Compilation of a high-level temporal planning language into PDDL 2.1

Martin C. Cooper     Frédéric Maris     Pierre Régnier
IRIT, Université Paul Sabatier,
118 route de Narbonne
31062 Toulouse, cedex 9, France
{cooper, maris, regnier}@irit.fr

Abstract—An important aspect of any automatic planner is the language in which the user expresses problem instances. A rich language is an advantage for the user, whereas a simple language is an advantage for the programmer who must write a program to solve all planning problems expressible in the language. Considering the temporal planning language PDDL 2.1 as a low-level language, we show how to automatically compile a much richer language into PDDL 2.1. The worst-case complexity of this transformation is quadratic.

Our high-level language allows the user to declare timepoints and impose simple temporal constraints between them. Conditions and effects can be imposed at timepoints, over intervals and over sliding intervals within fixed intervals. Non-instantaneous transitions can also be modelled.

I. INTRODUCTION

Temporal planning is still in its infancy. Up until now, most research in temporal planning has concentrated on the important technical difficulties to be solved when adding the synchronisation of non-instantaneous actions to classical planning problems. These difficulties include solving temporally-expressive problems (which cannot be solved without using the concurrency of actions [4]). A typical example of a temporally-expressive problem is cooking: several ingredients or dishes must be cooked simultaneously in order to be ready at the same moment. Large-scale applications include the management of an airport or a railway station. Only some temporal planners can solve this kind of problems [3]. This paper shows how to render the solutions to these technical difficulties available to the non-expert user by means of a rich high-level temporal language together with an automatic compiler.

A temporal planning problem is composed of a set of durative actions, an initial state I and a goal G. I and G are sets of fluents (propositions). Initially, the fluents in I are true (all other fluents being undefined) and, after the execution of all the actions in the plan, all fluents in the goal state G must be true. Each action A has a duration, a set of conditions Cond(A) and a set of effects Eff(A). Cond(A) and Eff(A) are sets of propositions. A temporal plan is a set of action-instances ⟨A, tstart⟩, where A is an action and tstart its start time (positive rational number). For each action A in the temporal plan, each of its conditions p ∈ Cond(A) must be true when it is required by A, and each of its effects p ∈ Eff(A) is assigned true when it is produced by A.

Most of today’s temporal planners use the PDDL2.1 language [7] to represent domains and temporal planning problems. The notion of a durative action in PDDL2.1 is, however, quite restricted since it satisfies the postulate of [7]: actions are instantaneous but they can trigger or terminate continuous processes. According to this postulate, complex durative actions can be represented by decomposing them into simpler actions, each having effects at their start or end, and with conditions at their start, end or over all the duration of the action.

In order to easily model temporal problems, it is necessary to add high-level constructions to this basic language of PDDL2.1. Several extensions have been proposed, including conditions and/or effects over arbitrary intervals within an action, predictable external events and non-instantaneous goals. Fox et al. [8] showed that these extensions can be compiled into PDDL2.1 in polynomial time.

For the moment, only certain planners, such as TLP-GP [13] [14], directly incorporate extensions of this kind. As well as the above-mentioned extensions, TLP-GP also incorporates several high-level modalities (described in detail in Section II.B) which significantly extend the ease of representation of temporal problems. Other planners, such as IxTeT [9], [12] and HSTS [15], also use high-level temporal representation languages but these are tailor-made...
languages, and hence we restrict ourselves in the present paper to PDDL2.1 which is the most widely-used language. Due to lack of space, we refer the reader to [7] for a detailed description of PDDL2.1. As we will show in this article, the high-level constructions which are directly programmed in TLP-GP can also be compiled in polynomial time into PDDL2.1. This means that a user can code a temporal planning problem using the high-level constructions of TLP-GP and then use any planner which solves problems expressed in PDDL2.1.

In this article, as in almost all previous work in temporal planning, we do not authorize two instances of the same action to execute in parallel. We impose this restriction because if we allow several instances of the same action to overlap, then testing the existence of a valid plan becomes an EXPSPACE-complete problem [16]. This means that we conserve the PSPACE-complete complexity of classical planning [2]. Indeed, we also impose the slightly stronger condition that no two instances of the same action meet (in the sense of the primitive "meets" of [1]): there is always some non-empty interval between the end of one instance and the start of the following instance. This follows from a more general rule that any solution-plan should be stable to infinitesimal perturbations in the start times of actions, which is required to ensure that the correct execution of the plan does not rely on the perfect synchronization of actions (which may be impossible to achieve in practice).

