Temporal Planning With Required Concurrency Using Classical Planning

Sergio Jiménez and **Anders Jonsson** and **Héctor Palacios**
Dept. Information and Communication Technologies
Universitat Pompeu Fabra
Roc Boronat 138
08018 Barcelona, Spain
{sergio.jimenez,anders.jonsson,hector.palacios}@upf.edu

Abstract

In this paper we describe two novel algorithms for temporal planning. The first algorithm, TP, is an adaptation of the TEMPO algorithm. It compiles each temporal action into two classical actions, corresponding to the start and end of the temporal action, but handles the temporal constraints on actions through a modification of the Fast Downward planning system. The second algorithm, TPSHE, is a pure compilation from temporal to classical planning for the case in which required concurrency only appears in the form of single hard envelopes. We describe novel classes of temporal planning instances for which TPSHE is provably sound and complete. Compiling a temporal instance into a classical one gives a lot of freedom in terms of the planner or heuristic used to solve the instance. In experiments TPSHE significantly outperforms all planners from the temporal track of the International Planning Competition.

Introduction

In this paper we explore the application of classical planning to temporal planning tasks in which actions do not necessarily follow each other sequentially, have a duration and may overlap. Temporal planning can express features such as deadlines, conditional effects, conditions during the application of actions, or effects occurring at arbitrary time points (Fox and Long 2003). Interestingly, the temporal planners with the best performance do not fully handle the expressiveness of temporal planning, and typically implement incomplete approaches effective in domains where concurrency is not required, i.e. the majority of the IPC benchmarks (Coles et al. 2012). Examples are SGPlan (Chen, Wah, and Hsu 2006), winner at IPC-2008 and YAHSP (Vidal 2014), winner at IPC-2011 and IPC-2014.

In this work we introduce two novel temporal planners, TP and TPSHE. TP is based on the TEMPO algorithm (Cushing, Kambhampati, and Weld 2007), which can handle any form of concurrency as long as events are asynchronous but, as far as we know, has never been implemented. TP is a hybrid: on one hand it compiles every temporal action into two classical actions that correspond to the start and end of the action. On the other hand it modifies the Fast Downward planning system (Helmert 2006) to keep track of temporal constraints on actions. TPSHE is a compilation from temporal to classical planning for the class of temporal planning instances in which required concurrency appears only as single hard envelopes (Coles et al. 2009), i.e. single actions that temporarily add resources needed by other actions. In experiments, we show that even though TPSHE is incomplete, it significantly outperforms all planners from the temporal track of the IPC.

An incomplete planner is justified when the language is too expressive, and theoretical analysis can provide limits on such restrictions. For example, planning with numeric fluents is undecidable (Helmert 2002), and planners instead handle tractable fragments. Here, we identify novel classes of temporal instances that are provably solvable using different compilations. We first describe two classes of temporal instances that are provably *sequential*, i.e. any plan for these instances can be converted into a sequential plan. We then extend these classes to single hard envelopes.

The paper is organized as follows. We first introduce planning models and notation. Next, we introduce the two planners, TP and TPSHE. We then describe two classes of temporal instances that are provably sequential, and extend these classes to single hard envelopes. Finally, we present results from experiments, and conclude with a discussion of related work and possible extensions to our work.

Background

This section introduces the classical planning model, the temporal planning model, and *sequential temporal planning*, a subclass of temporal planning instances that can be solved straightforwardly using a classical planner.

Classical Planning

Let F be a set of propositional variables or *fluents*. A *state* $s \subseteq F$ is a subset of fluents that are true, while all fluents in $F \setminus s$ are implicitly assumed to be false. A subset of fluents $F' \subseteq F$ *holds* in a state s if and only if $F' \subseteq s$.

A classical planning instance is a tuple $P = \langle F, A, I, G \rangle$, where F is a set of fluents, A a set of actions, $I \subseteq F$ an initial state, and $G \subseteq F$ a goal condition (usually satisfied by multiple states). Each action $a \in A$ has precondition $\mathsf{pre}(a) \subseteq F$, add effect $\mathsf{add}(a) \subseteq F$, and delete effect $\mathsf{del}(a) \subseteq F$, each a subset of fluents. Action a is *applicable* in state $s \subseteq F$ if and only if $\mathsf{pre}(a)$ holds in s, and applying a in s results in a new state $s \oplus a = (s \setminus \mathsf{del}(a)) \cup \mathsf{add}(a)$.

A *plan* for P is a sequence of actions $\Pi = \langle a_1, \ldots, a_n \rangle$ such that a_1 is applicable in I and, for each $2 \leq i \leq n$, a_i is applicable in $I \oplus a_1 \oplus \cdots \oplus a_{i-1}$. The plan Π *solves* P if G holds after applying $a_1, \ldots, a_n$, i.e. $G \subseteq I \oplus a_1 \oplus \cdots \oplus a_n$.

Temporal Planning

A temporal planning instance is a tuple $P = \langle F, A, I, G \rangle$, where F, I, and G are defined as for classical planning. However, each $a \in A$ is a temporal action composed of

- $d(a)$: duration,
- $\mathsf{pre}_s(a)$, $\mathsf{pre}_o(a)$, $\mathsf{pre}_e(a)$: preconditions of a at start, over all, and at end, respectively,
- $\mathsf{add}_s(a)$, $\mathsf{add}_e(a)$: add effects of a at start and at end,
- $\mathsf{del}_s(a)$, $\mathsf{del}_e(a)$: delete effects of a at start and at end.

Since $\mathsf{pre}_o(a)$ has to hold for the duration of a, we use *context* as an abbreviation for "precondition over all".

A plan for P is not a sequence but rather a set of action-time pairs $\Pi = \{(a_1, t_1), \ldots, (a_n, t_n)\}$, where t_i, $1 \leq i \leq n$, is the scheduled start time of action a_i. Let $(a, t) \in \Pi$ be an action-time pair of the plan. Although a has continuous duration, its effects only apply at the start and end. We can therefore associate two discrete *events* start_a and end_a to a, such that start_a has associated time t and end_a has associated time $t + d(a)$. The plan Π induces an event sequence $\Pi_e = \langle e_1, \ldots, e_{2n} \rangle$ that includes start_a and end_a for each $(a, t) \in \Pi$ and is ordered by the associated times of events. We only consider plans that associate unique times to events, i.e. actions cannot start and/or end simultaneously.

We can represent the events start_a and end_a as classical actions to describe the semantics of a temporal plan Π:

- $\mathsf{pre}(\mathsf{start}_a) = \mathsf{pre}_s(a)$, $\quad \mathsf{pre}(\mathsf{end}_a) = \mathsf{pre}_e(a)$,
- $\mathsf{add}(\mathsf{start}_a) = \mathsf{add}_s(a)$, $\quad \mathsf{add}(\mathsf{end}_a) = \mathsf{add}_e(a)$,
- $\mathsf{del}(\mathsf{start}_a) = \mathsf{del}_s(a)$, $\quad \mathsf{del}(\mathsf{end}_a) = \mathsf{del}_e(a)$.

