Forward-Search Temporal Planning with Simultaneous Events

Daniel Furelos-Blanco and **Anders Jonsson**
Department of Information and Communication Technologies
Universitat Pompeu Fabra
Roc Boronat 138, 08018 Barcelona, Spain
{daniel.furelos, anders.jonsson}@upf.edu

Héctor Palacios
Nuance Communications
Montreal, Canada
hector.palaciosverdes@nuance.com

Sergio Jiménez
Department of Computer Systems and Computation
Universitat Politècnica de València
Camino de Vera s/n. 46022 Valencia, Spain
serjice@dsic.upv.es

Abstract

In this paper we describe STP, a novel algorithm for temporal planning. Similar to several existing temporal planners, STP relies on a transformation from temporal planning to classical planning, and constructs a temporal plan by finding a sequence of classical actions that solve the problem while satisfying a given set of temporal constraints. Our main contribution is that STP can solve temporal planning problems that require simultaneous events, i.e. the temporal actions have to be scheduled in such a way that two or more of their effects take place concurrently. To do so, STP separates each event into three phases: one phase in which temporal actions are scheduled to end, one phase in which simultaneous effects take place, and one phase in which temporal actions are scheduled to start. Experimental results show that STP significantly outperforms state-of-the-art temporal planners in a domain requiring simultaneous events.

Introduction

How expressive can a forward-search temporal planner be? The third *International Planning Competition* (IPC) introduced the most common language for modeling temporal planning problems, PDDL 2.1 (Fox and Long 2003). PDDL 2.1 is compatible with classical planning problems, and its semantics are defined in terms of the semantics of classical actions. This connection was studied further by Rintanen (2007), who proved that plan existence for temporal planning with succinct models is EXPSPACE-complete. As a motivation, Rintanen mentioned a basic reduction from temporal planning to classical planning called TEMPO (Cushing et al. 2007), and proposed a restriction to PDDL 2.1 that is reducible to classical planning, implying a decrease in complexity from EXPSPACE to PSPACE.

Later, Jiménez, Jonsson, and Palacios (2015) built on this idea and provided an effective implementation of TEMPO relying on a simple modification of a classical planner. In contrast to Cushing et al. (2007) and later work, Jiménez, Jonsson, and Palacios dealt with more expressive forms of concurrency, i.e. concurrency does not only occur in the form of single hard envelopes.

Rintanen (2007) argued that the complexity of temporal planning remains in PSPACE if we avoid an unbounded dependency on past and present information for determining the next temporal state. His restrictions prohibit a temporal action from executing in parallel with itself, and assumes that time is discrete.

In this work we explore further the expressivity of classical planning for solving complex temporal problems, focusing on the case of *simultaneous events* in which the effects of temporal actions take place concurrently. Many situations in the real-world involve simultaneous events. A clear example are relay races where a runner gives the relay at the same time that another runner receives it. This scenario, for instance, could be translated into an assembly line where robotic arms give and receive mechanical pieces.

Given the importance of classical planning as a basic model for multi-agent planning (Brafman and Domshlak 2008), we foresee the importance of truly concurrent temporal planning for enabling interesting forms of multi-agent temporal planning. Furthermore, Rintanen (2015b) showed that PDDL 2.1 induces temporal gaps between consecutive independent actions; thus, no current approach taking PDDL problems as input are capable of producing plans with simultaneous events.

Our approach builds on previous work by Jiménez, Jonsson, and Palacios (2015), but we stress the differences with respect to the initial ideas of Rintanen (2007), such that our translation is not necessarily affected by the time scale or the durations of the actions.

The rest of the paper is organized as follows. First we introduce classical and temporal planning models. Next we present STP, motivating the compilation and proving its soundness. Then we present experimental results, showing that STP is particularly strong in the case of simultaneous events. Finally, we comment on related work and conclude.

Background

In this section we introduce the formalisms of classical planning and temporal planning. Since we are interested in temporal planning with simultaneous events, we focus specifically on the semantics of concurrent action execution. Furthermore, we introduce the mechanism for preserving the temporal constraints between actions.

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 is a set of actions, $I \subseteq F$ is an initial state, and $G \subseteq F$ is 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 $s \ltimes a = (s \setminus \mathsf{del}(a)) \cup \mathsf{add}(a)$.

