A Description Language for Planning with Domain and Control Knowledge

MARTA CIALDEA, CARLA LIMONGELLI

RT-DIA-68-2001 Luglio 2001

Università di Roma Tre,
Dipartimento di Informatica e Automazione
Via della Vasca Navale, 79
00146 Roma, Italy.

This work was partially supported by ASI (Agenzia Spaziale Italiana)
ABSTRACT

Recently, considerable attention has been devoted to the automatic extraction of useful information from the description of a planning domain and the exploitation of declaratively specified heuristic knowledge. Control knowledge is often formulated in some logical language, whose correct use, however, is not always easy for the domain expert. In this paper we address the problem of making the statement of such extra knowledge easier and reducing the risk of occasional errors. With this aim, we define an algorithm-independent language, LPK (Language for Planning with domain and control Knowledge), allowing the user to describe a planning domain, equipped with different forms of domain-specific knowledge. In particular, heuristic information is declaratively represented by means of general control schemata. The language also allows the representation of problems that do not naturally fit into the model of a single final goal and instantaneous actions.

The semantics of LPK is given by means of a translation into temporal logic formulae: the specification of a planning domain is completely represented by a logical theory. Planning can thus be reduced to model search and the language LPK can be directly “executed” by means of a straightforward use of its semantics. More important, the reduction makes it possible to implement useful tools aiming at pointing out possible inconsistencies or redundancies in the specification.
1 Essentials of LPK

In this section we introduce the essential components of a problem description (constituting what is called the kernel of the specification) and describe the fundamental principles informing the semantics of LPK, in terms of its translation into a set of propositional LTL formulae (a more detailed presentation can be found in [7, 8]). The model of time underlying LTL is a countably infinite sequence of states (or time points), that can be identified with \( \mathbb{N} \). An interpretation \( \mathcal{M} \) is a function mapping each time point \( k \) to the set of propositional letters true at \( k \). Truth in a model is truth in its initial state. In this work we use only the unary future time operators: \( \square A \) (“always \( A \)”’) is true at \( k \) iff \( A \) is true at any time point \( j \geq k \), \( \diamond A \) (“eventually \( A \)”’) is true at \( k \) iff \( A \) is true at some time point \( j \geq k \), and \( \circ A \) (“next \( A \)”’) is true at \( k \) iff \( A \) is true at \( k + 1 \). This restriction of the target language of the translation makes it easier to provide analogous translations into other languages.

A temporal interpretation describes how the world evolves in time, hence a set of formulae constrains the behaviour of the world and can be taken as the specification of a planning domain. If \( \Sigma \) is the LPK-specification of a planning problem \( \Pi \), and \( |\Sigma| \) is the set of LTL formulae constituting the semantics of \( \Sigma \), then solving the problem \( \Pi \) can be reduced to finding a model of \( |\Sigma| \). The kernel of the specification is substantially what is necessary to characterize the problem (excluding, for instance, control knowledge): it is a subset \( \Sigma^* \subseteq \Sigma \) such that \( |\Sigma^*| \) is a correct and complete representation of \( \Pi \), i.e. any model of \( |\Sigma^*| \) represents a solution to \( \Pi \) and, vice-versa, any solution of \( \Pi \) is represented by some model of \( |\Sigma^*| \).

LPK uses a first order many-sorted temporal language, where each domain is finite and fixed. The kernel of the specification defines:

1. The object domains. For each domain (or type), a finite set of constants is specified, naming the objects of the domain.

2. The fluents involved in the problem, and the “static predicates”, i.e. predicates whose meaning does not change over time. A type is associated to each fluent and static predicate.

3. A specific section contains knowledge about static predicates (assuming a closed world, what is not derivable from such a background theory, is false). In general, the background theory contains formulae without temporal operators. Such formulae are meant to be true throughout time, i.e. they represent facts that do not vary over time (state invariants). Also, additional relevant predicates can be defined in this section (like in [3]).