Thus, to determine whether Π solves P, we first define $s_j = I \oplus e_1 \oplus \cdots \oplus e_j$, $0 \leq j \leq 2n$. For Π to solve P, each event e_j, $1 \leq j \leq 2n$, has to be applicable in s_{j-1}, and G has to hold in the resulting state, i.e. $G \subseteq s_{2n}$. In addition, Π_e has to respect the contexts of temporal actions. Specifically, for each $(a, t) \in \Pi$, let j and k be the indices of start_a and end_a in Π_e. Event sequence Π_e respects the context of a if and only if $\mathsf{pre}_o(a) \subseteq s_i$, $j \leq i < k$, i.e. the context has to hold in each intermediate state between the start and end of a.

Sequential Temporal Planning

A temporal plan Π is *sequential* if its induced event sequence is $\Pi_e = \langle \mathsf{start}_{a_1}, \mathsf{end}_{a_1}, \ldots, \mathsf{start}_{a_n}, \mathsf{end}_{a_n} \rangle$, i.e. temporal actions are fully applied in sequence, one by one. A temporal instance P is *sequential* if there exists a sequential plan Π solving P. P is *inherently sequential* if *all* its solutions can be rescheduled to form a sequential plan.

To solve a sequential temporal instance, we can map each temporal action $a \in A$ to a *compressed action* (Coles et al. 2009), i.e. a classical action c_a that simulates all of a at once. Formally, the compressed action c_a is defined as follows:

- $\mathsf{pre}(c_a) = \mathsf{pre}_s(a) \cup ((\mathsf{pre}_o(a) \cup \mathsf{pre}_e(a)) \setminus \mathsf{add}_s(a))$,
- $\mathsf{add}(c_a) = (\mathsf{add}_s(a) \setminus \mathsf{del}_e(a)) \cup \mathsf{add}_e(a)$,
- $\mathsf{del}(c_a) = (\mathsf{del}_s(a) \setminus \mathsf{add}_e(a)) \cup \mathsf{del}_e(a)$.

The precondition of c_a is the union of the precondition at start of a with the preconditions over all and at end not achieved by the add effect at start. The effect of c_a is the effect at start of a followed immediately by its effect at end.

An approach to solving a sequential temporal instance P comprises three steps: 1) compiling P into a classical instance $P_c = \langle F, A_c, I, G \rangle$ where $A_c = \{c_a : a \in A\}$ is the set of compressed actions; 2) computing a solution Π' to P_c using a classical planner; and 3) *scheduling* Π' to reduce the makespan, e.g., extracting a partial order from the sequential plan (Veloso, Perez, and Carbonell 1990). Since compression may discard contexts, scheduling has to ensure that contexts are preserved. Evidently, temporal planners following this approach are incomplete: they cannot solve, nor indeed represent, temporal instances that are not sequential.

The TP Planner

In this section we describe the TP planner, which partially compiles temporal instances into classical instances, adapting the TEMPO algorithm (Cushing, Kambhampati, and Weld 2007). TP can be applied to any temporal instance, but is not always complete; we discuss the limitations regarding completeness at the end of this section.

The TEMPO Algorithm

TEMPO is based on *lifted temporal states*, i.e. tuples $\mathcal{N} = \langle s, \mathcal{A}, \tau_e, \mathcal{T} \rangle$, where s is a state on fluents, $\mathcal{A}$ is a set of *active* temporal actions (that started but not yet ended), τ_e is a time variable associated with the latest event (i.e. $\tau_e = \tau_a$ or $\tau_e = \tau_a + d(a)$, depending on whether the latest event was start_a or end_a, where time variable τ_a models the start time of a), and $\mathcal{T}$ is a set of *temporal constraints* on time variables.

To plan, TEMPO uses two successor rules: *Fattening*, for starting an action, and *Advancing Time*, for ending it. When action a starts in lifted temporal state $\mathcal{N}$, *Fattening* generates a successor $\mathcal{N}' = \langle s \oplus \mathsf{start}_a, \mathcal{A} \cup \{a\}, \tau_a, \mathcal{T} \cup \{\tau_e < \tau_a\} \rangle$. The rule updates the state as a result of applying start_a to the current state, adds a to the set of active actions and forces a to start after the latest event e. *Fattening* is applicable in $\mathcal{N}$ if the preconditions at start and over all of a hold in s and start_a does not delete the context of another active action.

When a ends in $\mathcal{N}$, *Advancing Time* generates a successor $\mathcal{N}' = \langle s \oplus \mathsf{end}_a, \mathcal{A} \setminus \{a\}, \tau_a + d(a), \mathcal{T} \cup \{\tau_e < \tau_a + d(a)\} \rangle$. The rule applies end_a, removes a from the set of active actions and forces a to end after e. *Advancing Time* is applicable in $\mathcal{N}$ if $a \in \mathcal{A}$, the precondition at end of a holds in s, and end_a does not delete the context of another active action.

TEMPO plans by starting and ending actions, updating the lifted temporal state accordingly. When TEMPO finds a sequential plan, it uses the temporal constraints to schedule the start time of each temporal action a, i.e. assign a value to τ_a. However, it might not be possible to schedule actions in a way that satisfies the temporal constraints. In theory, we would then backtrack and try different event sequences. Our approach is to schedule actions online during planning.

Compiling Temporal Actions

We simulate the Fattening and Advancing Time rules of TEMPO using classical actions start_a and end_a, respectively, compiled from the temporal action a.

Coles et al. (2009) identified three key challenges associated with compiling temporal actions into classical actions:

1. Ensure that temporal actions end before reaching the goal.
2. Ensure that contexts are not violated.
3. Ensure that temporal constraints are preserved.

Our compilation addresses the first two challenges by defining extra preconditions and effects of start_a and end_a. A potential bottleneck is the large branching factor that results from being able to start and/or end temporal actions arbitrarily. We therefore introduce a bound K on the number of active actions, i.e. the size of the set $\mathcal{A}$. The third challenge is addressed by modifying the Fast Downward planning system, as explained in the next section.

Let $P = \langle F, A, I, G\rangle$ be a temporal instance and define $F_o = \{f \in F : \exists a \in A \text{ s.t. } f \in \mathsf{pre}_o(a)\}$ as the subset of fluents that appear as contexts. We compile P into a classical instance $P_K = \langle F_K, A_K, I_K, G_K\rangle$, where the set of fluents F_K extends F with the following new fluents:

- For each $a \in A$, fluents free_a and active_a indicating that a is *free* (did not start) or *active* (started but did not end).
- For each $f \in F_o$ and $0 \le l \le K$, a fluent count^l_f indicating that l active actions have f as context.

As a result, the number of fluents of the classical instance P_K is given by $|F_K| = |F| + 2|A| + (K+1)|F_o|$.