A *plan* for P is an action sequence $\pi = \langle a_1, \ldots, a_n \rangle$ that induces a state sequence $\langle s_0, s_1, \ldots, s_n \rangle$ such that $s_0 = I$ and, for each i such that $1 \leq i \leq n$, a_i is applicable in s_{i-1} and results in the next state $s_i = s_{i-1} \ltimes a_i$. The plan π *solves* P if and only if G holds in the last state, i.e. if $G \subseteq s_n$.

Actions may have *conditional effects*, a common extension to classical actions. Each conditional effect has a condition and effects $\langle \mathsf{cond}_i(a), \mathsf{cadd}_i(a), \mathsf{cdel}_i(a) \rangle$. When an action a with conditional effects is applicable in an state s, the effects of a include effects whose conditions hold in s, assuming the usual consistency requirements. Let's say

$$\mathsf{tadd} = \mathsf{add}(a) \cup \textstyle\bigcup_{s \models \mathsf{cond}_i(a)} \mathsf{cadd}_i(a),$$
$$\mathsf{tdel} = \mathsf{del}(a) \cup \textstyle\bigcup_{s \models \mathsf{cond}_i(a)} \mathsf{cdel}_i(a).$$

Then, $s \ltimes a = (s \setminus \mathsf{tdel}(a)) \cup \mathsf{tadd}(a)$.

A *concurrent action* $\mathcal{A} = \{a^1, \ldots, a^k\}$ is a set of multiple actions from A. We adopt the definition of valid concurrent actions from PDDL 2.1 (Fox and Long 2003):

Definition 1. *A concurrent action* $\mathcal{A} = \{a^1, \ldots, a^k\}$ *is* valid *if an only if does not exist a fluent* $f \in F$ *and an action pair* $(a^i, a^j) \subseteq \mathcal{A}$ *such that* $f \in \mathsf{add}(a^i) \cup \mathsf{del}(a^i)$ *and* $f \in \mathsf{pre}(a^j) \cup \mathsf{add}(a^j) \cup \mathsf{del}(a^j)$.

Intuitively, if f is an effect of an action $a^i \in \mathcal{A}$, f cannot appear as a precondition or effect of another action $a^j \in \mathcal{A}$. Though this definition imposes a strong restriction on concurrent actions, it is commonly used in temporal planning, and implemented as part of VAL (Howey, Long, and Fox 2004), a tool used to validate temporal plans.

We can view a valid concurrent action $\mathcal{A} = \{a^1, \ldots, a^k\}$ as a classical action by defining its precondition and effects as the union of the individual preconditions and effects:

$$\mathsf{pre}(\mathcal{A}) = \bigcup_{i=1}^{k} \mathsf{pre}(a^i), \quad \mathsf{add}(\mathcal{A}) = \bigcup_{i=1}^{k} \mathsf{add}(a^i),$$

with $\mathsf{del}(\mathcal{A})$ defined analogously. Due to Definition 1, $\mathcal{A}$ is a well-defined classical action without conflicting effects.

A *concurrent plan* for P is a sequence of concurrent actions $\pi = \langle \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$. The concurrent plan π solves P if and only if each concurrent action $\mathcal{A}_i$, $1 \leq i \leq n$, is valid and the resulting sequence of equivalent classical actions solves P according to the semantics of classical plans.

Temporal Planning

A *temporal planning problem*[1] is a tuple $P = \langle F, A, I, G \rangle$, where the fluent set F, initial state I and goal condition G are defined as for classical planning. The action set A consists of *temporal* or *durative* actions $a \in A$ 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.

Although a has a duration, its effects apply instantaneously at the start and end of a, respectively. The preconditions $\mathsf{pre}_s(a)$ and $\mathsf{pre}_e(a)$ are also checked instantaneously, but $\mathsf{pre}_o(a)$ has to hold for the entire duration of a.

The semantics of temporal actions can be defined in terms of two discrete *events* start_a and end_a, each of which is naturally expressed as a classical action (Fox and Long 2003):

$$\begin{aligned} \mathsf{pre}(\mathsf{start}_a) &= \mathsf{pre}_s(a), & \mathsf{pre}(\mathsf{end}_a) &= \mathsf{pre}_e(a), \\ \mathsf{add}(\mathsf{start}_a) &= \mathsf{add}_s(a), & \mathsf{add}(\mathsf{end}_a) &= \mathsf{add}_e(a), \\ \mathsf{del}(\mathsf{start}_a) &= \mathsf{del}_s(a), & \mathsf{del}(\mathsf{end}_a) &= \mathsf{del}_e(a). \end{aligned}$$

