first search algorithm using a look-ahead strategy.

The experiments we are going to report were carried out on the domains that have already been briefly described in this paper, i.e. the teatime domain with one robot, the blocks world and the briefcase domain.[11]

Of course, SATPLAN, YAHSP and LPG ran with no heuristic knowledge, since they do not support control information. The GOLOG planners and PDK ran with substantially equivalent control knowledge. Specifically:

The blocks world: do not destroy "good towers"; do not create a "bad tower" by moving a block onto another one; create "good towers" as soon as possible.

The briefcase domain: drop an object only at its destination and only when all the other objects having the same destination are either at place or in the briefcase (so that every location is visited at most once); don't take an object from its destination; take and drop objects (when allowed to) as soon as possible; don't go to a place then exit it immediately (i.e. don't go wandering).

The teatime domain: don't enter a room then exit it immediatly (the only place where the robot needs to pass without doing anything is the hallway, that is not a "room"); get a cup as soon as possible, when there are still rooms to be served.

The results on the execution times (in seconds) are reported in the following tables. The execution times of PDK obviously include the encoding times. Similarly, SATPLAN, YAHSP and LPG ran on problem specifications written in PDDL. For the GOLOG planners, on the contrary, problems had to be encoded by hand; the GOLOG planners have always been called with a sufficiently large depth limit. In all experiments, the computation time has been limited to 300 secs (a dash in the tables means a timeout). The tables have two columns related to the GOLOG planners: the first (Dfs) contains data on the depth-limited planner, the second (Bfs) on the breadth-first planner. LPG and YAHSP, that can be called with different options, have been executed in the modality that finds better quality solutions. Since LPG is a non deterministic

[9] LPG is available at `http://zeus.ing.unibs.it/lpg/`

[10] YAHSP ("Yet Another Heuristic Search Planner") is available at `http://www.cril.univ-artois.fr/~vidal/yahsp.en.html`

[11] The domains used in the experiments represent classical problems which have been given different formalizations in the literature or in existing planners' repositories. Some versions of all the three domains can be found for instance in the repository of the European Planning Network of Excellence (PLANET) at `http://scom.hud.ac.uk/planet/repository/`, as well as in the planning repository of the University of Freiburg at `http://www.informatik.uni-freiburg.de/~koehler/ipp/`

planner, each measure in its column represents the average of 50 executions on the same problem instance.

The blocks world: execution times

problem	*Pdk*	GOLOG Dfs	GOLOG Bfs	SATPLAN	YAHSP	LPG
Prob4-1	0.01	0.07	0.08	0.05	0.00	0.25
Prob6-1	0.03	0.07	0.08	0.34	0.00	0.26
Prob8-1	0.26	0.07	0.23	23.71	0.03	0.87
Prob10-1	0.77	0.08	11.66	–	0.24	2.75
Prob12-1	3.04	0.11	23.39	–	0.25	8.34
Prob14-1	3.14	0.13	–	–	1.37	14.41
Prob16-1	6.88	0.22	–	–	2.16	37.65
Prob18-1	41.47	0.39	–	–	16.78	98.67

The briefcase domain: execution times

problem	*Pdk*	GOLOG Dfs	GOLOG Bfs	SATPLAN	YAHSP	LPG
3×3	0.01	0.07	0.11	0.21	0.00	0.21
4×4	0.07	0.07	0.41	7.29	0.00	0.21
6×3	0.03	0.08	29.85	0.15	0.00	0.20
6×6	0.36	0.07	273.72	–	0.00	0.26
8×4	0.22	0.08	–	3.95	0.00	0.28
10×10	2.11	0.09	–	–	0.03	0.38
12×10	2.32	0.10	–	–	0.04	0.47
14×10	3.81	0.10	–	–	0.06	0.62
16×10	3.41	0.11	–	–	0.08	0.76

The teatime domain: execution times

problem	*Pdk*	GOLOG Dfs	GOLOG Bfs	SATPLAN	YAHSP	LPG
2	0.00	0.07	0.09	0.13	0.00	0.21
3	0.01	0.08	0.25	13.88	0.00	0.22
4	0.02	0.08	2.15	–	0.00	0.24
6	0.14	0.09	265.15	–	0.04	0.31
8	0.29	0.11	–	–	0.37	0.39
12	0.65	0.23	–	–	18.18	0.70
16	1.44	0.54	–	–	–	1.29
20	2.29	1.19	–	–	–	2.22
22	3.22	1.79	–	–	–	2.84

In the tables, problem names in the blocks world have the form Prob$n$-k, where n is the number of blocks involved in the problem.[12] Each problem in the briefcase domain is named $n \times m$, where n is the number of objects and m the number of rooms. The numbers in the *problem* column of the teatime domain refer to the number of rooms.