4. The initial state and goal. The specification \( S_0 \) of the initial state, together with the background theory \( K \), is assumed to be complete with respect to fluents, i.e. for any fluent \( R \), either \( S_0 \cup K \models R \) or \( S_0 \cup K \models \neg R \).
5. The operators. The kernel description of each operator specifies its name, parameters (with associated type), and pre- and post-conditions. Preconditions can have any form. Conditional and universally quantified effects are also allowed.

In ADL-like languages the specification of static predicates is included in the description of the initial state and the fact that their truth value does not vary over time is not explicit. Rather, it must be derived analysing the initial state and the operators. The language LPK, on the contrary, allows the explicit specification (in the background theory) of any domain invariant the user is aware of.

A planning problem can be represented by a set of LTL formulae according to different encodings, as shown in [5]. The semantics of LPK consists of a simple form of progression encoding, that recalls the encoding of planning problems in the situation calculus [14] and the linear encoding of [11]. Such an encoding schema is provably correct and complete [5].

Each ground atom in the language of the LPK-specification (fluent, static predicate or operator instance) is mapped to a propositional letter. We shall get on using a first order syntax in order to enhance readability, where (typed) quantifiers are abbreviations of finite conjunctions or disjunctions. LPK 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 [3]. 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 since there are no non-arithmetical functional symbols), then \( t = u \) holds iff \( t \) and \( u \) are identical.

The system PADOK accepts the LPK-specification of a problem, translates it into a set of propositional LTL formulae, according to its semantics, and searches for a model of such a set of formulae. In the translation, the background theory is used to filter operator instances, by elimination of those whose preconditions or effects are either self-contradictory or inconsistent with the theory. In other terms, by use of the background theory, operator instances that would be ruled out by static reject rules (in the terminology of [10]) are eliminated. Moreover, the background theory is used for simplification purposes: in particular, the simplified encoding of a problem contains no occurrences of static predicates, since each of them is replaced by “true” or “false”, according to the background knowledge.

The fact that “actions” (operator instances) 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 door-latch problem, borrowed from [1], that is not easily and naturally representable in ADL-like 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 \textit{open} (the door is open) and the two operators \textit{pull\_door} and \textit{hold\_lock} (with no parameters). The following is a correct LPK specification, accepted by PADOW, which correctly “opens the door” by executing the two actions together (the double arrow \(\Rightarrow\) is used for conditional effects):

\[
\begin{align*}
\text{fluent} &= \textit{open}; \\
\text{op pull\_door} &:\textit{open} \\
\text{op hold\_lock} &:\textit{open} \\
\text{POST hold\_lock} &\Rightarrow \textit{open}; \\
\text{POST pull\_door} &\Rightarrow \textit{open};
\end{align*}
\]

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.

2 Basic Categories of Control Knowledge

Control knowledge can be given the form of a high level, non-deterministic program (in the style of Golog [13]), that is to be “executed” starting at the initial state. A different approach, that is adopted here (as well as, for instance, in [3, 12]), identifies a set of formulae \(A_1, \ldots, A_k\) that should hold at any state in time. In this view, the LTL encoding of a planning problem represents control knowledge by means of a set \(K = \{\square A_1, \ldots, \square A_k\}\), where the \(A_i\)'s are called \textit{control formulae}. The addition of a set of control formulae in general reduces the set of models of the resulting theory, and, consequently, the search space.

A main issue about \(K\) is that its addition to the set of formulae constituting the semantics of the kernel of the specification can cause incompleteness (see Section 4). It is important, therefore, to identify some guidelines for the formulation of control knowledge in arbitrary planning domains. Hence, although LPK also accepts control formulae in a specific section, heuristic knowledge is more easily described by means of specific \textit{control schemata}, that are described in the rest of this section. The use of predefined schemata reduces the risk of introducing occasional errors. The set of schemata allowed by LPK has been identified by analysing different domains and examples given in the literature.

In the sequel, the following sample domains are used: the briefcase domain (different objects distributed over several locations must be re-located, making use of a briefcase to move them; when an object is in the briefcase, it moves with it) and the gripper domain (a robot with two grippers has to move a given number of balls from room A to room B).