The initial state and goal condition of P_K are defined as $I_K = I \cup \{\mathsf{count}^0_f : f \in F_o\} \cup \{\mathsf{free}_a : a \in A\}$ and $G_K = G \cup \{\mathsf{free}_a : a \in A\}$. Initially, no active actions require contexts, and temporal actions are free both initially and in the goal. The new action set A_K contains start_a and end_a for each temporal action $a \in A$, with start_a defined as

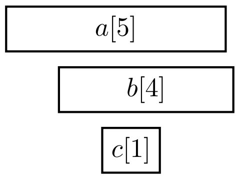
$$\begin{aligned} \mathsf{pre}(\mathsf{start}_a) &= \mathsf{pre}_s(a) \cup (\mathsf{pre}_o(a) \setminus \mathsf{add}_s(a)) \cup \{\mathsf{free}_a\} \\ &\quad \cup \{\mathsf{count}^0_f : f \in F_o \cap \mathsf{del}_s(a)\}, \\ \mathsf{add}(\mathsf{start}_a) &= \mathsf{add}_s(a) \cup \{\mathsf{active}_a\}, \\ \mathsf{del}(\mathsf{start}_a) &= \mathsf{del}_s(a) \cup \{\mathsf{free}_a\}. \end{aligned}$$

Likewise, end_a is defined as

$$\begin{aligned} \mathsf{pre}(\mathsf{end}_a) &= \mathsf{pre}_e(a) \cup \{\mathsf{active}_a\} \\ &\quad \cup \{\mathsf{count}^0_f : f \in F_o \cap \mathsf{del}_e(a)\}, \\ \mathsf{add}(\mathsf{end}_a) &= \mathsf{add}_e(a) \cup \{\mathsf{free}_a\}, \\ \mathsf{del}(\mathsf{end}_a) &= \mathsf{del}_e(a) \cup \{\mathsf{active}_a\}. \end{aligned}$$

In addition, the action start_a includes a conditional effect $\{\mathsf{count}^l_f\} \rhd \{\neg\mathsf{count}^l_f, \mathsf{count}^{l+1}_f\}$ for each $f \in \mathsf{pre}_o(a)$ and each l, $0 \le l < K$, effectively incrementing the count for f. Likewise, end_a decrements the count for each $f \in \mathsf{pre}_o(a)$.

To start a, all contexts of a not added at start should already hold. Moreover, a should be free and not delete any active context at start, i.e. the count of all delete effects at start should be 0. As a result, a is active and no longer free. To end a, a should be active and not delete any active context at end. As a result, a is free and not active.



1. $\tau_a + 0.001 \le \tau_b$,
2. $\tau_b + 0.001 \le \tau_c$,
3. $\tau_c + 0.001 \le \tau_c + 1$,
4. $\tau_c + 1 + 0.001 \le \tau_a + 5$,
5. $\tau_a + 5 + 0.001 \le \tau_b + 4$.

Figure 1: Example event sequence and induced constraints.

Preserving Temporal Constraints

Temporal constraints on time variables can be represented using *simple temporal networks*, or STNs (Dechter, Meiri, and Pearl 1991). An STN is a directed graph with time variables τ_i as nodes, and an edge (τ_i, τ_j) with label c represents a constraint $\tau_j - \tau_i \le c$. Dechter, Meiri, and Pearl showed that scheduling fails if and only if an STN contains negative cycles. Else the earliest feasible assignment to a time variable τ_i is given by the cost of the shortest path from τ_i to a reference time variable τ_0, assumed to be 0. Floyd-Warshall's shortest path algorithm can be used to compute shortest paths and test for negative cycles (i.e. whether the cost of a shortest path from a node to itself is negative).

To construct STNs we introduce a *slack unit* of time u such that the constraint $\tau_a < \tau_b$ in TEMPO becomes $\tau_a + u \le \tau_b$. As a result, each event introduces slack u, so we should set u small enough as to not interfere with the duration of actions. Throughout the paper we use $u = \min_a d(a)/1000$.

Figure 1 shows three overlapping actions a, b, and c with duration 5, 4, and 1, respectively, and the induced constraints on their associated events (the example is taken from Cushing, Kambhampati, and Weld, 2007). The fifth constraint subsumes the first, and the third is trivially satisfied. The corresponding STN contains no negative cycles, and a plan is given by $\{(a, 0), (b, 1.001), (c, 1.002)\}$, where τ_a is the reference time variable assumed to be 0 since a starts first.

Instead of attempting to model an STN and simulate Floyd-Warshall in PDDL, we implement these components as part of the Fast Downward planning system (Helmert 2006). A search node in Fast Downward contains the current state and bookkeeping information. We add a list of active actions and an STN representing the temporal constraints to the bookkeeping information, such that a search node fully represents a temporal state. Each time a compiled action is applied (either start_a or end_a for some a), the STN is updated with a single (quadratic) sweep of Floyd-Warshall, pruning search nodes for which the STN contains negative cycles, i.e. when scheduling fails.

It is easy to show that the resulting planner is sound: if the planner returns a sequential plan, the temporal actions can be scheduled in a way that preserves the event order of the plan. Cushing, Kambhampati, and Weld (2007) showed that TEMPO is complete for asynchronous events, but our implementation is incomplete for two other reasons: 1) the bound K on the number of concurrent active actions; and 2) the fact that the planner prunes duplicate states. The bookkeeping information stored in search nodes, which is where we represent part of the temporal state, is ignored when checking for duplicate states. The first problem could be addressed by iteratively increasing the bound K. A possible solution to the second problem is to rerun the search without pruning duplicate states. In experiments, the planner either always returned a plan or ran out of memory, never reporting that a problem was unsolvable. This suggests but does not prove that pruning duplicate states is not a serious issue in practice.

The TPSHE Planner

In this section we introduce the TPSHE planner which, unlike TP, fully compiles temporal instances into classical instances. TPSHE can be applied to any temporal instance, and can solve temporal instances that are not sequential as long as the following assumptions hold:

1. The only required concurrency is in the form of *single hard envelopes* (Coles et al. 2009).
2. Temporal actions are instantiated from *temporal action templates* (as in PDDL), each with a fixed duration.
3. One single hard envelope is sufficient to satisfy each context of an associated content action.

We first define single hard envelopes, then introduce the concept of a *concurrency graph*, and finally describe TPSHE.

Single Hard Envelopes

To define single hard envelopes we first introduce the notion of a *temporary resource*, or resource for short. Given a temporal planning instance P, a resource is a fluent $f \in F$ such that $f \notin I$ and one of the following holds for each $a \in A$:

1. f does not appear in any effect of a, or
2. $f \in \mathsf{add}_s(a) \cap \mathsf{del}_e(a)$.

An action a of the second type is a *producer* of f, adding f at start and deleting f at end. Consequently, a resource f is only available during the execution of one of its producers.

Coles et al. (2009) define a single hard envelope as a temporal action a that adds a fluent f at start and deletes it at end, i.e. $f \in \mathsf{add}_s(a) \cap \mathsf{del}_e(a)$. We extend the definition in two ways. First, we require f to be a resource, i.e. a is a producer of f. Second, there has to exist another action b (the *content*) with shorter duration than a that has f as context, i.e. $d(b) < d(a)$ and $f \in \mathsf{add}_s(a) \cap \mathsf{del}_e(a) \cap \mathsf{pre}_o(b)$.