[12]The blocks world problems are from the repository of the Second IPC: `http://www.cs.colostate.edu/meps/repository/aips2000.html`

The results reported above show that, in terms of execution times, PDK is comparable with YAHSP and LPG. It is generally faster than SATPLAN and the breadth-first GOLOG planner, since the performances of these two planners, acting by iterative deepening, rapidly degrade when the length of the solution plan increases. For instance, SATPLAN takes almost 700 seconds to solve the teatime problem with 4 rooms. The depth-limited GOLOG planner is faster than PDK, but, if the depth limit is not large enough, it employs a large amount of time to discover that there is no solution. For instance, GOLOG needs nearly 35 seconds to realize that there is no solution to the teatime problem with 6 rooms in the limit of 45 actions.

In interpreting the results shown above one has to remember that only PDK and the GOLOG planners use heuristic knowledge and the domains used in the comparison can all be given meaningful information (in a relatively simple form).

With respect to plan quality, in classical planning problems, where actions are assumed to have the same cost, a reasonable measure is plan length. Plan length, in turn, can be measured either in terms of number of actions, or, if parallel actions are allowed, in terms of number of "layers" (sets of actions that can be executed in parallel). SATPLAN, for instance, is optimal in terms of number of layers, but not necessarily in terms of number of actions. The breadth-first GOLOG planner, on the contrary, is optimal in terms of number of actions but does not allow for parallelism. Both measures are arbitrary, in general: there are problems where it is important to reach the goal as soon as possible, and problems where action execution costs cannot be ignored. In the tables that follow, plan length is measured in terms of number of actions, not to penalise planners than cannot build parallel plans (the GOLOG planners, YAHSP and LPG). The real numbers in the LPG column are the average of plan lengths resulting from 50 executions on the same problem.

THE BLOCKS WORLD: NUMBER OF ACTIONS

	Pdk	GOLOG		SATPLAN	YAHSP	LPG
problem		Dfs	Bfs			
Prob4-1	5	5	5	5	5	5.34
Prob6-1	5	5	5	7	5	5.62
Prob8-1	10	10	10	15	10	10.44
Prob10-1	17	17	16	28	19	18.26
Prob12-1	21	21	17	–	20	22.86
Prob14-1	19	22	–	–	24	23.88
Prob16-1	27	28	–	–	34	36.68
Prob18-1	32	32	–	–	42	43.10

The briefcase domain: number of actions

problem	*Pdk*	GOLOG Dfs	GOLOG Bfs	SATPLAN	YAHSP	LPG
3×3	10	10	10	10	12	11.16
4×4	13	13	13	13	16	15.12
6×3	16	16	16	16	18	18.36
6×6	19	19	19	–	24	22.94
8×4	21	21	–	21	24	24.50
10×10	31	31	–	–	40	38.22
12×10	35	35	–	–	44	42.30
14×10	39	39	–	–	48	46.40
16×10	43	43	–	–	52	51.52

The teatime domain: number of actions

problem	*Pdk*	GOLOG Dfs	GOLOG Bfs	SATPLAN	YAHSP	LPG
2	16	16	14	14	14	15.00
3	24	23	22	22	22	22.40
4	32	32	30	30	30	30.40
6	50	50	48	–	48	48.58
8	68	68	–	–	66	67.30
12	104	104	–	–	102	105.88
16	140	140	–	–	–	145.10
20	176	176	–	–	–	185.14
22	194	194	–	–	–	204.36

From the tables, it appears that Pdk, both GOLOG planners and in most cases also SATPLAN find plans close to the optimal ones. With respect to the breadth-first GOLOG planner and SATPLAN, this is due to their iterative deepening mechanism. The results regarding the depth-first GOLOG planner and Pdk are due to the addition of suitable control knowledge.

With respect to YAHSP and LPG, we can note that plan lengths are comparable with Pdk in the teatime domain, where also their execution times are closer, while the plans found by YAHSP and LPG are about 20% longer than those found by Pdk in the other two domains, and the percentage increases with problem difficulty.

The experimental results reported above show that Pdk often finds the right balance between execution time and plan quality.

7 Concluding remarks

Automatic planning is a computationally hard task. In fact, the general plan-existence problem for STRIPS-like operators is known to be PSPACE-complete [10]. Even if modern domain-independent planners handle far more complex problems than a few years ago there is a widespread trend towards the use of extra problem dependent information, in order to obtain enhanced performances.

