When is Temporal Planning *Really* Temporal?

William Cushing and **Subbarao Kambhampati**
Dept. of Comp. Sci. and Eng.
Arizona State University
Tempe, AZ 85281

Mausam and **Daniel S. Weld**
Dept. of Comp. Sci. and Eng.
University of Washington
Seattle, WA 98195

Abstract

While even STRIPS planners must search for plans of unbounded length, temporal planners must also cope with the fact that actions may start at any point in time. Most temporal planners cope with this challenge by restricting action start times to a small set of *decision epochs*, because this enables search to be carried out in *state-space* and leverages powerful state-based reachability heuristics, originally developed for classical planning. Indeed, decision-epoch planners won the International Planning Competition's Temporal Planning Track in 2002, 2004 and 2006.

However, decision-epoch planners have a largely unrecognized weakness: they are incomplete. In order to characterize the cause of incompleteness, we identify the notion of *required concurrency*, which separates *expressive temporal action languages* from *simple* ones. We show that decision-epoch planners are only complete for languages in the simpler class, and we prove that the simple class is 'equivalent' to STRIPS! Surprisingly, no problems with required concurrency have been included in the planning competitions. We conclude by designing a complete state-space temporal planning algorithm, which we hope will be able to achieve high performance by leveraging the heuristics that power decision epoch planners.

1 Introduction

Although researchers have investigated a variety of architectures for temporal planning (e.g., plan-space: ZENO [Penberthy and Weld, 1994], VHPOP [Younes and Simmons, 2003]; extended planning graph: TGP [Smith and Weld, 1999], LPG [Gerevini and Serina, 2002]; reduction to linear programming: LPGP [Long and Fox, 2003]; and others), the most popular current approach is progression (or regression) search through an extended state space (e.g., SAPA [Do and Kambhampati, 2003], TP4 [Haslum and Geffner, 2001], TALPlan [Kvarnström *et al.*, 2000], TLPlan [Bacchus and Ady, 2001], and SGPlan [Chen *et al.*, 2006]) in which a search node is represented by a world-state augmented with the set of currently executing actions and their starting times. This architecture is appealing, because it is both conceptually simple and facilitates usage of powerful reachability heuristics, first developed for classical planning [Bonet *et al.*, 1997; Hoffmann and Nebel, 2001; Nguyen *et al.*, 2001; Helmert, 2004]. Indeed, SGPlan, which won the International Planning Competition's Temporal Planning Track in both 2004 and 2006, is such a progression planner.

There is an important technical hurdle that these temporal state-space planners need to overcome: each action could start at any of an infinite number of time points. Most of these planners avoid this infinite branching factor by a (seemingly) clever idea: restricting the possible start-time of actions to a small set of special time points, called *decision epochs*. Unfortunately, the popularity of this approach belies an important weakness — decision epoch planners are *incomplete* for many planning problems requiring concurrency [Mausam and Weld, 2006].

Seen in juxtaposition with their phenomenal success in the planning competitions, this incompleteness of decision epoch planners raises two troubling issues:

1. Are the benchmarks in the planning competition capturing the essential aspects of temporal planning?
2. Is it possible to make decision epoch planners complete while retaining their efficiency advantages?

In pursuit of the first question, we focused on characterizing what makes temporal planning really temporal — i.e. different from classical planning in a fundamental way. This leads us to the notion of *required concurrency*: the ability of a language to encode problems for which *all* solutions are concurrent. This notion naturally divides the space of temporal languages into those that can require concurrency (*temporally expressive*) and those that cannot (*temporally simple*). What is more, we show that the temporally simple languages are only barely different from classical, non-temporal, languages. This simple class, unfortunately, is the only class for which decision epoch planners are complete.

In pursuit of the second question, we show that the incompleteness of decision epoch planners is fundamental: anchoring actions to absolute times appears doomed. This leaves the temporal planning enterprise in the unenviable position of having one class of planners (decision epoch) that are fast but incomplete in fundamental ways, and another class of planners (e.g., partial order ones such as Zeno and VH-

POP) that are complete but often unacceptably slow. Fortunately, we find a way to leverage the advantages of both approaches: a temporally lifted state-space planning algorithm called TEMPO. TEMPO uses the advantage of lifting (representing action start times with real-valued variables and reasoning about constraints) from partial order planners, while still maintaining the advantage of logical state information (which allows the exploitation of powerful reachability heuristics). The rest of the paper elaborates our findings.

2 Temporal Action Languages

Many different modeling languages have been proposed for planning with durative actions, and we are interested in their relative expressiveness. The TGP language [Smith and Weld, 1999], for example, requires that an action's preconditions hold all during its execution, while PDDL 2.1.3 allows more modeling flexibility.[1] We study various restrictions of PDDL 2.1.3, characterized by the *times* at which preconditions and effects may be 'placed' within an action. Our notation uses superscripts to describe constraints on preconditions, and subscripts to denote constraints on effects: $\mathsf{L}^{\text{preconditions}}_{\text{effects}}$ is the template. The annotations are:

s "at-start"
e "at-end"
o: "over-all" (over the entire duration)

For example, $\mathsf{L}^{o}_{s,e}$ is a language where every action precondition must hold over all of its execution and effects may occur at start or at end. PDDL 2.1.3 does not define, or allow, effects over an interval of time: o is only used as an annotation on preconditions.