Concurrency Graph

We use the (single hard) envelopes of a temporal instance to construct a *concurrency graph*, based on assumption 2 above. The concurrency graph $G = (V, E)$ has the temporal action templates as nodes, and an edge $(n_1, n_2) \in E$ indicates that each action b instantiated from n_2 is the content of some envelope a instantiated from n_1. Figure 2 shows the resulting concurrency graph of the TMS domain. This graph indicates, among other things, that fire-kiln1 actions are envelopes for bake-ceramic3 and bake-structure actions.

Since contents are completely contained in envelopes, the edges of G represent a nesting relation among actions. Our idea is to use a stack to simulate the execution of such nested actions. When an action starts, we push it onto the stack, and when an action ends, we pop it from the stack. We use G to infer on what levels of the stack each action should appear.

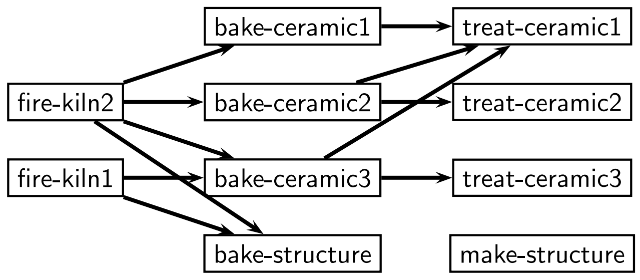


Figure 2: The concurrency graph of the TMS domain.

Let L be the depth of G, i.e. the maximum length of any directed path. L is well-defined since G is acyclic due to the condition $d(b) < d(a)$ on envelopes. For each action a, let $L(a) \subseteq \{0, \ldots, L\}$ be the set of levels of a, i.e. lengths of incoming directed paths to the associated template. In Figure 2, $L = 2$, and both fire-kiln1 and make-structure actions have level 0, while treat-ceramic2 actions have level 2.

Compiling Single Hard Envelopes

Our compilation models single hard envelopes using a stack of active actions, i.e. that started but not yet ended. Only envelope actions and their remaining duration are stacked; all other actions are compiled into compressed actions. This construction is based on assumption 3 above: only one envelope is needed to provide the contexts of each content.

We first precompute all combinations of possible remaining durations of envelopes. Each envelope a may have multiple contents; let $\{d_1, \ldots, d_k\}$ be the set of durations of contents of a. The set of *remaining durations* of a is $D(a) = \{d(a) - \sum_i w_i d_i : \forall i, w_i \in \mathbb{N} \wedge \sum_i w_i d_i < d(a)\}$. We define $D(a) = \{d(a)\}$ for non-envelopes and $D = \bigcup_a D(a)$.

Formally, a temporal instance $P = \langle F, A, I, G\rangle$ is compiled into a classical instance $P_{SHE} = \langle F', A', I', G'\rangle$. The set F' includes fluents in F plus the following extra fluents:

- For each envelope $a \in A$ and level $l \in L(a)$, a fluent active_a^l indicating that a is active on level l of the stack.
- For each $f \in F_o$ and $0 \leq l \leq L$, a fluent count_f^l indicating that l active actions have f as context.
- For each $0 \leq l \leq L$, a fluent stack^l indicating that the stack contains l active actions.
- For each $d \in D$ and $0 \leq l < L$, a fluent rem_d^l indicating that the active action on level l has remaining duration d.
- For each duration pair $d, e \in D$ such that $d - e \in D$, a static fluent $\mathsf{sub}(d, e, d - e)$ representing subtraction.

The total number of fluents of P_{SHE} is bounded by $|F'| \leq |F| + L|A| + (L + 1)(|F_o| + 1) + L|D| + \frac{|D|(|D|-1)}{2}$.

The initial state represents an empty stack and initializes counts and subtractions, i.e. $I' = I \cup \{\mathsf{stack}^0\} \cup \{\mathsf{count}_f^0 : f \in F_o\} \cup \{\mathsf{sub}(d, e, d - e) : d, e \in D \text{ s.t. } d - e \in D\}$.

The goal condition verifies that goals are achieved and that no actions are still executing, i.e. $G' = G \cup \{\mathsf{stack}^0\}$.

For each envelope $a \in A$, each level $l \in L(a)$ and each pair of durations $d, e \in D$, A' contains two classical actions $\mathsf{push}_a^l(d, e)$ and $\mathsf{pop}_a^l(d)$. Action $\mathsf{push}_a^l(d, e)$ pushes a onto level l of the stack and is defined as

$$\begin{aligned}
\mathsf{pre}(\mathsf{push}_a^l(d,e)) &= \mathsf{pre}_s(a) \cup (\mathsf{pre}_o(a) \setminus \mathsf{add}_s(a)) \\
&\quad \cup \{\mathsf{stack}^l, \mathsf{rem}_d^{l-1}, \mathsf{sub}(d, e, d(a))\} \\
&\quad \cup \{\mathsf{count}_f^0 : f \in F_o \cap \mathsf{del}_s(a)\}, \\
\mathsf{add}(\mathsf{push}_a^l(d,e)) &= \mathsf{add}_s(a) \\
&\quad \cup \{\mathsf{stack}^{l+1}, \mathsf{rem}_e^{l-1}, \mathsf{active}_a^l, \mathsf{rem}_{d(a)}^l\}, \\
\mathsf{del}(\mathsf{push}_a^l(d,e)) &= \mathsf{del}_s(a) \cup \{\mathsf{stack}^l, \mathsf{rem}_d^{l-1}\}.
\end{aligned}$$

If $l > 0$, d and e are used to subtract $d(a)$ from the remaining duration of the active action on level $l - 1$. If $l = 0$, parameters d and e and associated preconditions and effects can be removed. The stack size is incremented and a marked as active on level l, setting its remaining duration to $d(a)$. As before, $\mathsf{push}_a^l(d, e)$ increments the count for each $f \in \mathsf{pre}_o(a)$.

Action $\mathsf{pop}_a^l(d)$ pops a from level l of the stack:

$$\begin{aligned}
\mathsf{pre}(\mathsf{pop}_a^l(d)) &= \mathsf{pre}_e(a) \cup \{\mathsf{stack}^{l+1}, \mathsf{active}_a^l, \mathsf{rem}_d^l\} \\
&\quad \cup \{\mathsf{count}_f^0 : f \in F_o \cap \mathsf{del}_e(a)\}, \\
\mathsf{add}(\mathsf{pop}_a^l(d)) &= \mathsf{add}_e(a) \cup \{\mathsf{stack}^l\}, \\
\mathsf{del}(\mathsf{pop}_a^l(d)) &= \mathsf{del}_e(a) \cup \{\mathsf{stack}^{l+1}, \mathsf{active}_a^l, \mathsf{rem}_d^l\}.
\end{aligned}$$

Popping a is only possible if a is active on level l and on top of the stack. As a result, the stack size is decremented, a is no longer active and has no remaining duration. As before, $\mathsf{pop}_a^l(d, e)$ decrements the count for each $f \in \mathsf{pre}_o(a)$.