This article is structured as follows. Section II presents the extensions to PDDL2.1 that are directly implemented in TLP-GP (time-points and modalities). Section III shows how the high-level modalities can be coded using only the atomic modalities of PDDL2.1. Section IV shows how just two time-points, such as the beginning or end of a set of simple linear binary (in)equality constraints between time points.

II. THE TEMPORAL LANGUAGE OF TLP-GP

The problem representation language that TLP-GP can process allows a much greater flexibility than PDDL2.1 for defining temporal domains and problems. Although its expressive power is identical to that of PDDL2.1, the following extensions that we have implemented in TLP-GP [13] [14] allow the user to express real-world problems much more easily:

- Time points within actions, other than start and end, along with sets of simple linear binary (in)equality constraints between time points.

- The high-level temporal modalities somewhere, anywhere and →over (transition-over).

High-level modalities allow the user to define complex relationships between a condition or effect and an interval, including being valid over the whole interval, at some point in the interval, within a sub-interval or being subject to a continuous transition over the interval. We show in Section III how just two atomic modalities, supported and forbidden, allow us to code the modalities at, over (which already exist in PDDL2.1 for the semantics of intervals corresponding to conditions of actions), as well as the new modalities somewhere, anywhere and →over.

A. Time-points

To provide a rich temporal representation of actions, we often need to associate time-points with each action. The most common are the start and end points of the action, but it may be necessary to define other time-points, such as the beginning or end of a condition or effect. Temporal constraints often exist between these points. For example, for each action, we have the constraint end = start + duration. In the following, we use the acronym PDDL-TE to represent our Temporally-Extended version of PDDL2.1.

Definition 1 (PDDL-TE action): An action has a duration, a set of conditions and a set of effects. A PDDL-TE action also has three extra sets compared to a PDDL2.1 action:

- timepoints is the set Tp of the action’s time-points. Tp contains, at least, the points start and end.
- timeconstraints is the set Tc of the constraints which must be satisfied by Tp. Tc contains, at least, the constraint end = start + duration.
- timealiases is the set Ta of aliases (names) for temporal intervals. Ta contains, at least, the alias all for the interval [start end].

Example: The action skyjet-fly reserves a skybase during a departure window [start end] and assigns the moment that the jet takes off (flypoint) to any time inside this window. The skyjet is actually flying from flypoint to flypoint + (flying-time). This can be coded in PDDL-TE as follows:

```plaintext
(:durative-action skyjet-fly
 :parameters (?s - skyjet ?b - skybase)
 :duration
 (= ?duration (flying-timewindow ?s))
 :timepoints (start end flypoint)
 :timealiases (all [start end])
```
B. High-level modalities

In PDDL-TE each condition and effect has a modality. The syntax and semantics of these different modalities are as follows.

\textbf{at a p}: This simple modality, already present in PDDL2.1, expresses the necessity of a condition or the occurrence of an effect \( p \) at a given moment:

- \( p \) is true at the instant \( a \).

\textbf{over [a b] p}: This modality expresses the truth of \( p \) over the interval \([a, b]\). It already existed for conditions over the duration of an action in PDDL2.1. Smith [17] extended it to any temporal intervals and Cushing et al. [5] extended it to effects. In PDDL-TE, it applies to conditions or effects over arbitrary intervals with the meaning:

- \( p \) is true over all the interval \([a, b]\).

\textbf{somewhere [a b] p}: This modality expresses uncertainty concerning the actual instant at which an effect occurs, or a condition is required. For example, a delivery company guarantees that packages will arrive within 24 hours, but the actual time of arrival is unknown (since dependent on traffic conditions). The semantics not only ensure that \( p \) is true at instant \( b \), but also protect the whole interval to ensure that \( p \) is not destroyed after being established:

- \( p \) is true at instant \( b \) (although the actual time when \( p \) becomes true is unknown).
- \( p \) cannot change from true to false in \([a, b]\).