Many other language features could be included as possible restrictions to analyze; however, most end up being less interesting than one might expect. For example, deadlines, exogenous events (timed literals), conditional effects, parameters (non-ground structures), preconditions required at intermediate points/intervals inside an action, or effects occurring at arbitrary metric points (as in ZENO) can all be compiled into $\mathsf{L}^{s,o,e}_{s,e}$ [Smith, 2003; Fox *et al.*, 2004]. In particular, an analysis of just $\mathsf{L}^{s,o,e}_{s,e}$ is simultaneously an indirect analysis of these syntactically richer languages. Naturally these compilations can have dramatic performance penalties if carried out in practice; the purpose of such compilations is to ease the burden of analysis and proof. Of course, we also exclude some interesting language features (for the sake of simplicity), for example, metric resources and continuous change.

2.1 Basic Definitions

Space precludes a detailed specification of action semantics; thus, we merely paraphrase some of the relevant aspects of the PDDL 2.1.3 semantics for durative actions [Fox and Long, 2003].

Definition 1 (Actions) *A* model *is a total function mapping fluents to values and a* condition *is a partial function mapping fluents to values. A* transition *is given by two conditions: its preconditions, and its effects.*

An action*, A, is given by a beginning transition $begin(A)$, an ending transition $end(A)$, an over-all condition $o(A)$, and a positive, rational, duration $\delta(A)$.*

Definition 2 (Plans) *A* plan*, $P = \{s_1, s_2, s_3, \ldots, s_n\}$, is a set of steps, where each* step*, s, is given by an action, $action(s)$, and a positive, rational, starting time $t(s)$. The* makespan *of P equals*

$$\delta(P) = \max_{s \in P}(t(s) + \delta(action(s))) - \min_{s \in P}(t(s))$$

A rational model of time provides arbitrary precision without Real complications.

Definition 3 (Problems) *A* problem*, $\mathcal{P} = (\mathcal{A}, I, G)$, consists of a set of actions ($\mathcal{A}$), an initial model (I), and a goal condition (G).*

Definition 4 (States) *A (temporal)* state*, N, is given by a model, $state(N)$, a time, $t(N)$, and a plan, $agenda(N)$, recording the actions which have not yet finished (and when they started).*

A precise formulation of plan *simulation* is long and unnecessary for this paper; see the definition of PDDL 2.1.3 [Fox and Long, 2003]. Roughly, the steps of a plan, $P = \{s_1, \ldots, s_n\}$, are converted into a transition sequence, i.e., a classical plan. Simulating P is then given by applying the transitions, in sequence, starting from the initial model (a classical state). Simulation fails when the transition sequence is not executable, simulation also fails if any of the over-all conditions are violated. In either case, P is not *executable*. P is a *solution* when the goal condition is true in the model that results after simulation.

Plans can be *rescheduled*; one plan is a rescheduling of another when the only differences are the dispatch times of steps. Let $s' = delay(s, d)$ be the result of delaying a step s by d units of time: $t(s') = t(s) + d$ (and $action(s') = action(s)$). Similarly, $P' = delay(P, d)$ is the result of delaying an entire plan: $P' = \{delay(s, d) : s \in P\}$. Hastening steps or plans is the result of applying negative delay. A step s has *slack* d in an executable plan P when $P \setminus \{s\} \cup \{delay(s, -t)\}$ is also an executable plan for every value of t between 0 and d. A step without slack is *slackless*, likewise, a plan is *slackless* when every step is slackless, that is, the plan is "left-shifted".

Definition 5 (Completeness) *A planner is* complete *with respect to an action language L, if for all problems expressible in L, the planner is guaranteed to find a solution if one exists. A planner is* optimal*, with respect to language L and cost function c, if for all solvable problems expressible in L, the planner is guaranteed to find a solution minimizing c. A planner is* makespan optimal *if it is optimal with* makespan *as the cost function.*

2.2 Required Concurrency

We now come to one of the key insights of this paper. In some cases it is handy to execute actions concurrently; for example, it may lead to shorter makespan. But in other cases, concurrency is essential at a very deep level.

[1] PDDL 2.1.3 denotes level 3 of PDDL2.1 [Fox and Long, 2003].

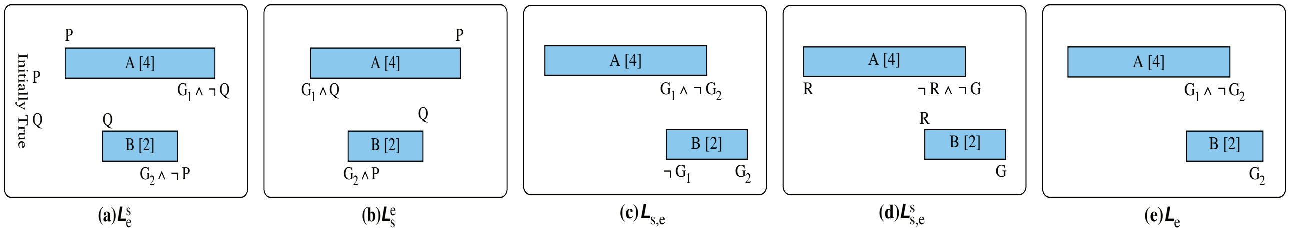

Figure 1: Preconditions are shown above actions at the time point enforced; effects are depicted below. Action durations are shown in square brackets. (**a**), (**b**), and (**c**): The first three problems demonstrate that L_e^s, L_s^e, and $\mathsf{L}_{s,e}$ are temporally expressive, respectively. In the first two problems, every solution must have both A and B begin before either can end. In (**c**), every solution must have B contain the end of A. (**d**): Modeling resources can easily lead to required concurrency. In this example, A provides temporary access to a resource R, which B utilizes to achieve the goal. (**e**): B must start in the middle of A, when nothing else is happening, to achieve makespan optimality.