For each action $a \in A$ that is not an envelope, each level $l \in L(a)$ and each pair of durations $d, e \in D$, the set A' contains a compressed action $c_a^l(d, e)$ defined as

$$\begin{aligned}
\mathsf{pre}(c_a^l(d,e)) &= \mathsf{pre}(c_a) \cup \{\mathsf{stack}^l, \mathsf{rem}_d^{l-1}, \mathsf{sub}(d, e, d(a))\} \\
&\quad \cup \{\mathsf{count}_f^0 : f \in F_o \cap \mathsf{del}(c_a)\}, \\
\mathsf{add}(c_a^l(d,e)) &= \mathsf{add}(c_a) \cup \{\mathsf{rem}_e^{l-1}\}, \\
\mathsf{del}(c_a^l(d,e)) &= \mathsf{del}(c_a) \cup \{\mathsf{rem}_d^{l-1}\}.
\end{aligned}$$

Here, c_a is the compressed action of temporal action a. As before, if $l > 0$, d and e are used to subtract $d(a)$ from the remaining duration of the active action on level $l - 1$.

We show that the proposed compilation is sound. Given a plan for the classical instance P_{SHE}, we first schedule all actions at level 0 of the stack in sequence. We then schedule each action a at level 1 or higher either immediately after its envelope started, or immediately after the previous action ended (in case another action was applied inside the same envelope). The resulting plan is valid since the induced event sequence is the same as in the solution to P_{SHE} (thus reaching the goal G), and the remaining duration of an envelope cannot fall below 0 due to preconditions of type sub.

Implementation

In this section we describe implementation issues and optimizations for TPSHE. Although stack levels and durations could be represented using numeric fluents, planners that support numeric fluents performed poorly on P_{SHE} in testing. Instead, we represent stack levels as objects, associate each duration $d \in D$ with a unique time object and replace durations with time objects in fluents $\mathsf{sub}(d, e, d - e)$ and rem_d^l and actions $\mathsf{push}_a^l(d, e)$, $\mathsf{pop}_a^l(d)$ and $c_a^l(d, e)$.

In some domains, the parameters d and e of actions $\mathsf{push}_a^l(d, e)$, $\mathsf{pop}_a^l(d)$ and $c_a^l(d, e)$, $l > 0$, cause a significant blowup in the number of actions. To reduce the number of actions, we split all such actions into two parts. The second part of the action is only responsible for updating the remaining duration of active actions, while the first part is responsible for all other effects. The key property is that the first part no longer needs duration parameters. To control the sequential application of each new pair of actions generated this way, we introduce an intermediate fluent that is an add effect of the first and a precondition and delete effect of the second. In some domains, the number of actions is reduced by more than an order of magnitude due to this optimization.

Finally, a context of an envelope is only threatened if one of its contents deletes the context. We can thus traverse the concurrency graph to test whether a context f is threatened. If not, there is no need to maintain a count for f, nor the associated preconditions and effects.

Separable Temporal Instances

In this section we introduce two novel classes, $\textsc{Sep}_s$ and $\textsc{Sep}_e$, of temporal planning instances that are provably (inherently) sequential. Intuitively, the idea is to identify conditions under which it is always possible to transform a parallel plan into a sequential plan by repeatedly separating a temporal action from the remaining actions.

Our definitions of $\textsc{Sep}_s$ and $\textsc{Sep}_e$ are based on *separability* and *mutual exclusion*, which we define in turn.

Definition 1 *Let P be a temporal instance. Action $a \in A$ is* separable at start *from $b \in A$ if all the following holds:*

1. $\mathsf{pre}_e(a) \cap \mathsf{add}_s(b) = \emptyset$*,*
2. $\mathsf{del}_e(a) \cap \mathsf{pre}_s(b) = \emptyset$*,*
3. $\mathsf{del}_e(a) \cap \mathsf{add}_s(b) = \emptyset$ *and* $\mathsf{add}_e(a) \cap \mathsf{del}_s(b) = \emptyset$*,*
4. If $d(b) < d(a)$*,*
  (a) $\mathsf{pre}_e(a) \cap \mathsf{add}_e(b) = \emptyset$*,*
  (b) $\mathsf{del}_e(a) \cap (\mathsf{pre}_o(b) \cup \mathsf{pre}_e(b)) = \emptyset$*,*
  (c) $\mathsf{del}_e(a) \cap \mathsf{add}_e(b) = \emptyset$ *and* $\mathsf{add}_e(a) \cap \mathsf{del}_e(b) = \emptyset$*.*

Intuitively, if a and b appear in parallel with a starting first, Conditions 1–4 ensure that we can reschedule end_a before start_b and end_b such that the partial event sequence on a and b becomes $\langle \mathsf{start}_a, \mathsf{end}_a, \mathsf{start}_b, \mathsf{end}_b \rangle$.

Definition 2 *Let P be a temporal instance. Action a is* separable at end *from b if all the following holds:*

5. $\mathsf{pre}_s(a) \cap \mathsf{del}_e(b) = \emptyset$*,*
6. $\mathsf{add}_s(a) \cap \mathsf{pre}_e(b) = \emptyset$*,*
7. $\mathsf{del}_s(a) \cap \mathsf{add}_e(b) = \emptyset$ *and* $\mathsf{add}_s(a) \cap \mathsf{del}_e(b) = \emptyset$*,*

8. If $d(b) < d(a)$,

(a) $\mathsf{pre}_s(a) \cap \mathsf{del}_s(b) = \emptyset$,

(b) $\mathsf{add}_s(a) \cap (\mathsf{pre}_s(b) \cup \mathsf{pre}_o(b)) = \emptyset$,

(c) $\mathsf{del}_s(a) \cap \mathsf{add}_s(b) = \emptyset$ *and* $\mathsf{add}_s(a) \cap \mathsf{del}_s(b) = \emptyset$.

Intuitively, if a ends last, Conditions 5–8 ensure that we can reschedule start_a after start_b and end_b to achieve the partial event sequence $\langle \mathsf{start}_b, \mathsf{end}_b, \mathsf{start}_a, \mathsf{end}_a \rangle$.

Definition 3 *Let P be a temporal instance. Two temporal actions a and b are* mutually exclusive *if they cannot overlap in a plan Π for P. For each copy $(a, t_a) \in \Pi$ of a, each copy $(b, t_b) \in \Pi$ of b satisfies $t_b + d(b) < t_a$ or $t_a + d(a) < t_b$, i.e. b has to end before a starts or start after a ends.*

One way to detect mutually exclusive temporal actions is to identify *mutex invariants*. In classical planning, a mutex invariant is a subset of fluents $C \subseteq F$ such that at most one is true at any moment, i.e. $|I \cap C| = 1$ and each action that adds a fluent in C deletes another. We modify the definition for temporal planning as follows. A subset $C \subseteq F$ is a mutex invariant if and only if $|I \cap C| = 1$ and for each $a \in A$ it holds that $\mathsf{add}_s(a) \cap C = \mathsf{del}_e(a) \cap C = \emptyset$ and either

1. $\mathsf{del}_s(a) \cap C = \mathsf{add}_e(a) \cap C = \emptyset$, or
2. $|\mathsf{pre}_s(a) \cap \mathsf{del}_s(a) \cap C| = 1$ and $|\mathsf{add}_e(a) \cap C| = 1$.