In the specification of control information, the possibility to refer to the goal to be achieved is often useful. To this aim, the syntax of LPK formulae is extended
by means of the unary modal operator Goal: the intended interpretation of a formula Goal(A) is that A is a goal. The operator can be given a semantics like in [3]. At present, however, it is allowed a restricted use: the operator can only be applied to classical, non-disjunctive formulae and is treated “syntactically”, i.e. Goal(A) means that A is one of the specified goals of the planning problem. A formula of the form Goal(A) is either always true or always false. Consequently, Goal(A) is treated like static predicates and simplified out in the final encoding.

At present, LPK accepts two kinds of control schemata: fluent-oriented and action-oriented. Fluent-oriented control information is provided in specific sections of the problem description, while action-oriented control knowledge is given in special fields of the operator descriptions. The set of schemata described below is not meant to avoid redundancies. On the contrary, substantially equivalent control formulae can be added by means of different control schemata, at the choice of the user.

2.1 Fluent-Oriented Control Schemata

Bad situations are situations that should never be caused by the agent’s actions [13]. A special section in the domain description accepts formulae representing bad situations. The semantics of each bad situation A is the formula:

$$\Box (\neg A \rightarrow \Diamond \neg A)$$

expressing the fact that, if A is false, it must stay false.

Good situations are subgoals that, once achieved, must never (and never need to) be undone [13]. They are special cases of the “next-state” control formulae in [3]. A special section in the domain description accepts formulae representing good situations. The semantics of each good situation A is the formula:

$$\Box (A \rightarrow \Diamond A)$$

expressing the fact that, once A becomes true, it stays true forever.

For instance, in the briefcase domain, every instance of the formula Goal(at(x,y)) → at(x,y) (an object being at its destination) represents a good situation:

good_situations = \forall x : obj \forall y : loc GoodSit (Goal(at(x,y)) \rightarrow at(x,y));

If this field is included in the problem description, it is translated (and simplified) into a set of formulae of the form

$$\Box (at(o, dest) \rightarrow \Diamond at(o, dest))$$

where o is any object and dest its destination, i.e. the location such that Goal(at(o, dest)) holds. Note that in the good_situations field, the operator
*GoodSit* is used, in order to distinguish meta-level quantification ("for all object \( x \) and location \( y \), \( \text{Goal}(at(x,y)) \rightarrow at(x,y) \) is a good situation") from object-level quantification ("\( \forall x \colon \text{obj} \forall y \colon \text{loc} (\text{Goal}(at(x,y)) \rightarrow at(x,y)) \) is a good situation").

### 2.2 Action-Oriented Control Schemata

The operator-oriented control formulae considered here can be broadly classified into two main categories (the terminology is taken from [10]):

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

   \[
   \Box \forall x_1 : t_1, \ldots, \forall x_n : t_n (F \rightarrow \neg \text{name}(x_1, \ldots, x_n))
   \]

   where \( F \) is a formula, \( \text{name} \) is the name of an operator and \( x_1, \ldots, x_n \) its parameters.

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

   \[
   \Box \forall x_{j_1} : t_{j_1}, \ldots, \forall x_{j_m} : t_{j_m} (F \rightarrow \exists x_{k_1} : t_{k_1} \ldots \exists x_{k_p} : t_{k_p} (G \land \text{name}(x_1, \ldots, x_n)))
   \]

   where \( F \) and \( G \) are formulae, \( \text{name} \) 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.

LPK implements both reject and select schemata by means of additional fields in the definition of an operator (besides the PRE and POST fields, specifying the pre- and post-condition of the operator). Let us consider first the reject schemata provided by the language.