Definition 6 (Required Concurrency) *Let $P=\{s_1,\ldots,s_n\}$ be a plan. A step, $s \in P$, is* concurrent[2] *when there is some other step, $s' \in P$, so that either $t(s) \le t(s') \le t(s) + \delta(action(s))$ or $t(s') \le t(s) \le t(s') + \delta(action(s'))$. A plan is* concurrent *when any step is concurrent, otherwise the plan is* sequential*. A solvable planning problem has* required concurrency *when all solutions are concurrent.*

To make this concrete, consider the plan in Figure 1(d). The literals above the action denote its preconditions and below denote effects. Starting with G and R false, assuming A and B are the only actions, the problem of achieving G has required concurrency. That is, both of the sequential plans (A before B or vice versa) fail to be executable, let alone achieve G.

Definition 7 (Temporally Simple / Expressive) *An action language, L, is* temporally expressive *if L can encode a problem with required concurrency; otherwise L is* temporally simple.

2.3 Temporally Simple Languages

Theorem 1 *L_e^o is temporally simple (and so is the TGP representation[3]).*

Proof: We will prove that L_e^o is temporally simple by showing that every concurrent solution of every problem in the language can be rescheduled into a sequential solution.

Fix a concurrent solution, P, of a problem $\mathcal{P} = (\mathcal{A}, I, G)$. Without loss of generality, assume the step which ends last, say $s \in P$, is a concurrent step.[4] Since actions have effects only at end, the model that holds after simulating all of $P \setminus \{s\}$ is identical to the model that holds immediately before applying the effects of $action(s)$ when simulating P. Since every precondition of an action holds over its entire duration, the preconditions of $action(s)$ hold immediately prior to applying its effects, i.e., in the final model of $P \setminus \{s\}$. Therefore $P' = (P \setminus \{s\}) \cup \{delay(s, \delta(action(s)))\}$ is an executable rescheduling of P. The final models in simulations of P and P' are identical, since both result from applying $action(s)$ to the same model. By induction on the number of concurrent steps (note that P' has fewer concurrent steps), there is a rescheduling of P into a sequential solution. □

Theorem 1 is interesting, because a large number of temporal planners (TGP, TP4 [Haslum and Geffner, 2001], HSP* [Haslum, 2006], TLPlan [Bacchus and Ady, 2001], and CPT [Vidal and Geffner, 2004]) have restricted themselves to the TGP representation, which is now shown to be so simple that essential temporal phenomena cannot be modeled! Note, for example, that the common motif of temporary resource utilization (Figure 1(**d**)) cannot be encoded in these representations. Yet some of these planners did extremely well in the last three International Planning Competitions. The reality: the majority of the problems in the Temporal Track do not require concurrency!

Note that the proof of Theorem 1 demonstrates a significantly stronger result than the theorem itself; not only does every problem in L_e^o have sequential solutions, but there is in fact a sequential rescheduling of every concurrent solution. This idea can be applied in reverse: problems in temporally simple languages can be optimally solved using classical techniques.

Theorem 2 *Let $\mathcal{P}$ be a planning problem in a temporally simple subset of* PDDL 2.1.3*, and let $\mathcal{P}'$ be a corresponding STRIPS problem where durations are ignored and every action is collapsed into a single transition.*[5]

There is a linear-time computable bijection between the slackless solutions of $\mathcal{P}$ and the solutions of $\mathcal{P}'$.

In particular, with the appropriate heuristics, optimal solutions to $\mathcal{P}$ can be found by solving $\mathcal{P}'$ instead. That is, STRIPS and temporally simple languages are essentially equivalent; though we do not delve into the details, one can show this correspondence in a formal manner using Nebel's

[2]Actions execute over closed intervals in PDDL 2.1.3, so actions with overlapping endpoints are executing concurrently — at an instantaneous moment in time.

[3]While the TGP representation is temporally simple, there is no perfect correspondence to any strict subset of PDDL 2.1.3, because they have slightly different semantics and hence different mutex rules. L_e^o is extremely close, however.

[4]If not, consider the problems $\mathcal{P}' = (\mathcal{A}, I, S)$ and $\mathcal{P}'' = (\mathcal{A}, S, G)$ where S is the model that holds after simulating just past all the concurrent steps of P; the suffix of P is a sequential solution to the latter problem, and the argument gives a sequential rescheduling of the prefix of P solving the former problem.

[5]This transformation is performed by MIPS, LPG, and SGPlan: see those planners for details [Edelkamp, 2003; Gerevini and Serina, 2002; Chen *et al.*, 2006].

framework of expressive power [Nebel, 2000].[6]

Proof: We give a linear-time procedure for mapping solutions of the STRIPS problem to slackless solutions, which is the bijection of the theorem. However, we omit showing that the inverse is a linear-time computable total function on slackless solutions. We also omit proof for any case besides L_e^o; the same basic technique (PERT scheduling) can be applied, with minor modifications, to every other case.