While an action of the first type has no effect on fluents in C, an action of the second type is a *modifier* of C, requiring a fluent in C as precondition at start, deleting the same fluent at start, and adding a fluent in C at end. Note that during the execution of a modifier, *no* fluent in C is true, thus preventing other modifiers of C from starting. Hence all modifiers of C are mutually exclusive. We envision that there are other ways to detect mutually exclusive temporal actions.

We are now ready to define the class $\textsc{Sep}_s$, and show that temporal instances in this class are (inherently) sequential.

Definition 4 *A temporal instance P belongs to the class $\textsc{Sep}_s$ if and only if, for each pair $(a, b) \in A$, either a and b are mutually exclusive or a is separable at start from b.*

Lemma 5 *Each instance $P \in \textsc{Sep}_s$ is inherently sequential.*

Proof Let Π be an arbitrary plan solving P and let $\Pi_e = \langle e_1, \ldots, e_{2n} \rangle$ be the induced event sequence. Let a be the temporal action that starts first, i.e. $e_1 = \mathsf{start}_a$, and let i be the index of end_a. We first show that the event sequence $\langle e_1, e_i, e_2, \ldots, e_{i-1}, e_{i+1}, \ldots, e_{2n} \rangle$ solves P. We can thus reschedule end_a immediately after start_a. We then show that repeatedly applying this process results in a sequential plan.

Consider another action $b \in A$. If a and b are mutually exclusive, events $e_2, \ldots, e_{i-1}$ cannot involve b, else a and b would overlap in Π, leading to a contradiction. Else the definition of $\textsc{Sep}_s$ implies that a is separable at start from b. In this case, start_b could be among $e_2, \ldots, e_{i-1}$, as well as end_b if $d(b) < d(a)$. If $d(b) \geq d(a)$ there is not enough time to start and end b during the execution of a.

We now show that the sequence $\langle e_1, e_i, e_2, \ldots, e_{i-1} \rangle$ is applicable in I and results in the same state as $\langle e_1, \ldots, e_i \rangle$.

- Let f be a precondition at end of a. Due to Conditions 1 and 4.a of Definition 1, $e_2, \ldots, e_{i-1}$ cannot add f. Since f holds in $I \oplus e_1 \oplus \cdots \oplus e_{i-1}$, f also holds in $I \oplus e_1$.
- Let f be a precondition of e_j, $2 \leq j < i$. Due to Conditions 2 and 4.b of Definition 1, a cannot delete f at end. Since f holds in $I \oplus e_1 \oplus \cdots \oplus e_{j-1}$, f also holds in $I \oplus e_1 \oplus e_i \oplus e_2 \cdots \oplus e_{j-1}$. If $e_j = \mathsf{end}_b$ for some b, Condition 4.b also ensures that no context of b is deleted.
- Conditions 3 and 4.c ensure that no effect at end of a is "undone" by an event among $e_2, \ldots, e_{i-1}$. Likewise, no effect of $e_2, \ldots, e_{i-1}$ is undone by e_i. Thus it holds that $I \oplus e_1 \oplus e_i \oplus e_2 \oplus \cdots \oplus e_{i-1} = I \oplus e_1 \oplus \cdots \oplus e_i$.

It is easy to reschedule the actions in Π to form a new plan Π' whose event sequence is $\langle e_1, e_i, e_2, \ldots, e_{i-1}, e_{i+1}, \ldots, e_n \rangle$: simply add $d(a)$ to the start time of each action except a.

Once we have separated action a we can repeat the procedure for the remaining actions. Definition 1 guarantees that we can always separate the action starting first among the remaining actions. Applying this process repeatedly will eventually result in a sequential plan solving P. $\square$

The definition of the class $\textsc{Sep}_e$ is analogous:

Definition 6 *A temporal instance P belongs to the class $\textsc{Sep}_e$ if and only if, for each pair $(a, b) \in A$, either a and b are mutually exclusive or a is separable at end from b.*

Lemma 7 *Each instance $P \in \textsc{Sep}_e$ is inherently sequential.*

Proof sketch Analogous to the proof of Lemma 5, but instead of separating out the action a starting first from a given parallel plan, we separate out the action a ending last. $\square$

Cushing, Kambhampati, and Weld (2007) defined a family $L_{\text{eff}}^{\text{pre}}$ of temporal planning languages by restricting the preconditions and effects of actions to the types in pre and eff (e.g. L_e^s only admits actions with preconditions at start and effects at end). The authors defined a topology of temporal planning languages and showed that $L_s, L_s^s, L_s^o, L_s^{s,o}$ and $L_e, L_e^e, L_e^o, L_e^{o,e}$ are temporally simple, i.e. all instances expressed in these languages are inherently sequential.

We show that temporal planning instances expressed in $L_s, L_s^s, L_s^o, L_s^{s,o}$ belong to the class $\textsc{Sep}_s$ and that those expressed in $L_e, L_e^e, L_e^o, L_e^{o,e}$ belong to the class $\textsc{Sep}_e$. Conditions 1–3 of Definition 1 and Conditions 5–7 of Definition 2 hold for any planning instance expressed in these languages because they relate *at start* components with *at end* components. In addition, in the case of $L_s, L_s^s, L_s^o, L_s^{s,o}$, Conditions 4.a–4.c of Definition 1 also hold since they refer to at end components. Likewise, in the case of $L_e, L_e^e, L_e^o, L_e^{o,e}$, Conditions 8.a–8.c of Definition 2 refer to at start components.

Our definitions of $\textsc{Sep}_s$ and $\textsc{Sep}_e$ provide additional insight, enabling us to detect (inherently) sequential temporal instances despite the fact that they are represented in temporally expressive languages. In particular, any instance belonging to the most expressive temporal language $L_{s,e}^{s,o,e}$ is identified as (inherently) sequential if either the $\textsc{Sep}_s$ or $\textsc{Sep}_e$ conditions hold for each pair of actions.

Extension to Single Hard Envelopes

In this section we define two classes SHE_s^1 and SHE_e^1 of temporal instances, similar to the classes $\textsc{Sep}_s$ and $\textsc{Sep}_e$. We show that the TPSHE planner is complete for instances in SHE_s^1 and SHE_e^1. Note that our definition of a single hard envelope violates Conditions 4.b and 8.b of Definitions 1 and 2. Hence, as expected, temporal instances with envelopes belong to neither $\textsc{Sep}_s$ nor $\textsc{Sep}_e$.