The field **ONLY-IF** in an operator description specifies conditions that must hold in a state when an operator instance is true at that state:

<table>
<thead>
<tr>
<th>Schema</th>
<th>Semantics of the <strong>ONLY-IF</strong> field</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{op name } x_1 : t_1, \ldots, x_k : t_k ) <strong>ONLY-IF</strong> ( F_1, \ldots, F_n )</td>
<td>( \Box \forall x_1 : t_1, \ldots, \forall x_k : t_k (\text{name}(x_1, \ldots, x_k) \rightarrow F_1 \land \ldots \land F_n) )</td>
</tr>
</tbody>
</table>

Here, \( F_1, \ldots, F_n \) are formulae, whose free variables are among \( x_1, \ldots, x_k \). For instance, in the briefcase domain, the *takeout* operator can be specified as follows:

---

3The fact that the semantics of the **ONLY-IF** field is the same as that of action preconditions is not surprising (see [4]). It is however important that control information is kept separate from knowledge inherent to the domain (see Section 4).
\textbf{op} takeout \( x : \text{obj} \\
\text{PRE} \ in(x) \\
\text{POST} \ \neg in(x) \\
\text{ONLY-IF} \ \exists y : \text{loc} \ (at(x, y) \land \text{Goal}(at(x, y)))

The \textbf{ONLY-IF} field, stating that an object must be taken out of the briefcase only if it is at its destination, is equivalent to the control formula:

\( \Box \forall x : \text{obj} \ (\text{takeout}(x) \rightarrow \exists y : \text{loc} \ (at(x, y) \land \text{Goal}(at(x, y)))) \)

As a further example, in the gripper domain, the following field can be added to the definition of the \textit{move} operator, with parameters \textit{from} and \textit{to}, both of type \textit{room}:

\textbf{ONLY-IF} \ \textit{from} = B \land \textit{to} = A \rightarrow \forall g : \text{gripper}\ \text{free}(g), \\
\textit{from} = A \land \textit{to} = B \rightarrow \neg \exists g : \text{gripper}\ \text{free}(g) \lor \neg \exists x : \text{ball}\ at(x, A)

Then the robot must not leave room B unless its gripper are all free, and must not leave room A until there are balls in it and any gripper is still free.

Another reject schema is given by the \textit{NEXT} field, specifying conditions that must hold in the next state, when an action is performed:

<table>
<thead>
<tr>
<th><strong>Schema</strong></th>
<th><strong>Semantics of the NEXT field</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{op} name ( x_1 : t_1, \ldots, x_k : t_k ) \textbf{NEXT} ( F_1, \ldots, F_n )</td>
<td>( \Box \forall x_1 : t_1 \ldots \forall x_k : t_k ) \text{((name(x_1, \ldots, x_k) \rightarrow \circ (F_1 \land \ldots \land F_n)))}</td>
</tr>
</tbody>
</table>

Here, \( F_1, \ldots, F_n \) are formulae, whose free variables are among \( x_1, \ldots, x_k \). Typically, the \textit{NEXT} field is used to force a sequence of actions. For instance, in the briefcase domain, the random movement of the robot can be avoided by requiring that after a \textit{go} action, either a \textit{putin} or a \textit{takeout} action is performed:

\textbf{op} \textit{go} \ to : \textit{loc} \\
\text{POST} \ at(to), \ \forall y : \text{loc} \ (at(y) \Rightarrow \neg at(y)), \\
\ \forall x : \text{obj} \ \forall y : \text{loc} \ (in(x) \land at(x, y) \Rightarrow \neg at(x, y) \land at(x, to)) \\
\text{\textbf{NEXT} } \exists x : \text{obj} \ (\text{takeout}(x) \lor \text{putin}(x))

The \textit{NEXT} field is equivalent to the following control formula:

\( \Box \forall to : \text{loc} \ (\text{go}(to) \rightarrow \circ \exists x : \text{obj} \ (\text{takeout}(x) \lor \text{putin}(x))) \)

Temporal formulae can also occur in a control field. For instance, if, in the briefcase domain, we are only interested in solutions where each location is visited at most once, the \textit{NEXT} field in the above specification of the operator \textit{go} can be added the formula \( \Box \neg go(to) \), so that only models satisfying also \( \Box \forall to : \text{loc} \ (\text{go}(to) \rightarrow \circ \Box \neg go(to)) \) are considered.

A \textbf{LAST} field is also provided by the language, specifying what must hold before performing an action, but we omit its analysis here.