\textbf{anywhere [a b] p}: This modality expresses the fact that the planner has a choice concerning the actual instant at which an effect occurs, or a condition is required. The effect (condition) must be true over some non-empty sub-interval of \([a, b]\). For example, an employee with flexible working hours has to work every day, but can decide when. We can also impose a minimal duration over which \( p \) must remain true using minimal duration \( d \):

\textbf{supported [a b] p}: \( p \) is necessarily true over the whole interval \([a, b]\). In the case of a condition, this means that \( p \) must remain true over \([a, b]\) after having been produced by another action (or being true since the initial state). In the case of an effect, this

\textbf{enforced [a b] p}: \( p \) is true at any instant \( c \) of the interval \([a, b]\). (Using minimal duration \( d \), \( p \) must remain true during the sub-interval \([c, c+d]\)).

\textbf{unenforced [a b] p}: \( p \) can be true over a part of the interval \([a, b]\) but not at the instant \( c \) (or during the sub-interval \([c, c+d]\) if minimal duration \( d \) is used).

\textbf{N.B.} The new modality \textit{anywhere} is, in fact, redundant, since it could be simulated using \textit{over} after the introduction of time-points \( c \) and \( c' \), together with the constraints \( a \leq c, c' = c + d \leq b \).

\[\rightarrow \text{over [a b] p}: \text{This modality (pronounced as transition over) can only be used in the definition of an effect. It expresses a continuous transition of a proposition \( p \) over an interval \([a, b]\). For example, when hammering in a nail, several hammer blows may be necessary, during which time the nail is neither completely in nor completely out. To allow a correct continuous transition of \( p \) by the action, the semantics also protect it over the interval:}\]

- \( p \) is true at instant \( b \).
- no other action can produce \( p \) or \( \neg p \) in \([a, b]\).

\[\text{N.B.} \text{In its original form [5], this modality represented the continuous transition from one truth value to another over the duration of an action. But this semantics implicitly imposes a condition within an effect, since, in order to pass from false to true, \( p \) must be false at the start of the action. To avoid imposing a condition within an effect, we do not insist that \( p \) be false at the beginning of the interval. For example, when hammering in a nail, it is not necessary to know whether the nail is already completely hammered in or not at the beginning of the action, just that it will be at the end.}\]

III. THE ATOMIC MODALITIES

This section introduces two atomic modalities from which it is possible to construct all the high-level modalities described in Section II.

\textbf{Definition 2 (modalities supported/forbidden)}: For a temporal interval \([a, b]\) and a literal \( p \), the atomic modalities \textit{supported} and \textit{forbidden} are defined by:

- \textit{supported [a b] p}: \( p \) is necessarily true over the whole interval \([a, b]\).
- \textit{forbidden [a b] p}: \( p \) is not true over the whole interval \([a, b]\).
means that the action itself produces and protects $p$ over $[a,b]$.

- **forbidden** $[a,b] p$: $p$ cannot be established (i.e. change from false or undefined to true) within the interval $[a,b]$. For an effect, the meaning is that no other action can produce $p$ within $[a,b]$.

The atomic modality **supported** indicates that $p$ is necessary on the temporal interval, and is equivalent to the modality necessary ($\square_{[a,b]} p$) in normal modal logic [10], [11]. The atomic modality **forbidden** indicates that the establishment of $p$ is impossible over the temporal interval. This modality is different to the negation of the modality possible of normal modal logic ($\neg \Diamond_{[a,b]} f$), since $p$ may be true over the interval provided it was established before $a$.

With the two atomic modalities supported and forbidden, we can impose or forbid any truth value or change of truth value on an interval. For example, supported $[a b] \neg p$ forbids $p=\text{true}$ on $[a,b]$. Imposing the establishment of a proposition $p$ within an interval is even simpler: it suffices to use a condition or effect at $a$ and $b$. For example, the conditions $\neg p$ at $a$ and $p$ at $b$ impose the condition that $p$ changes from false to true within $[a,b]$.

We now show how to define each of the high-level modalities at, over, somewhere, anywhere and $\rightarrow_{\text{over}}$ in terms of supported and forbidden (in some cases introducing new time-points and constraints).