However, the detection and correct statement of effective control knowledge is often a difficult task. For this reason, it is important to have a high-level, natural and easy-to-use specification language, requiring no specific programming skills.

To this aim, this paper presents the language PDDL-K, an extension of the ADL-subset of the standard PDDL, that provides the domain expert with a sort of guide, through a set of predefined schemata, and allows one to easily and naturally specify heuristic information. The planner PDK implements the semantics of PDDL-K, given in terms of a translation into LTL formulae. It builds the encoding of the input specification and reduces planning to LTL model search. Thanks to the fact that the whole problem is encoded by a logical theory, PDK also provides a set of tools based on metalevel properties, that can help the user in the debugging task. Experimental results show that the performances of PDK are comparable with other planners, both in terms of execution time and plan quality.

References

[1] J. F. Allen. Temporal reasoning and planning. In J. F. Allen, H. A. Kautz, R. N. Pelavin, and J. D. Tenenberg, editors, *Reasoning about Plans*, pages 1–68. Morgan Kaufmann Publishers, Inc., 1991.

[2] J.F. Allen. Planning as temporal reasoning. In *Proc. of the Second Int. Conf. on Principles of Knowledge Representation and Reasoning (KR-91)*, pages 3–14, 1991.

[3] F. Bacchus and F. Kabanza. Planning for temporally extended goals. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pages 1215–1222. AAAI Press / The MIT Press, 1996.

[4] F. Bacchus and F. Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 16:123–191, 2000.

[5] M. Bacchus, F. Ady. Precondition control. Manuscript, 1999.

[6] C. Baral. *Knowledge representation, reasoning, and declarative problem-solving with answer sets.* Cambridge University Press, 2003.

[7] H. Barringer, M. Fisher, D. Gabbay, G. Gough, and R. Owens. METATEM: a framework for programming in temporal logic. In *Proc. of REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, volume 430 of *LNCS*. Springer, 1989.

[8] H. Barringer, M. Fisher, D. Gabbay, and A. Hunter. Meta-reasoning in executable temporal logic. In *Proc. of the Second Int. Conf. on Principles of Knowledge Representation and Reasoning*, 1991.

[9] W. Bibel. Let's plan it deductively. In *15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, volume 2, pages 1549–1562. Morgan Kauffmann, 1997.

[10] T. Bylander. Complexity results for planning. In *Proc. of the 12th Int. Joint Conf. on Artificial Intelligence (IJCAI-91)*, pages 274–279, 1991.

[11] D. Calvanese, G. De Giacomo, and M. Y. Vardi. Reasoning about actions and planning in LTL action theories. In *Proc. of the 8th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR 2002)*, pages 593–602, 2002.

[12] S. Cerrito and M. Cialdea Mayer. Using linear temporal logic to model and solve planning problems. In F. Giunghiglia, editor, *Proceedings of the 8th International Conference on Artificial Intelligence: Methodology, Systems, Applications (AIMSA'98)*, pages 141–152. Springer, 1998.

[13] A. Cesta and A. Oddi. DDL.1: a formal description of a constraint representaton language for physical domains. In M. Ghallab and A. Milani, editors, *New Direction in AI Planning*, pages 341–352. IOS Press, 1996.

[14] M. Cialdea Mayer, A. Orlandini, G. Balestreri, and C. Limongelli. A planner fully based on linear time logic. In S. Chien, S. Kambhampati, and C.A. Knoblock, editors, *Proc. of the 5th Int. Conf. on Artificial Intelligence Planning and Scheduling (AIPS-2000)*, pages 347–354. AAAI Press, 2000.

[15] P. Doherty and J. Kvarnström. TALplanner: A temporal logic based planner. *AI Magazine*, 22:95–102, 2001.

[16] K. Erol, J. Hendler, and D. S. Nau. Complexity results for HTN planning. In *Proceedings of AAAI-94*, 1994.

[17] R. E. Fikes and N.J. Nilsson. STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3–4):189–208, 1971.

[18] A. Finzi, F. Pirri, and R. Reiter. Open world planning in the Situation Calculus. In *Proceedings of the 7th Conference on Artificial Intelligence (AAAI-00) and of the 12th Conference on Innovative Applications of Artificial Intelligence (IAAI-00)*, pages 754–760. AAAI Press, 2000.