The \textit{opportunistic rules} in [13] are implemented by means of \textbf{PREFER} fields, specifying operators whose application should be preferred to the described one:

8
<table>
<thead>
<tr>
<th>Schema</th>
<th>Semantics of the PREFER field</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{op name } x_1 : t_1, \ldots, x_k : t_k )</td>
<td>( \square \forall x_1 : t_1 \ldots \forall x_k : t_k ) ( (F_1 \land \exists G_1 \rightarrow \neg \text{name}(x_1, \ldots, x_k)) )</td>
</tr>
<tr>
<td>( \text{PREFER } \text{op}_1 \text{ when } F_1 )</td>
<td>( \quad \ldots )</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( \quad \ldots )</td>
</tr>
<tr>
<td>( \text{op}_n \text{ when } F_n )</td>
<td>( \square \forall x_1 : t_1 \ldots \forall x_k : t_k ) ( (F_n \land \exists G_n \rightarrow \neg \text{name}(x_1, \ldots, x_k)) )</td>
</tr>
</tbody>
</table>

Here, \( \text{op}_1, \ldots, \text{op}_n \) are operator names, \( F_1, \ldots, F_n \) formulae, and each of the \( \exists G_i \) on the right is the existential closure (w.r.t. the parameters of the operator \( \text{op}_i \)) of the conjunction of the preconditions and ONLY-IF restrictions in the description of \( \text{op}_i \). Each formula on the right states that if \( F_i \) holds and the operator \( \text{op}_i \) is applicable, then no action \( \text{name} \) must be performed.

LPK provides select schemata representing suggestions of the kind: perform a given action as soon as possible, possibly under other conditions. We shall call them ASAP (As Soon As Possible) schemata.

The weaker form of ASAP schema concerning an operator 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 must be applied, with some values of the parameters:

<table>
<thead>
<tr>
<th>Schema</th>
<th>Semantics of the ASAP field</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{op name } x_1 : t_1, \ldots, x_k : t_k )</td>
<td>( \square (\exists x_1 : t_1 \ldots \exists x_k : t_k \left( G \land F_1 \land \ldots \land F_n \right) \rightarrow \exists x_1 : t_1 \ldots \exists x_k : t_k \left( F_1 \land \ldots \land F_n \land \text{name}(x_1, \ldots, x_k) \right) ) )</td>
</tr>
<tr>
<td>ASAP ( F_1, \ldots, F_n )</td>
<td>( \quad )</td>
</tr>
</tbody>
</table>

Here, \( F_1, \ldots, F_n \) are formulae, whose free variables are among \( x_1, \ldots, x_k \), and \( G \) is the conjunction of the formulae in the PRE and ONLY-IF fields in the definition of the operator.

For instance, in the gripper domain, we can force the robot to perform a pick action, whenever possible if the robot is in room A, by specifying:

\[
\text{op pick b : ball, r : room, g : gripper} \\
\text{PRE atRobby(r), at(b, r), free(g)} \\
\text{POST \neg at(b, r), \neg free(g), carry(b, g)} \\
\text{ASAP r = A}
\]

The ASAP field generates the following (simplified) control formula:

\[
\square (\exists b : \text{ball} \exists g : \text{gripper(atRobby(A) \land at(b, A) \land free(g)) \rightarrow} \\
\exists b : \text{ball} \exists g : \text{gripper pick(b, A, g)})
\]

A stronger form of ASAP guide (“strong ASAP”) forces the (simultaneous) application of the operator to all the values of the parameters to which it can be applied. It corresponds to the S-ASAP field in an operator description:
Here, $F_1, ..., F_n$ are formulae, whose free variables are among $x_1, ..., x_k$, and $G$ is the conjunction of the formulae in the PRE and ONLY-IF fields in the definition of $name$.

For instance, the following formula in the briefcase domain, stating that any object must be taken out of the briefcase as soon as it is at its destination:

$$\forall x : \text{obj} \ (in(x) \land \exists y : \text{loc} \ (at(x, y) \land \text{Goal}(at(x, y))) \rightarrow \text{takeout}(x))$$

is equivalent to an S-ASAP field, with formula $\exists y : \text{loc} \ (at(x, y) \land \text{Goal}(at(x, y)))$, in the definition of the operator takeout (given at page 8).