Consider some solution of $\mathcal{P}'$, $P' = A_1, A_2, \ldots, A_n$. Associate with every literal, $f{=}v$, the time at which it was achieved, $\tau(f{=}v)$, initially 0 if v is the initial value of f, and -1 otherwise. Find the earliest dispatch time of each A_i, $\tau(A_i)$, by the following procedure[7]; initializing $\tau(A_1)$ to ϵ and i to 1:

1. For all $(f{=}v) \in \mathit{effects}(A_i)$, if $v \neq \mathit{argmax}_{v'}\tau(f{=}v')$ set $\tau(f{=}v) = \tau(A_i) + \delta(A_i)$
2. Set $\tau(A_{i+1}) = \epsilon + \max(\{\tau(f{=}v) : (f{=}v) \in \mathit{precond}(A_{i+1})\} \cup \{\tau(A_i) + \delta(A_i) - \delta(A_{i+1})\})$
3. Increment i, loop until $i > n$

Then $P = \{s_i : \mathit{action}(s_i) = A_i \text{ and } t(s_i) = \tau(A_i)\}$ is a slackless rescheduling of P', preserving the order of the ends of actions, starting each action only after all of its preconditions have been achieved. In particular, P is a slackless solution. □

2.4 Temporally Expressive Languages

We have already seen one language, $\mathsf{L}_{s,e}^s$, which can express problems with required concurrency (Figure 1(**d**)). Of course, the full language, PDDL 2.1.3, is also a temporally expressive language. It is no surprise that by adding at-start effects to L_e^o one can represent required concurrency, but it is interesting to note that merely shrinking the time that preconditions must hold to at-start (i.e. the language L_e^s) also increases expressiveness. In fact, L_e^s is a particularly special temporally expressive language in that it exemplifies one of three fundamental kinds of dependencies that allow modeling required concurrency.

Theorem 3 L_e^s *is temporally expressive.*

The dual of L_e^s, L_s^e, is an odd language — all preconditions must follow effects. Nonetheless, the language is interesting because it is also one of the three minimal temporally expressive languages.

Theorem 4 L_s^e *is temporally expressive.*

It is not surprising that adding at-start effects (to a language allowing at-end effects) allows modeling required concurrency, because there is an obvious technique to exploit the facility: make a precondition of some action available only during the execution of another action. Figure 1(**d**) is a good example.

[6]The basic idea is to compile the scheduling into the planning problem.

[7]Technically, one must take ϵ to be some positive value by the requirements of PDDL 2.1.3. In a temporally simple language without a non-zero separation requirement (such as TGP) one can take ϵ as 0 instead.

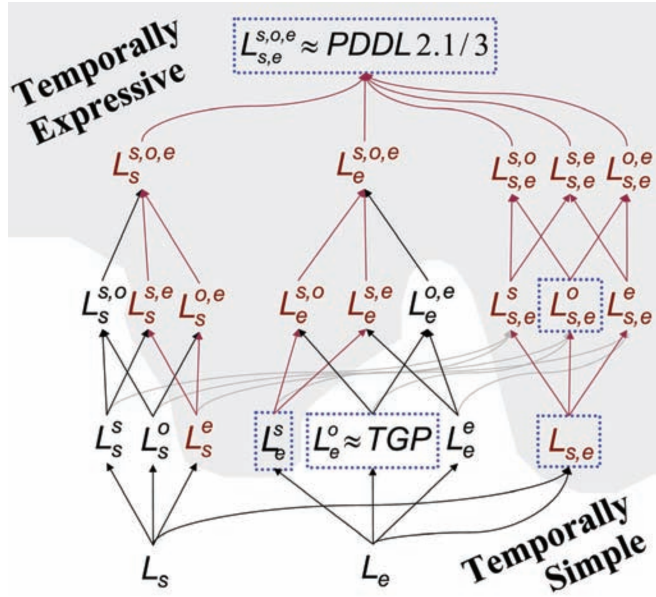

Figure 2: The taxonomy of temporal languages and their expressiveness; those outlined by dotted lines are discussed in the text.

What is surprising is that there is a different technique to exploit allowing effects at multiple times, one that does not even require any preconditions at all.

Theorem 5 $\mathsf{L}_{s,e}$ *is temporally expressive.*

Proof (of Theorems 3, 4, and 5): We prove that L_e^s, L_s^e, and $\mathsf{L}_{s,e}$ are temporally expressive by demonstrating problems in each language that require concurrency. See Figure 1(**a**), (**b**), and (**c**), respectively. □

2.5 Temporal Gap

Figure 2 places the languages under discussion in the context of a lattice of the PDDL sub-languages, and shows the divide between temporally expressive and simple. We have already shown that L_e^o, our approximation to TGP, is temporally simple. Surprisingly, the simple syntactic criteria of *temporal gap* is a powerful tool for classifying languages as temporally expressive or temporally simple.

Roughly, an action has temporal gap when there is no single time point in the action's duration when all preconditions and effects hold together (which is easy to check via a simple scan of the action's definition). A language permits temporal gap if actions in the language could potentially have temporal gap, otherwise a language forbids temporal gap. We show that a language is temporally simple if and only if it forbids temporal gap. This makes intuitive sense since without a temporal gap, the duration of an action becomes a secondary attribute such as cost of the action.[8]

[8]This understanding of temporal expressiveness in terms of temporal gap is reminiscent of the "unique main sub-action restriction" [Yang, 1990] used in HTN schemas to make handling of task interactions easy. The resemblance is more than coincidental, given that temporal actions decompose into transitions in much the same way as HTNs specify expansions (see Section 2.1).

Definition 8 *A* before-condition *is a precondition which is required to hold at or before at least one of the action's effects. Likewise, an* after-condition *is a precondition which is required to hold at or after at least one of the action's effects.*

A gap *between two temporal intervals, non-intersecting and not meeting each other, is the interval between them (so that the union of all three is a single interval). An action has* temporal gap *if there is a gap between its preconditions/effects, i.e., if there is*

a gap between a before-condition and an effect, or
a gap between an after-condition and an effect, or
a gap between two effects.