Unlike the class $\textsc{Sep}_s$, the class SHE_s^1 admits single hard envelopes. Given a content $b \in A$, let $\mathsf{env}(b) \subseteq \mathsf{pre}_o(b)$ be the subset of contexts of b provided by envelopes. To define SHE_s^1 we first modify Condition 4.b of Definition 1:

4(b) $\mathsf{del}_e(a) \cap ((\mathsf{pre}_o(b) \setminus \mathsf{env}(b)) \cup \mathsf{pre}_e(b)) = \emptyset$.

Hence for a to be separable at start from b, a is allowed to delete contexts in $\mathsf{env}(b)$ at end. This is possible since we will ensure that b remains inside another envelope that provides the contexts in $\mathsf{env}(b)$.

The superscript 1 in SHE_s^1 indicates that a single envelope is sufficient to provide the contexts in $\mathsf{env}(b)$ (cf. assumption 3 for TPSHE). Formally, SHE_s^1 is defined as follows:

Definition 8 *A temporal instance P belongs to the class SHE_s^1 if and only if a single envelope is sufficient to provide the contexts in* $\mathsf{env}(b)$ *of each content $b \in A$ and one of the following holds for each pair $(a, b) \in A$:*

1. *Actions a and b are mutually exclusive,*
2. *Action a is not a content of any envelope and is separable at start from b,*
3. *Action a is a content of some envelope and can be rescheduled before b, i.e. we can always order* start_a *and* end_a *before* start_b*, even if* start_a *occurs after* end_b.

For space reasons we do not formalize Condition 3, which is stronger than separability at start. For a to be separable at start from b, we only require the end of a to be rescheduled before b. On the contrary, Condition 3 requires the start of a to be rescheduled before b, even if a starts after b ends. In other words, if the partial event sequence on a and b is $\langle \mathsf{start}_b, \mathsf{end}_b, \mathsf{start}_a, \mathsf{end}_a \rangle$, we can reschedule a and b such that the event sequence becomes $\langle \mathsf{start}_a, \mathsf{end}_a, \mathsf{start}_b, \mathsf{end}_b \rangle$, without affecting the applicability of these and other events.

Lemma 9 *TPSHE is complete for instances in* SHE_s^1.

Proof sketch Given an instance $P \in \mathrm{SHE}_s^1$, let Π be an arbitrary plan solving P. For Π to solve P, each content has to be completely contained inside at least one of its envelopes. Assign each content to one such envelope, such that each envelope has an associated set of contents.

Similarly to the proof of Lemma 7 we can separate the temporal action starting first from the remaining actions. However, if the action starting first is an envelope, we separate it together with all its contents. In doing so, we might have to reschedule a content action before other events, which is safe due to Condition 3 of Definition 8.

Consequently, each plan for P has a sequential form in which concurrency is only in the form of contents nested inside envelopes. This is precisely the kind of solution that TPSHE models. In particular, if such a solution exists, it will also be a solution to the compiled instance P_{SHE}. □

For space reasons we omit the definition of SHE_e^1. Note that although TPSHE is not provably complete for instances outside SHE_s^1 and SHE_e^1, we can always apply the compilation: if there are no single hard envelopes, all temporal actions are simply compiled into compressed actions.

Results

We performed an evaluation in all 10 domains of the temporal track of IPC-2014: seven with sequential instances, and three (MATCHCELLAR, TMS, TURN&OPEN) that involve single hard envelopes. Moreover, we added the *DriverLog Shift* (DLS) domain (Coles et al. 2009) that includes single hard envelopes, a domain based on Allen's interval algebra (AIA) (Allen 1983) that models relations between intervals, and a domain based on the example in Figure 1 (EXAMPLE).

To classify a temporal instance P, we ran Fast Downward's translator on the compressed instance P_c to identify (classical) mutex invariants. We used these to identify temporal mutex invariants for the original temporal instance P, and marked all modifiers of a mutex invariant as mutually exclusive. We then checked membership in the separable classes. As it turns out, instances of DRIVERLOG, FLOORTILE, and PARKING provably belong to the class $\textsc{Sep}_s$, and instances of MATCHCELLAR belong to SHE_s^1. Instances of other domains do not fall into any of the proposed classes.

We ran the TP planner with three bounds on the number of active actions: $K = 1$, $K = 2$ and $K = 3$. The TPSHE planner used the LAMA-2011 setting of Fast Downward to solve the compiled instance. We compared our planners with the following planners from IPC-2014 (Chrpa, Vallati, and McCluskey 2014): ITSAT, the best performer in domains that required concurrency, the temporal version of Fast-Downward (TDF), and YAHSP3-MT, the best performer in sequential domains (and overall winner). We also included POPF2, the runner-up at IPC-2011. Table 1 shows, for each planner, the IPC quality score and the coverage, i.e. the number of instances solved per domain. Experiments were run on a cluster of Linux AMD Opteron processors at 2.4 GHz, with a cutoff of 1h or 4.2GB of RAM memory.

The TPSHE planner solved at least one instance in all but one domain, and obtained the highest IPC score and coverage. We remark that we were unable to reproduce the results of YAHSP3-MT from IPC-2014, where it solved more instances in DRIVERLOG, FLOORTILE and STORAGE. Even so, the coverage of TPSHE on domains from IPC-2014 was 155, compared to 97 of YAHSP3-MT, the winner.

The only domains with required concurrency *not* in the form of single hard envelopes are AIA and EXAMPLE. Specifically, three instances of AIA require events to occur in parallel; these were not solved by any planner. As previously mentioned, TP and TPSHE can only handle asynchronous events. As expected, TP(1) and TPSHE could not solve any instance in EXAMPLE, but TP(2) and TP(3) solved all of them. The additional machinery of TP made it comparably slow in many other domains, however.

Regarding quality, TPSHE generates comparably long plans in many domains, especially TMS and DLS where ITSAT is able to exploit parallelism to generate much shorter plans. This is due to two issues: 1) TPSHE completely ignores duration when solving an instance, and 2) our stack-based compilation does not enable actions to execute in parallel inside an envelope. In the future we plan to address the first issue by assigning costs to actions that depend on the duration. ITSAT also generates very long plans in some domains, however, notably PARKING and AIA.

	TP(1)	TP(2)	TP(3)	TPSHE	ITSAT	POPF2	TFD	YAHSP3-MT
DRIVERLOG[20]	1.18/4	0.98/3	0.96/3	**18.26/20**	1/1	-	-	1.68/4
FLOORTILE[20]	-	-	-	4.92/5	**19.47/20**	-	-	1.69/2
MAPANALYSER[20]	-	-	-	13.84/**20**	-	-	**15.85**/17	15.24/**20**
MATCHCELLAR[20]	-	-	-	15.79/**20**	19/19	-	**20/20**	-
PARKING[20]	5.43/**20**	5.43/**20**	5.43/**20**	9.35/**20**	1.29/6	1/1	13.01/**20**	**19.14/20**
RTAM[20]	-	-	-	12.21/15	-	-	-	**17.57/20**
SATELLITE[20]	6.35/17	5.44/14	4.54/12	**16.64**/18	-	1/1	14.35/17	13.70/**20**
STORAGE[20]	5.26/**9**	-	-	**9/9**	-	-	-	-
TMS[20]	-	-	-	0.06/9	**18/18**	-	-	-
TURN&OPEN[20]	-	-	-	**14.03/19**	6.57/7	3.72/4	13.16/18	-
DLS[20]	-	2.51/7	2.51/7	2.77/13	**20/20**	-	1/1	-
AIA[9]	2.09/3	3.26/**6**	3.26/**6**	2.22/3	0.74/6	-	**6/6**	2.22/3
EXAMPLE[20]	-	17.28/**20**	**18.60/20**	-	-	-	2.86/3	-
TOTAL	20.31/53	34.90/70	35.30/68	**119.09/171**	86.07/97	5.72/6	86.23/102	71.24/89

Table 1: IPC quality score / coverage per domain for each planner. Total number of instances of each domain within brackets.