Existential and universal quantification in a restriction of the kind “as soon as possible” can also be mixed. For instance, let us consider the gripper domain with two robots, each one with two grippers, and three rooms, A, B and C. The balls are initially in A and B, and must be carried into C. Robot 1 is allowed to enter rooms A and C, and robot 2 can enter rooms B and C. The latter restriction guarantees that the pick actions of the two robots do not interfere, i.e. they can always be performed simultaneously (whenever possible). A reasonable control information would be the following: all robots must perform some possible pick action, whenever possible and when they are in either room A or room B. Adding a parameter of type robot to each relevant fluent and operator, a suitable control formula is

$$\forall r : \text{robot} \ \forall x : \text{room}$$
$$\quad (at(r, x) \land (x = A \lor x = B) \lor \exists b : \text{ball} \ at(b, x) \land \exists g : \text{gripper} \ \text{free}(r, g)$$
$$\quad \rightarrow \exists b : \text{ball} \ \exists g : \text{gripper} \ \text{pick}(r, b, x, g))$$

Such a formula is equivalent to the “generalized” ASAP field shown below:

**op** pick  r : robot, b : ball, x : room, g : gripper

PRE at(r, x), at(b, x), free(r, g)
POST ¬at(b, x), ¬free(r, g), carry(r, b, g)

ASAP $(x = A \lor x = B)$ FOR-ALL r x

In fact, an ASAP field may assume the following general form:

<table>
<thead>
<tr>
<th>Schema</th>
<th>Semantics of an ASAP/FOR-ALL field</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>op</strong> name $x_1 : t_1, ..., x_k : t_k$</td>
<td>(\forall x_{p_1} : t_{p_1} \ldots \forall x_{p_p} : t_{p_p} \quad (G \land F_1 \land \ldots \land F_n) \rightarrow \exists q_1 : t_{q_1} \ldots \exists q_r : t_{q_r} \ (F_1 \land \ldots \land F_n \land name(x_1, ..., x_k)))</td>
</tr>
<tr>
<td><strong>ASAP</strong> $F_1, ..., F_n$</td>
<td>(\forall x_{q_1} : t_{q_1} \ldots \forall x_{q_r} : t_{q_r} \ (G \land F_1 \land \ldots \land F_n) \rightarrow \exists q_1 : t_{q_1} \ldots \exists q_r : t_{q_r} \ (F_1 \land \ldots \land F_n \land name(x_1, ..., x_k)))</td>
</tr>
</tbody>
</table>

Here, \(\{x_{q_1}, ..., x_{q_r}\} = \{x_1, ..., x_k\} - \{x_{p_1}, ..., x_{p_p}\}\) and, again, $G$ is the conjunction of the formulae in the PRE and ONLY-IF fields of the operator. An S-ASAP schema is therefore nothing but a generalized ASAP schema where all the parameters occur in the FOR-ALL specification.
3 Towards a generalized view of planning

Many real problems do not naturally fit into the model of planning with instantaneous actions and a single final goal. In this section we show how to describe in LPK a simple problem where planning and scheduling aspects are intermixed. The goal of the plan is constituted by the accomplishment of a series of intermediate tasks. Other states of the system constrain the execution of each task. The example is the drilling machine example, presented in [6]: a drilling machine has to make holes in several workpieces, by use of two different bits b1 and b2, automatically switching between them. In order for a piece to be worked, it must stay fixed on a support, for a period of time strictly containing its working time. Moreover, working a piece must be preceded and followed by a period in which the drill is in an idle status.

The problem is that of controlling the dynamics of a physical system, by determining a sequence of values for some relevant “state variables”. The agent can directly influence the values of such variables, provided the physical constraints on the system are not violated. In our view, any property of the world that can be directly influenced by the agent behaviour is represented by an operator (on the contrary, a fluent is only indirectly influenced by the agent’s actions). Hence, the predicates **working, fixed** and **wait** are all action predicates and the language of the domain contains no fluents and no static predicates.