Actions without temporal gap have a critical point*: the (unique) time at which all the effects occur.*

Theorem 6 *A sub-language of* PDDL 2.1.3 *is temporally simple if and only if it forbids temporal gap.*

Proof: We begin by showing that forbidding temporal gap is necessary for a language to be temporally simple.

Languages permitting a gap between a before-condition and an effect, a gap between an after-condition and an effect, or a gap between effects are super-languages of L_e^s, L_s^e, or $\mathsf{L}_{s,e}$, respectively. By Theorems 3, 4, and 5, such languages are temporally expressive. Therefore temporally simple languages require that, for every action, all before-conditions hold just before any effect is asserted, all after-conditions hold just after any effect is asserted, and that all effects are asserted at the same time. That is, a temporally simple language must forbid temporal gap.

For the reverse direction, i.e., the interesting direction, we show that any language forbidding temporal gap is temporally simple, by demonstrating that slackless solutions to any problem can be rescheduled into sequential solutions, a generalization of the proof of Theorem 1. Fix some slackless solution to some problem in a language forbidding temporal gap.

Consider the sequence of critical points in the slackless solution, along with the models that hold between them, i.e., $M_0, c_1, M_1, c_2, \ldots, M_{n-1}, c_n, M_n$, where the c_i are critical points and the M_i are models. It is trivial to insert an arbitrary amount of delay between each critical point, lengthening the period of time over which each model holds, without altering them, by rescheduling steps. For example, multiplying each dispatch time by the maximum duration of an action achieves a sequential rescheduling preserving the sequence. For each critical point c_i, all of that action's before-conditions hold in M_{i-1} and all of its after-conditions hold in M_i, because the original plan is executable. Since those models are unaltered in the sequential rescheduling, the rescheduling is also executable, and thus a solution. □

Coming back to the space of languages, we have already noted that several popular temporal planners (e.g. TGP, TP4, HSP*, TLPlan, CPT) restrict their attention to temporally simple languages, which are essentially equivalent to STRIPS. The next section shows that most of the planners which claim to address temporally expressive languages, are actually incomplete in the face of required concurrency.

3 Decision Epoch Planning

The introduction showed that most temporal planners, notably those dominating the recent IPC Temporal Tracks, use the decision epoch (DE) architecture. In this section, we look in detail at this method, exposing a disconcerting weakness: incompleteness for temporally expressive action languages.

SAPA [Do and Kambhampati, 2003], TLPlan [Bacchus and Ady, 2001], TP4, and HSP$_a^*$ [Haslum and Geffner, 2001], among others, are all decision-epoch based planners. Rather than consider each in isolation, we abstract the essential elements of their search space by defining DEP. The defining attribute of DEP is search in the space of temporal states. The central attribute of a temporal state, N, is the world state, $state(N)$. Indeed, the world state information is responsible for the success and popularity of DEP, because it enables the computation of state-based reachability heuristics developed for classical, non-temporal, planning.

We define DEP's search space by showing how temporal states are refined; there are two ways of generating children:

Fattening: Given a temporal state, N, we generate a child, N_A, for every action, $A \in \mathcal{A}$. Intuitively, N_A represents an attempt to start executing A; thus, N_A differs from N only by adding a new step, s, to $agenda(N_A)$ with $action(s) = A$ and $t(s) = t(N)$.

Advancing Time: We generate a single child, N_{epoch}, by simulating forward in time just past the next transition in the agenda: $N_{epoch}{=}simulate(N, d + \epsilon)$, where $d{=}min\,\{t : s \in agenda(N)$ and $(t{=}t(s)$ or $t{=}t(s) + \delta(action(s)))$ and $t \geq t(N)\}$.

Our definition emphasizes simplicity over efficiency. We rely on $simulate$ to check action executability; inconsistent temporal states are pruned. Obviously, a practical implementation would check these as soon as possible.

The key property of DEP is the selection of decision epochs, that is, the rule for advancing time. In order for DEP to branch over action selection at a given time point, time must have advanced to that point. Since time always advances just past the earliest transition in the agenda, DEP can only choose to start an action when some other action has just ended, or just begun. Conversely, DEP is unable to generate solutions where the beginning of an action does not coincide with some transition. Forcing this kind of behavior is surprisingly easy.

Theorem 7 DEP *is incomplete for temporally expressive languages.*

Proof: It suffices to show that DEP is incomplete for L_e^s, L_s^e, and $\mathsf{L}_{s,e}$ to show that DEP is incomplete for all temporally expressive languages, by Theorem 6 (see Figure 2).

Figure 1(**c**) gives a $\mathsf{L}_{s,e}$ example which stumps DEP—achieving the goal requires starting B in the middle of A, but there are no decision epochs available in that interval. DEP can solve the problems in Figure 1(**a**) and (**b**), but not minor modifications of these problems. For example, altering A to delete G_2, in (**a**), forces B to start where there are no decision epochs. □

Theorem 8 DEP *is complete for temporally simple sub-languages of* PDDL 2.1.3*, but not makespan optimal.*

Proof: Figure 1(**e**) presents an example of makespan sub-optimality; DEP would find the serial solution, but not the optimal (concurrent) plan shown. Completeness follows trivially from Theorem 2: temporally simple languages have sequential solutions, and DEP includes every sequential plan in its search space (consider advancing time whenever possible).