Related work

Several authors have proposed compiling temporal planning actions into multiple classical actions. An early approach, LPGP (Long and Fox 2003), turned out to be unsound and incomplete since it failed to address the three challenges identified by Coles et al. (2009), outlined above. Cooper, Maris, and Régnier (2013) provided theoretical justification for splitting durative actions into classical actions.

Certainly, the planners most similar to ours are CRIKEY and CRIKEY$_{SHE}$ (Coles et al. 2009), as well as their successors, POPF (Coles et al. 2010) and OPTIC (Benton, Coles, and Coles 2012). CRIKEY maintains a *set* of envelopes and associates an STN with each envelope. Each envelope (and its associated STN) are potentially updated each time a new action is chosen. Our approach has two main advantages over CRIKEY. First, compiling as much as possible of the temporal instance into classical planning enables the application of well-studied heuristics. Second, maintaining a single STN (as opposed to multiple STNs) and letting the planner handle interactions automatically appears simpler.

Just like our compilation, CRIKEY$_{SHE}$ only handles required concurrency in the form of single hard envelopes. Unlike our compilation, however, CRIKEY$_{SHE}$ is based on the same machinery as CRIKEY, using STNs to detect unsatisfiable temporal constraints. In contrast, our compilation performs this test directly in the encoding, using the precondition $\{\mathsf{rem}_d^{l-1}, \mathsf{sub}(d, e, d(a))\}$ to test whether the remaining duration d of an envelope a' is sufficiently large to accommodate an action a with duration $d(a)$ inside a'. Compiling the entire temporal instance into classical planning makes it possible to use any planner to solve the instance.

Rintanen (2007) proposed another compilation from temporal to classical planning that explicitly represents time units as objects. The compilation includes classical actions that start temporal actions, and keeps track of time elapsed in order to determine when temporal actions should end. The compilation only handles integer durations, potentially making the planner incomplete when events have to be scheduled fractions of time units apart. As far as we know, the compilation has never been implemented as part of an actual planner.

Conclusion

This work makes several contributions to temporal planning. First, we propose a planner, TP, that adapts the TEMPO algorithm (Cushing, Kambhampati, and Weld 2007) by partially compiling temporal actions into classical planning. We also introduce a novel compilation from temporal planning with single hard envelopes to classical planning, thereby devising a planner, TPSHE, competitive with state-of-the-art temporal planners. Another contribution is identifying novel classes of temporal instances that are provably sequential.

Although we cannot safely claim that TP subsumes TPSHE, TP can handle forms of required concurrency that TPSHE cannot. As results from the IPC show, however, incomplete planners can exploit the structure of temporal instances, which often pays off in terms of performance. TPSHE can be seen as an extension of purely sequential planners to handle the specific case of single hard envelopes.

Our vision of a temporal planner is to first classify a temporal instance according to several criteria, and then apply an appropriate algorithm or compilation. We believe that our notions of separability and mutual exclusion are first steps in this direction, and that it should be possible to modify our criteria to classify more benchmark domains as sequential.

One disadvantage of our planners is that they ignore information about duration. In the future we will experiment with assigning costs proportional to duration to compiled STRIPS actions, which may reduce the makespan in some cases (but will likely still not achieve optimal makespan). Another possible extension is to compile parallel events into classical planning, which we believe could be done in a manner similar to how conditional effects are handled by planners.

Acknowledgment

Sergio Jiménez is partially supported by the *Juan de la Cierva* program funded by the Spanish government.

References

Allen, J. F. 1983. Maintaining knowledge about temporal intervals. *Commun. ACM* 26(11):832–843.

Benton, J.; Coles, A. J.; and Coles, A. I. 2012. Temporal planning with preferences and time-dependent continuous costs. In *Proceedings of the Twenty Second International Conference on Automated Planning and Scheduling (ICAPS-12)*.

Chen, Y.; Wah, B.; and Hsu, C.-W. 2006. Temporal Planning Using Subgoal Partitioning and Resolution in SGPlan. *J. Artif. Int. Res.* 26(1):323–369.

Chrpa, L.; Vallati, M.; and McCluskey, L. 2014. University of huddersfield. *Booklet International planning competition*.

Coles, A.; Fox, M.; Halsey, K.; Long, D.; and Smith, A. 2009. Managing concurrency in temporal planning using planner-scheduler interaction. *Artificial Intelligence* 173(1):1 – 44.

Coles, A. J.; Coles, A. I.; Fox, M.; and Long, D. 2010. Forward-chaining partial-order planning. In *Proceedings of the Twentieth International Conference on Automated Planning and Scheduling (ICAPS-10)*.

Coles, A. J.; Coles, A. I.; Garcia, A.; Jiménez, S.; Linares, C.; Sanner, S.; and Yoon, S. 2012. A survey of the seventh international planning competition. *AI Magazine* 33(1):83–88.

Cooper, M.; Maris, F.; and Régnier, P. 2013. Managing Temporal Cycles in Planning Problems Requiring Concurrency. *Computational Intelligence* 29(1):111–128.

Cushing, W.; Kambhampati, S.; and Weld, D. 2007. When is temporal planning really temporal. In *In IJCAI*.

Dechter, R.; Meiri, I.; and Pearl, J. 1991. Temporal Constraint Networks. *Artificial Intelligence* 49:61–95.

Fox, M., and Long, D. 2003. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *J. Artif. Int. Res.* 20(1):61–124.

Helmert, M. 2002. Decidability and Undecidability Results for Planning with Numerical State Variables. In *Proceedings of the 6th International Conference on Artificial Intelligence Planning Systems (AIPS)*, 44–53.

Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research* 26:191–246.

Long, D., and Fox, M. 2003. Exploiting a graphplan framework in temporal planning. In *ICAPS*, 52–61.

Rintanen, J. 2007. Complexity of Concurrent Temporal Planning. In *ICAPS*, 280–287.

Veloso, M.; Perez, A.; and Carbonell, J. 1990. Nonlinear planning with parallel resource allocation. In *In Proceedings of the DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control*, 207–212. Morgan Kaufmann.

Vidal, V. 2014. YAHSP3 and YAHSP3-MT in the 8th International Planning Competition. In *Proceedings of the 8th International Planning Competition (IPC-2014)*, 64–65.