The relevant constraints on the “operators” are described in special fields of the operator definitions, requiring that certain temporal relations must hold between the considered operator and other “states” of the world. The fields used in this examples are **MEETS**, **MET-BY** and **DURING**. The semantics of such fields is given in the following table:

<table>
<thead>
<tr>
<th>Schema</th>
<th>Semantics of the temporal constraints fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>op name x₁ : t₁ , . . . , xₖ : tₖ</td>
<td>□∀x₁ : t₁ ∀xₖ : tₖ ((name(x₁ , . . . , xₖ ) → □(name(x₁ , . . . , xₖ ) ∨ \bigwedge_{i=1}^{n} F_i )))</td>
</tr>
<tr>
<td>MEETS F₁ , . . . , Fₙ</td>
<td>□∀x₁ : t₁ ∀xₖ : tₖ ((□name(x₁ , . . . , xₖ ) → □(name(x₁ , . . . , xₖ ) ∨ \bigwedge_{i=1}^{m} G_i )))</td>
</tr>
<tr>
<td>MET-BY G₁ , . . . , Gₘ</td>
<td>□∀x₁ : t₁ ∀xₖ : tₖ ((□name(x₁ , . . . , xₖ ) → □(name(x₁ , . . . , xₖ ) ∨ \bigwedge_{i=1}^{m} G_i )))</td>
</tr>
<tr>
<td>DURING H₁ , . . . , Hₚ</td>
<td>□∀x₁ : t₁ ∀xₖ : tₖ ((name(x₁ , . . . , xₖ ) → \bigwedge_{i=1}^{p} H_i ))</td>
</tr>
</tbody>
</table>

Note that a more faithful representation of the ommomous relations between time intervals defined in [2] (and here denoted by **MEETS**, **MET-BY**, **DURING**, respectively) can be given by means of conjunctions of the corresponding “un-starred” ones:
<table>
<thead>
<tr>
<th>Schema</th>
<th>Equiv. to</th>
<th>Schema</th>
<th>Equiv. to</th>
<th>Schema</th>
<th>Equiv. to</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEETS* F</td>
<td>MEETS F</td>
<td>MEETS F</td>
<td>DURING ~F</td>
<td>DURING* H</td>
<td>DURING H</td>
</tr>
<tr>
<td>MET-BY* G</td>
<td>MET-BY G</td>
<td>DURING* H</td>
<td>DURING H</td>
<td>MEETS H</td>
<td>MET-BY H</td>
</tr>
</tbody>
</table>

Finally, intermediate tasks are easily represented in LPK by means of the inclusion of formulae of the form ◇Task (“eventually, Task is accomplished”) in the description of the initial state. The complete description of the domain, with 8 workpieces (excluding the possible representation of control knowledge) is the following:

\[
type = (wp : 1..8) (bit : b1 \ b2); \\
init = wait, \forall x : wp \neg fixed(x), \\
\quad \forall x : wp \forall b : bit \diamond working(b, x); \\
\]

\[
\text{op } wait \text{ DURING } \forall b : bit \forall x : wp \neg working(b, x); \\
\]

\[
\text{op } working \ b : bit, \ x : wp \\
\quad \text{DURING } \forall z : bit \forall y : wp (working(z, y) \rightarrow z = b \land x = y), \\
\quad fixed(x), \neg wait \\
\quad \text{MEETS } fixed(x), \text{ wait} \\
\quad \text{MET-BY } fixed(x), \text{ wait}; \\
\]

\[
\text{op } fixed \ x : wp \\
\quad \text{DURING } \forall y : wp (fixed(y) \rightarrow x = y); \\
\]

4 Off-line Checks

A problem specification may suffer from different forms of incorrectness, consequently making the planner incomplete. Moreover, the addition of control knowledge risks sometimes to make the search harder, instead of being of help, because of the overhead due to the processing of the control theory itself (see [12]). It is therefore important that its encoding is kept as compact as possible and redundancies are avoided. Representing the whole planning problem in a logical language gives us the possibility to perform some important off-line consistency and redundancy checks with little extra effort. Such operations consider the set of formulae obtained from the specification of the problem, always excluding the description of the goal. A special utility in the system PADOK allows one to check the specification and warn the user, with respect to the following properties:

**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

12
control knowledge. This is a minimal requirement for the specification to be sound.