[19] M. Fisher and R. Owens. An introduction to executable modal and temporal logics. In M. Fisher and R. Owens, editors, *Executable modal and temporal logics (Proc. of the IJCAI'93 Workshop)*, volume 897 of *LNAI*, pages 1–20. Springer, 1995.

[20] 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.

[21] H. Geffner. Perspectives on artificial intelligence planning. In *Proceedings Eighteenth National Conference on Artificial Intelligence (AAAI-2002)*, pages 1013–1023, 2002.

[22] Yi-Cheng Huang, Bart Selman, and Henry A. Kautz. Control knowledge in planning: Benefits and tradeoffs. In *AAAI/IAAI*, pages 511–517, 1999.

[23] G. L. J. M. Janssen. *Logics for Digital Circuit Verification. Theory, Algorithms and Applications*. CIP-DATA Library Technische Universiteit Eindhoven, 1999.

[24] H. Kautz, D. McAllester, and B. Selman. Encoding plans in propositional logic. In *Proc. of the 5th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'96)*, pages 374–384, 1996.

[25] H. Kautz and B. Selman. Planning as satisfiability. In B. Neumann, editor, *10th European Conference on Artificial Intelligence (ECAI-92)*, pages 360–363. Wiley & Sons, 1992.

[26] H. Kautz and B. Selman. BLACKBOX: A new approach to the application of theorem proving to problem solving. In *Working notes of the AIPS-98 Workshop on Planning as Combinatorial Search*, pages 58–60, 1998.

[27] H. Kautz and B. Selman. The role of domain-specific knowledge in the planning as satisfiability framework. In *Proc. of the Fourth Int. Conf. on Artificial Intelligence Planning Systems (AIPS-98)*, 1998.

[28] H. Kautz and B. Selman. Unifying sat-based and graph-based planning. In *IJCAI-99, Stockholm*, 1999.

[29] J. Koehler and R. Treinen. Constraint deduction in an interval-based temporal logic. In M. Fisher and R. Owens, editors, *Executable Modal and Temporal Logics, (Proc. of the IJCAI'93 Workshop)*, volume 897 of *LNAI*, pages 103–117. Springer, 1995.

[30] H. Levesque. What is planning in the presence of sensing? In *Proc. of the 13th National Conference on Artificial Intelligence, AAAI-96*, pages 1139–1146. AAAI Press, 1996.

[31] V. Lifschitz. Answer set planning. In *Proceedings of International Conference on Logic Programming (ICLP-1999)*, pages 23–37, 1999.

[32] M. Masseron, C. Tollu, and J. Vauzeilles. Generating plans in linear logic: I. Actions as proofs. *Theoretical Computer Science*, 113:349–370, 1993.

[33] J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of502 artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, pages 463–502. Edimburgh University Press, 1969.

[34] E. Pednault. ADL: exploiting the middle ground between STRIPS and the situation calculus. In *Proc. of the Int. Conf. on Principles of Knowledge Representation and Reasoning (KR-89)*, pages 324–332, 1989.

[35] M. Pistore and P. Traverso. Planning as model checking for extended goals in non-deterministic domains. In *Proc. IJCAI'01*. AAAI Press, 2001.

[36] R. Reiter. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In V. Lifschitz, editor, *Artificial Intelligence and mathematical theory of computation: Papers in honor of John McCarthy*, pages 359–380. Academic Press, 1991.

[37] R. Reiter. *Knowledge in Action: logical foundations for specifying and implementing dynamical systems*. MIT Press, 2001.

[38] J. Rintanen. Constructing conditional plans by a theorem-prover. *Journal of Artificial Intellingence Research*, 10:323–352, 1999.

[39] S. J. Rosenschein. Plan synthesis: a logical perspective. In *Proc. of IJCAI-81*, pages 331–337, 1981.

[40] T. Son, C. Baral, and S. McIlraith. Domain-dependent knowledge in answer set programming. *ACM Transactions on Computational Logic*, 2005. To appear.

[41] B. Stephan and S. Biundo. Deduction based refinement planning. In B. Drabble, editor, *Proceedings of the 3rd International Conference on Artificial Intelligence Planning Systems (AIPS-96)*, pages 213–220. AAAI Press, 1996.

[42] V. S. Subrahmanian and C. Zaniolo. Relating stable models and AI planning domains. In *Proceedings of International Conference on Logic Programming (ICLP-1995)*, pages 233–247, 1995.

[43] P. Wolper. The tableau method for temporal logic: an overview. *Logique et Analyse*, 28:119–152, 1985.

[44] Q. Yang. Formalizing planning knowledge for hierarchical planning. *Computational Intelligence*, 6:12–24, 1990.