4 Generalized Decision Epoch Planning

As the example of Figure 1(**c**) shows, DEP does not consider enough decision epochs. Specifically, it makes the mistaken assumption that every action will begin immediately after some other action has begun or ended. In Figure 1(**c**), however, action B has to *end* (not begin) after A ends. Thus, it is natural to wonder if one could develop a complete DE planner by exploiting this intuition. In short, the answer is "No." but the reason why the effort fails is instructive, so we present the DEP+ algorithm below.

We generalize DEP to DEP+ by considering both beginning and ending an action at the current decision epoch. This would involve altering the past in the case of ending an action if the decision epoch were not sufficiently far in the future; to address this, we take our decision epochs as the current time plus the maximum duration of any action in the problem. Let Δ be the maximum duration, i.e., $\Delta = \max_{A \in \mathcal{A}} \delta(A)$.

This raises a second issue: normally one would start the search at time 0, however, this would leave out the possibility of starting actions between 0 and Δ. We take the expedient of starting the search at time $-\Delta$, and continue to rely on $simulate$ to prune inconsistent temporal states, e.g., trying to start an action before time 0. In particular, the first decision epoch is at time 0, and attempting to end an action at the current decision epoch is not successful until the action would begin at or after time 0.[9]

Fattening: For every action A, we create two children of N. N_A^s is analogous to N_A in DEP— we commit to starting action A by adding a step s to $agenda(N_A^s)$ with $action(s)$=A and $t(s)$=$t(N_A^s) + \Delta$.

The latter, N_A^e, the essential difference from DEP, differs from N only in that $agenda(N_A^e)$ contains a new step, s, with $action(s)$=A and $t(s)$=$t(N_A^e) + \Delta - \delta(A)$.

Advancing Time: N_{epoch} is obtained from N by simulating to just after the first time where $t(N_{epoch}) + \Delta$ is the start or end of a step in $agenda(N_{epoch})$. Specifically, N_{epoch}=$simulate(N, d + \epsilon)$ where d=$\min\{t|s{\in}agenda(N)$ and $(t{=}t(s)$ or $t{=}t(s) + \delta(action(s)))$ and $t \geq t(N) + \Delta\}$.

Theorem 9 DEP+ *is incomplete for temporally expressive sub-languages of* PDDL 2.1.3.

Proof: DEP+ cannot generate the plan in Figure 3, because there are no decision epochs in the interval where C must execute; the beginning of B is not a decision epoch, because B is only included after the current decision epoch moves to just after the end of A, that is, past the interval that C must execute within. Similar examples demonstrate that DEP+ is incomplete for all other expressive languages. □

[9]This discussion has ignored the non-zero-separation requirement of PDDL 2.1.3, i.e., ϵ.

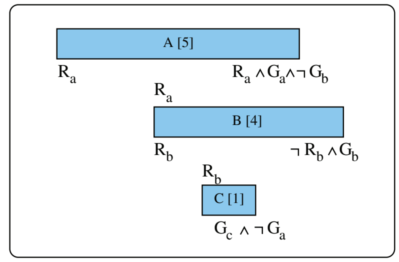

Figure 3: DEP+ can not find a plan to achieve $G_a \wedge G_b \wedge G_c$.

Furthermore, arbitrarily complex examples of chaining may be constructed; for example, split each action in Figure 3 into a million pieces. That is, trying to fix DEP+ by considering a denser set of decision epochs, or using some kind of lookahead, is a losing proposition.

DEP+ *does* improve on DEP, but, in just one way:

Theorem 10 DEP+ *is makespan optimal for temporally simple sub-languages of* PDDL 2.1.3.

Proof (Sketch): In a temporally simple language, by Theorem 6, every action has a critical point where all effects occur. Restrict child-generation so that the critical point of the action being added is always further in the future than the current decision epoch. Then every critical point eventually becomes a decision epoch. One can show that taking every critical point as a decision epoch is sufficient to allow the generation of every slackless solution. □

5 Temporally Lifted Progression Planning

The key observation about decision-epoch planning is that decisions about *when* to execute actions are made very eagerly — before all the decisions about *what* to execute are made. DEP attempts to create tight plans by starting actions only at those times where events are already happening. Unfortunately, for temporally expressive languages, this translates into the following two erroneous assumptions:

***1** Every action will start immediately after some other action has started or ended.

***2** The only conflicts preventing an earlier dispatch of an action, however indirect, involve actions which start earlier.

In developing DEP+, we noted the first flaw, and attempted to address it by allowing synchronization on the beginnings of actions as well as their ends. However, there does not appear to be any (practical) way of addressing the second flaw within the decision-epoch approach. One must either define *every time point* to be a decision epoch (branching over dense time!) or pick decision-epochs forwards and backwards, arbitrarily far, through time (as in LPGP [Long and Fox, 2003]).

Instead, we develop a complete state-space approach, by exploiting the idea of *lifting* over time: delaying the decisions about *when* to execute until all of the decisions about *what* to execute have been made. Note that VHPOP [Younes and Simmons, 2003] also lifts over time — we take a different approach that allows us to preserve state information at each search node.