**A condition which is required or an effect which occurs at a given instant:**

\[
\text{at } a \ p: \quad \text{supported } [a \ a] p
\]

**Maintaining a condition or effect over an interval:**

\[
\text{over } [a \ b] p: \quad \text{supported } [a \ b] p
\]

**Uncertainty concerning the time when a condition is required or an effect occurs:**

\[
\text{somewhere } [a \ b] p:
\begin{align*}
\text{supported } [b \ b] p \\
\text{forbidden } [a \ b] \neg p
\end{align*}
\]

**Possible choice concerning the time when a condition is required or an effect occurs:**

\[
\text{anywhere } [a \ b] p:
\begin{align*}
\text{supported } [c \ c'] p \\
\text{where } a \leq c \leq c' \leq b
\end{align*}
\]

‘Anywhere’ with a minimal duration:

\[
\text{minimal-duration } d \text{ anywhere } [a \ b] p:
\begin{align*}
\text{supported } [c \ c'] p \\
\text{where } a \leq c; c' - c \geq d; c' \leq b
\end{align*}
\]

**An effect in the form of a continuous transition over an interval (transition over):**

\[
\rightarrow_{\text{over}} [a \ b] p:
\begin{align*}
\text{supported } [b \ b] p \\
\text{forbidden } [a \ b] p \\
\text{forbidden } [a \ b] \neg p
\end{align*}
\]

The modality $\rightarrow_{\text{over}}$ cannot usefully be applied to a condition since, by imposing that $p$ is true at instant $b$ while also forbidding any change in the truth value of $p$ over $[a,b]$, means that it is semantically equivalent to supported $[a \ b] p$. It can, however, usefully be applied to an effect since forbidden only applies to other actions.

IV. **Compilation into PDDL2.1**

This section describes how to compile time-points and modalities into PDDL2.1.

**A. Representation of time-points**

A time-point $c_i$ (other than start and end) for an action $A$ is replaced by a new instantaneous action \text{TIMEPOINT}(A, c_i). The conditions required and the effects produced at $c_i$ by $A$ are simply added to the conditions and effects of \text{TIMEPOINT}(A, c_i).

To translate an action $A$ along with its time-points into PDDL2.1, we create two instantaneous dummy actions \text{BEFORE}(A) and \text{AFTER}(A), and an action \text{TIMEPOINT}(A, c_i) for each of the time-points $c_i$ of $A$, as shown in Figure 1 (for a single time-point). The dummy proposition Linked(A) is added to the initial state and to the goal of the problem, and as a condition of \text{BEFORE}(A). \text{BEFORE}(A) destroys Linked(A) and \text{AFTER}(A) re-establishes it. Each of the \text{TIMEPOINT}(A, c_i) actions has the proposition Enabled(A, c_i) as a condition and has $\neg$Enabled(A, c_i) as effects. Enabled(A, c_i) is also added to the effects of \text{BEFORE}(A) and $\neg$Enabled(A, c_i) is added to the conditions of \text{AFTER}(A).
In all figures in this paper we represent instantaneous actions by a vertical dotted line (as in Figure 1) and non-instantaneous actions by a rectangle. The duration of an action is given in square brackets after the name of the action. Conditions are written above an action, and effects below.

TIMEPOINT(A, ci) must be executed at least once to establish the condition ¬Enabled(A, ci) of AFTER(A), but cannot be executed more than once per instance of A since it has Enabled(A, ci) as a condition. This transformation therefore guarantees that each time-point ci is present exactly once per instance of A in the plan.

Our assumption that no two instances of an action A intersect or meet is essential so that it is possible to place the dummy actions BEFORE(A) and AFTER(A) between any two instances of action A.

1) Constraints

All the constraints between two time-points that can be expressed in the Simple Temporal Problem (STP) framework [6] can be modelled as follows:

- Order constraint ci< cj: we add (as shown below) the proposition Between(ci, cj) to the effects of the action TIMEPOINT(A, ci) and to the conditions of the action TIMEPOINT(A, cj), as well as adding ¬Between(ci, cj) to the initial state and to the effects of TIMEPOINT(A, ci). This clearly forces TIMEPOINT(A, ci) to be executed before TIMEPOINT(A, cj).