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

**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 operator instance, 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.

**Redundancy check:** each control formula is tested for derivability from the rest of the specification. In case of a positive answer, such a formula is pointed out as redundant.

Let us consider an example where control knowledge generates inconsistency. If, in the gripper domain, the pick operator is added a “strong ASAP” restriction (i.e. $\text{S-ASAP} \ r = A$), instead of a weak one (see page 9), then the corresponding control formula cannot safely be added to the kernel of the specification. In fact, if there is any ball in room $A$, it should be picked up contemporarily by the two grippers. But the two actions pick($x, A, Left$) and pick($x, A, Right$) are mutually exclusive, since the effect $\neg \text{at}(x, A)$ of one of them spoils the precondition at($x, A$) of the other. The kernel of the specification of the problem implies in fact $\Box \neg (\text{pick}(x, A, Left) \land \text{pick}(x, A, Right))$ (see [5]).

As a sample specification incorrectly preventing the applicability of an operator, let us consider a simplified form of the briefcase domain, where there is a single destination for all the objects, called home, that initially contains nothing. One could be tempted to say that the agent should not go to any location where there is nothing to pick up and bring home, adding the field

$$\text{ONLY-IF } \exists y : \text{obj at}(y, to)$$

To the specification of the operator go (with parameter to of type loc, see page 8). In this case, the control theory is consistent with the kernel of the specification, but the action go(home) can never be executed: there is nothing at home initially, nor will it ever be.

As an example of a redundant specification of control knowledge, consider the case of the gripper domain, where both good situations:

$$\text{good situations} = \forall x : \text{ball GoodSit(at}(x, B)) ;$$

and the ONLY-IF $r = A$ restriction on the operator pick are included. Then the formulae encoding good situations would be recognized as redundant.
5 Concluding Remarks

In this work we have described the planning language LPK, providing a set of
general schemata for the declarative specification of control information, as well
as the possibility of establishing temporal constraints between operators. The seman-
tics of LPK is given by means of a translation into temporal logic formulae, in
such a way that the specification of a planning domain is completely represented
by an LTL theory. This fact makes it possible to reduce planning to model search
in LTL and, more important, to implement useful tools aiming at pointing out
possible inconsistencies or redundancies in the specification.

The semantics of LPK does not exploit the full power of LTL, but makes use
only of future-time, unary operators. This makes it easier the direct translation
of an LPK specification into other planning languages.

The schemata provided by LPK to express control information are fairly rich:
all the examples of declaratively specified planning heuristics that can be found in
[3, 4, 9, 12, 13] can easily be expressed by means of either fluent-oriented schemata
or reject operator-oriented schemata. Note that when a planning language con-
tains no action predicates, every constraint must be formulated in terms of fluent
predicates, so that, for instance, an assertion of the kind “do not perform action
X” cannot be expressed directly, but in the form: “if F is true in the current
state, then G must not hold in the next state”, where F and G do not contain
any explicit reference to actions. On the contrary, LPK focuses on action-oriented
schemata, and sometimes the inverse conversion (from the fluent-oriented to the
action-oriented view) is needed. In all the cases we have considered, the repre-
sentation is quite natural, simple and compact.

As a final remark, we note that the select and reject rules described in [10] are
special cases of S-ASAP and ONLY-IF schemata, respectively, where the fields are
filled in with (either positive or negative) equalities, atoms and the goal predicate
applied to atoms, with the restriction that only variables occur as arguments in
such formulae.

References

R. N. Pelavin, and J. D. Tenenberg, editors, Reasoning about Plans, pages

In A. Bundy, editor, Proc. of the 8th Int. Conf. on Artificial Intelligence