Definition 9 *A* lifted temporal state*, $\mathcal{N}$, is given by the current temporal variable, $\tau(\mathcal{N})$, a model, $state(\mathcal{N})$, a lifted plan, $agenda(\mathcal{N})$, and a set of temporal constraints, $constraints(\mathcal{N})$.*

We retain the terminology used in DEP, and DEP+, to highlight the similarity of the approaches, despite the differences in details which arise from lifting time. For example, the agenda in a lifted temporal state is different from that in a (ground) temporal state — we replace exact dispatch times ($t()$) with temporal variables ($\tau_{begin}()$), and impose constraints through $constraints(\mathcal{N})$. In fact, we associate every step, s, with two temporal variables: $\tau_{begin}(s)$ and $\tau_{end}(s)$. All the duration constraints $\tau_{end}(s) - \tau_{begin}(s) = \delta(action(s))$ and mutual exclusion constraints $\tau_x(s) \neq \tau_y(t)$, for mutually exclusive transitions $x(action(s))$ and $y(action(t))$ (x and y are each one of *begin* or *end*), are always, implicitly, part of $constraints()$.

The aspect of lifted and ground temporal states that remains identical is the current world state, $state(N)$. In both cases this maps every fluent to the value it has at the current time. In particular, this is exactly the information needed to leverage the state-based reachability heuristics developed for classical planning. With respect to lifted temporal states, TEMPO is a complete and optimal state-space temporal planning algorithm, given by the following child-generator function:

Fattening: Given a lifted state, $\mathcal{N}$, we generate a child, $\mathcal{N}_A$, for every action, $A \in \mathcal{A}$. As before $\mathcal{N}_A$ represents starting A; we add a step s to $agenda(\mathcal{N}')$ with $action(s) = A$. In addition, we add "$\tau_{begin}(s) \geq \tau(\mathcal{N})$" to $constraints(\mathcal{N}')$. Unlike before, we immediately simulate: $\mathcal{N}_A = simulate(\mathcal{N}', \tau_{begin}(s))$. In particular, everything in $agenda(\mathcal{N}_A)$ has already started.

Advancing Time: For every $s \in agenda(\mathcal{N})$, we generate a child, $\mathcal{N}^A_{epoch}$, where $A = action(s)$. Note that A has already started; this is a decision to end A. Specifically, we add "$\tau_{end}(s) \geq \tau(\mathcal{N})$" to $constraints(\mathcal{N}')$ and then simulate: $\mathcal{N}^A_{epoch} = simulate(\mathcal{N}', \tau_{end}(s))$

In essence, TEMPO is searching the entire space of sequences of transitions (beginnings and endings of actions), in prefix order. That is, every search state corresponds to the unique sequence of transitions that, if (assigned dispatch times and) executed, result in the given (lifted) temporal state. Of course, just before terminating, TEMPO must actually pick some particular assignment of times satisfying $constraints(\mathcal{N})$ (for a state, $\mathcal{N}$, satisfying the goal) in order to return a ground plan. Since $constraints(\mathcal{N})$ will, among other things, induce a total ordering, this will not be very difficult. So it should not be very surprising that TEMPO is guaranteed to find solutions — if there is a solution, it has a sequence of transitions, and TEMPO will eventually visit that sequence, and find an assignment of times.

Theorem 11 TEMPO *is complete for any temporally expressive (or simple) sub-language of* PDDL 2.1.3*, moreover, makespan optimal.*

Proof: Every potential permutation of beginnings and endings of actions can be generated by appropriate decisions at *fattening* and *advance-time* choice-points (if not pruned by $simulate()$). The transition sequence of any concurrent plan is one such permutation, in particular a makespan optimal solution defines one such permutation. Pruning occurs if $simulate()$ fails at a search node, i.e., a precondition is violated. No descendant of this search node can ever change the state where the precondition is evaluated: every descendant would likewise fail to be executable. Solutions are, of course, executable, so TEMPO does not prune any solutions. It follows that TEMPO is complete; makespan optimality follows from the fact that the appropriate transition sequence is in the search space, and the optimal dispatch is easy to find. □

6 Discussion and Related Work

It should be noted that our analysis of temporal expressiveness was done at the language level, and most of our conditions for expressiveness were necessary rather than sufficient. In particular, it is obviously possible to write a domain in a temporally expressive language that does not require concurrency (or write a problem for a temporally expressive domain that does not require concurrency). For example, the (temporal) `Rovers` domain, contains actions with temporal gap. Nonetheless, `Rovers` is a temporally simple domain. This is not a contradiction of Theorem 6 — any language permitting the `Rovers` encoding also contains *other* domains and problems that require concurrency. It would be interesting to catalog domain/problem level necessary/sufficient conditions for required concurrency.

Several planners have considered using classical techniques augmented with simple scheduling to do temporal planning, for example, SGPLAN, MIPS, LPG-td, and CRIKEY [Chen *et al.*, 2006; Edelkamp, 2003; Gerevini and Serina, 2002; Halsey *et al.*, 2004]. That is, the planners only consider sequential solutions, but reschedule these using the temporal information. Actually, CRIKEY does not quite fit this classification; CRIKEY attempts to do classical planning as much as possible, and switches to a TEMPO like search to handle actions that could easily lead to required concurrency (*envelope* actions). Modulo unimportant details, an equivalent perspective on CRIKEY is as an implementation of TEMPO that strives to cut down the number of transition sequences actually considered by identifying actions where it is safe to immediately apply the ending transition after the beginning transition (*non-envelope* actions). Unfortunately, our preliminary investigation reveals that the pruning that results is not completeness-preserving; the conditions used to classify actions as safe are too generous.