- To code ci≤ cj we can use the same solution as for ci< cj but we also add an action corresponding to the simultaneous execution of the two actions TIMEPOINT(A, ci) and TIMEPOINT(A, cj) with conditions Enabled(A, ci), Enabled(A, cj) and effects ¬Enabled(A, ci), ¬Enabled(A, cj).

- Fixed-distance constraint cj−ci = d: we add a dummy action D of duration d with ci = start(D) and cj = end(D).

- Minimum-distance constraint cj−ci > d: this constraint can be coded using the same solution as for the constraint ci< cj except that we add a buffer action B of duration d, as shown below.

- Maximum-distance constraint cj−ci < d: we add a buffer action B of duration d together with two constraints which force ci to be after the start of B and cj to be before the end of B, as shown below.
ordered time-points verified (maintained) over all the interval between two

TIMEPOINT(A,ci)[0]

Between(ci,Bend)

Between(ci,B-start)

TIMEPOINT(A,ci)[0]

¬Between(ci,B-end)

Between(B-start,ci)

¬Between(ci,B-end)

To code \(c_i \leq d\) we can simply combine the solutions for \(c_i \leq d\) and \(c_i < d\) (i.e., we add a buffer action \(B[d]\) as well as the action \(D[d]\)). A similar remark holds for the constraint \(c_i 

Clearly, the dummy and buffer actions \(D\) and \(B\), created during the above transformations, are different for each different constraint. We have omitted suffixes for simplicity of presentation. The buffer action \(B\) has a condition at its end that could make some planners incomplete for temporally-cyclic problems; we have shown in a previous paper how to transform these actions to solve this problem [3].

We can also code constraints which go beyond simple temporal constraints. For example, the non-intersection of the intervals \([c_i, c_j]\) and \([c_k, c_l]\) is equivalent to \(c_k \not\in [c_i, c_j]\) and \(c_l \not\in [c_i, c_j]\). Assuming that the constraints \(c_i < c_j\) and \(c_k < c_l\) have already been coded as described above, then \(c_k \not\in [c_i, c_j]\) and \(c_l \not\in [c_i, c_j]\) can be coded by adding the effect \(\neg\text{Between}(c_i, c_j)\) to action\(\text{TIMEPOINT}(A, c_i)\) and the effect \(\neg\text{Between}(c_i, c_j)\) to action\(\text{TIMEPOINT}(A, c_k)\).

2) Conditions/effects between time-points

If a condition (respectively, an effect) \(p\) must be verified (maintained) over all the interval between two ordered time-points \(c_i < c_j\), then we add \(p\) to the conditions (effects) of \(\text{TIMEPOINT}(A, c_i)\) and, to avoid the destruction of \(p\) on the interval \([c_i, c_j]\), we transform every action \(C\) which produces \(\neg p\) into an action \(C'\) which produces \(\neg\text{Between}(c_i, c_j)\) at the same moment. This transformation is shown in Figure 2 for the case when \(p\) is a condition of \(A\). The transformed action \(C'\) cannot destroy \(p\) on the interval \([c_i, c_j]\), since \(\text{Between}(c_i, c_j)\) would also be destroyed and could not be re-established since, as observed above,

TIMEPOINT(A,ci) (the only action which produces \(\text{Between}(c_i, c_j)\)), cannot be executed a second time within the same instance of \(A\).

If the condition (effect) \(p\) must be verified (maintained) over all the interval between two unordered time-points \(c_i\) et \(c_j\) then (as shown in Figure 3) we create two instantaneous actions \(\text{LEFT}(A,\{c_i, c_j\})\) and \(\text{RIGHT}(A,\{c_i, c_j\})\), which define an interval \(I\) containing these two time-points. We then protect \(p\) over the interval \(I\) which is necessarily a superset of the interval \([c_i, c_j]\). The new proposition \(\text{In}(\{c_i, c_j\})\) is added as a condition or effect of the different actions in order to constrain \(\text{TIMEPOINT}(A, c_i)\) and \(\text{TIMEPOINT}(A, c_j)\) to belong to \(I\). Finally, to avoid the destruction of \(p\) on the interval \(I\), we transform every action \(C\) which produces \(\neg p\) into an action \(C'\) which produces \(\neg\text{In}(\{c_i, c_j\})\) at the same moment. To ensure that \(\text{LEFT}(A,\{c_i, c_j\})\) and \(\text{RIGHT}(A,\{c_i, c_j\})\) are executed exactly once within each instance of \(A\), we apply the same solution as for \(\text{TIMEPOINT}\) actions using the

Figure 2. Imposing a condition over the interval between two ordered time-points.

Figure 3. Imposing a condition over the interval between two unordered time-points.
fluent Enabled(A,\{c_i,c_j\}) and \neg Enabled(A,\{c_i,c_j\}). C' and Enabled(A,\{c_i,c_j\}) are not shown in Figure 3 to avoid cluttering up the figure.

B. Compilation of the modalities

In order to compile all of our high-level modalities into PDDL2.1, it suffices to show how to compile the two atomic modalities supported and forbidden. We have shown in Section IV.A.2 how to impose a condition or an effect p between two time-points c_i,c_j. This is exactly equivalent to the modality supported \(p \in [c_i,c_j]\). It remains to show how to compile forbidden \(p \in [c_i,c_j]\) where c_i,c_j are time-points of action A. (We consider the case \(c_i \leq c_j\), but the case of unordered time-points c_i,c_j can also be coded using the same method as in Section IV.A.2.)

Figure 4. Compilation into PDDL2.1 of forbidden all p.

We create a new propositional variable Prevent(p,A) and using the method described in Section IV.A.2 we impose the condition Prevent(p,A) between the time-points c_i,c_j of A. For simplicity of presentation, we illustrate this in Figure 4 in the case when the interval \([c_i,c_j]\) is the whole duration of action A (denoted by all in PDDL2.1). Prevent(p,A) is added to the initial state and a new instantaneous dummy action ADD(p,A) is created which simply establishes Prevent(p,A). This simply ensures that the new condition Prevent(p,A) between the time-points c_i,c_j of A can always be satisfied. We also transform every action B which produces p into an action B' which produces p and \neg Prevent(p,A) at the same moment. This prevents B' from producing p during the execution of A. This is illustrated in Figure 4 for an action B which established p at its end.

An alternative method for compiling atomic modalities applied to arbitrary intervals, and which may in practice create slightly less actions, is to use the decomposition of actions into sub-actions proposed by Fox et al. [7] and then perform our transformations on the relevant sub-actions. Figure 5 shows an example of the result of such a decomposition of an action A, followed by the appropriate transformations to compile the action into PDDL2.1, as described above. The original action A has a condition Forbidden [i j] p on the sub-interval [i,j] and an effect Supported [j end] q on the sub-interval [j,end]. Action A is replaced by a container action together with three sub-actions. For simplicity, we do not show the transformations that must be applied to other actions (as illustrated in Figure 2 and Figure 4) as a consequence of the transformation of action A.

Figure 5. Example of the decomposition of an action into sub-actions each compiled into PDDL2.1.

C. Complexity

Let n denote the size of the original temporal planning problem, which we can equate with the sum, over all actions A, of the total number of time-points, constraints, conditions and effects. The compilation into PDDL2.1 described in this paper introduces O(n) new actions (time-point actions, buffer actions and dummy actions, described in Section IV). The compilation of each modality may, in the worst case, introduce O(n) new conditions/effects (as in the actions C' in Figure 2 and the action B' in Figure 4). Thus the total complexity of the compilation into PDDL2.1 is O(n^2).
V. CONCLUSION

In this paper, we have shown how to compile time-points and high-level modalities of conditions/effects into the temporal planning language PDDL2.1. Our language PDDL-TE is rich enough to allow the user to introduce time-points constrained by binary linear (in)equality constraints and to express conditions and effects of actions either at time-points or over intervals, with a choice among different modalities over intervals. The planner TLP-GP solves temporal planning problems expressed in the language PDDL-TE. The compilation from PDDL-TE to PDDL2.1 described in this paper demonstrates the theoretical possibility of using any planner capable of solving problems expressed in PDDL2.1 to solve problems expressed in the temporally-extended language PDDL-TE. An avenue for future research is to compare experimentally the efficiency of this compilation as well as the quality of plans obtained via this compilation. Another interesting question is whether other constraints (e.g. \(c_i-c_j>c_k-c_l\)) and other modalities (such as \(p\) cannot change its value twice, or \(p\land q\) must remain false…) could be useful and how they could be coded.

REFERENCES