7 Conclusion

Motivated by the observation that the most successful temporal planners are incomplete [Mausam and Weld, 2006], this paper presents a detailed examination of temporal planning algorithms and action languages. We make the following contributions:

- We introduce the notion of *required concurrency* which divides temporal languages into *temporally simple* (where concurrency is never required in order to solve a problem) and *temporally expressive* (where it may be) classes. Using the notion of *temporal gap*, we then decompose subsets of PDDL 2.1.3 into a lattice which distinguishes the expressive and simple sub-languages.
- We show that temporally simple languages are essentially equivalent to STRIPS in expressiveness. Specifically, we show a linear-time computable mapping into STRIPS, with no increase in the number of actions. Thus, any classical planner may be used to generate solutions to temporally simple planning problems!
- We prove that a large class of popular temporal planners, those that branch on a restricted set of decision epochs (*e.g.*, all state-space planners like SAPA, SGPlan), are complete *only* for the temporally simple languages. In fact, there exist problems even in simple languages for which these planners are not optimal. Since these decision-epoch planners won the temporal track of the last three planning competitions, we question the choice of problems used in the competitions.[10]
- On a constructive note, we sketch the design of a complete state-space temporal planning algorithm, TEMPO, which we hope will be able to achieve high performance by leveraging the heuristics that power decision epoch planners.

Acknowledgments

We thank J. Benton, Minh B. Do, Maria Fox, David Smith, Sumit Sanghai, and Menkes van den Briel for helpful discussions and feedback. We also appreciate the useful comments of the anonymous reviewers on the prior draft. This work was supported by NSF grants IIS-0307906 and IIS-308139, ONR grants N00014-02-1-0932, N00014-06-1-0147, and N00014-06-1-0058, the Lockheed Martin subcontract TT0687680 to ASU as part of the DARPA Integrated Learning program, and the WRF/TJ Cable Professorship.

References

[Bacchus and Ady, 2001] F. Bacchus and M. Ady. Planning with resources and concurrency: A forward chaining approach. In *IJCAI*, 2001.

[Bonet *et al.*, 1997] B. Bonet, G. Loerincs, and H. Geffner. A robust and fast action selection mechanism for planning. In *AAAI*, 1997.

[Chen *et al.*, 2006] Y. Chen, C. Hsu, and B. Wah. Temporal planning using subgoal partitioning and resolution in SGPlan. *JAIR*, to appear, 2006.

[Do and Kambhampati, 2003] M. B. Do and S. Kambhampati. SAPA: A multi-objective metric temporal planner. *JAIR*, 20:155–194, 2003.

[Edelkamp, 2003] S. Edelkamp. Taming numbers and duration in the model checking integrated planning system. *JAIR*, 20:195–238, 2003.

[Fox and Long, 2003] M. Fox and D. Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *JAIR*, 20:61–124, 2003.

[Fox *et al.*, 2004] M. Fox, D. Long, and K. Halsey. An investigation into the expressive power of PDDL2.1. In *ECAI*, pages 328–342, 2004.

[Gerevini and Serina, 2002] A. Gerevini and I. Serina. LPG: A planner based on local search for planning graphs. In *AIPS*, 2002.

[Halsey *et al.*, 2004] K. Halsey, D. Long, and M. Fox. CRIKEY - a temporal planner looking at the integration of scheduling and planning. In *Workshop on Integrating Planning into Scheduling, ICAPS*, pages 46–52, 2004.

[Haslum and Geffner, 2001] P. Haslum and H. Geffner. Heuristic planning with time and resources. In *ECP*, 2001.

[Haslum, 2006] P. Haslum. Improving heuristics through relaxed search — an analysis of TP4 and HSP$^*_a$ in the 2004 planning competition. *JAIR*, 25:233–267, 2006.

[Helmert, 2004] M. Helmert. A planning heuristic based on causal graph analysis. In *ICAPS*, pages 161–170, 2004.

[Hoffmann and Nebel, 2001] J. Hoffmann and B. Nebel. The FF planning system: Fast plan generation through heuristic search. *JAIR*, 14:253–302, 2001.

[Kvarnström *et al.*, 2000] J. Kvarnström, P. Doherty, and P. Haslum. Extending TALplanner with concurrency and resources. In *ECAI*, 2000.

[Long and Fox, 2003] D. Long and M. Fox. Exploiting a graphplan framework in temporal planning. In *ICAPS*, pages 51–62, 2003.

[Mausam and Weld, 2006] Mausam and D. S. Weld. Probabilistic temporal planning with uncertain durations. In *AAAI*, 2006.

[Nebel, 2000] B. Nebel. On the compilability and expressive power of propositional planning formalisms. *JAIR*, 12:271–315, 2000.

[Nguyen *et al.*, 2001] X. Nguyen, S. Kambhampati, and R. Nigenda. Planning graph as the basis for deriving heuristics for plan synthesis by state space and CSP search. *AIJ*, 135:73–123, 2001.

[Penberthy and Weld, 1994] S. Penberthy and D. Weld. Temporal planning with continuous change. In *AAAI*, 1994.

[Smith and Weld, 1999] D. E. Smith and D. Weld. Temporal planning with mutual exclusion reasoning. In *IJCAI*, 1999.

[Smith, 2003] D. E. Smith. The case for durative actions: A commentary on PDDL2.1. *JAIR*, 20:149–154, 2003.

[Vidal and Geffner, 2004] V. Vidal and H. Geffner. CPT: An optimal temporal POCL planner based on constraint programming. In *IPC (ICAPS)*, 2004.

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

[Younes and Simmons, 2003] H.L.S. Younes and R. G. Simmons. VHPOP: Versatile heuristic partial order planner. *JAIR*, 20:405–430, 2003.

[10] While the competition's problems were encoded in a language *capable* of encoding problems with required concurrency, it appears that none of the actual problems *did* require concurrency.