Starting temporal action a in state s is equivalent to applying the classical action start_a in s, first verifying that $\mathsf{pre}(\mathsf{start}_a)$ holds in s. Ending a in state s' is equivalent to applying end_a in s', first verifying that $\mathsf{pre}(\mathsf{end}_a)$ holds in s'. The duration $d(a)$ and precondition over all $\mathsf{pre}_o(a)$ impose restrictions on this process: end_a has to occur exactly $d(a)$ time units after start_a and $\mathsf{pre}_o(a)$ has to hold in all states between start_a and end_a. For brevity, we use the term *context* to refer to a precondition over all $\mathsf{pre}_o(a)$, and we use $F_o \subseteq F$ to denote the set of fluents that appear in contexts.

A *temporal plan* for P is a set of action-time pairs $\pi = \{(a_1, t_1), \ldots, (a_k, t_k)\}$. Each action-time pair $(a, t) \in \pi$ is composed of a temporal action $a \in A$ and a scheduled start time t of a, and induces two events start_a and end_a with associated timestamps t and $t + d(a)$, respectively. If we order events by their timestamp and merge events with the same timestamp, the result is a concurrent plan $\pi' = \langle \mathcal{A}_1, \ldots, \mathcal{A}_m \rangle$ for the associated classical planning problem $P' = \langle F, A', I, G \rangle$, where $A' = \{\mathsf{start}_a, \mathsf{end}_a : a \in A\}$.

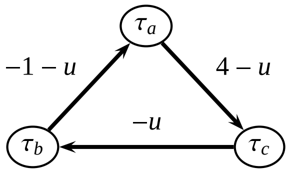
A temporal plan $\pi = \{(a_1, t_1), \ldots, (a_k, t_k)\}$ solves P if and only if the induced concurrent plan $\pi' = \langle \mathcal{A}_1, \ldots, \mathcal{A}_m \rangle$ solves the associated classical planning problem P' and, for each $(a, t) \in \pi$ with $\mathsf{start}_a \in \mathcal{A}_i$ and $\mathsf{end}_a \in \mathcal{A}_j$, the context $\mathsf{pre}_o(a)$ holds in the states $s_i, \ldots, s_{j-1}$ of the state sequence induced by π', i.e. in all states between actions $\mathcal{A}_i$ and $\mathcal{A}_j$.

For π to solve P, the concurrent actions of the induced concurrent plan $\pi' = \langle \mathcal{A}_1, \ldots, \mathcal{A}_m \rangle$ have to be valid according to Definition 1. The plan π contains *simultaneous events* if and only if $m < 2k$, i.e. if at least two induced events share time stamps. The context $\mathsf{pre}_o(a)$ of a temporal action a is not affected by simultaneous events: if start_a is part of a concurrent action $\mathcal{A}$, it is safe for another event in $\mathcal{A}$ to add a fluent $f \in \mathsf{pre}_o(a)$, and if end_a is part of a concurrent action $\mathcal{A}'$, it is safe for an event in $\mathcal{A}'$ to delete f.

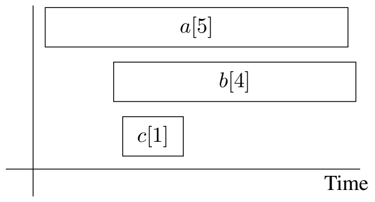
[1]We use the definition of PDDL 2.1 (Fox and Long 2003).

1) $\tau_a - \tau_b \le -u$
2) $\tau_b - \tau_c \le -u$
3) $\tau_c - \tau_c \le 1 - u$
4) $\tau_c - \tau_a \le 4 - u$
5) $\tau_a - \tau_b \le -1 - u$

(a) Temporal constraints.



(b) STN.



(c) Resulting temporal plan.

Figure 1: Example temporal constraints, associated STN and resulting temporal plan.

The quality of a temporal plan is given by its *makespan*, i.e. the temporal duration from the the start of the first temporal action to the end of the last temporal action. Without loss of generality, we assume that the first temporal action is scheduled to start at time 0, i.e. $\min_{(a,t)\in\pi} t = 0$. In this case, the makespan of a temporal plan π is formally defined as $\max_{(a,t)\in\pi}(t + d(a))$.

Simple Temporal Networks (STNs)

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 (1991) showed that scheduling fails if and only if an STN contains negative cycles. Else, the range of feasible assignments to a time variable τ_i is given by $[-d_{i0}, d_{0i}]$, where d_{ij} is the shortest distance in the graph from τ_i to τ_j and τ_0 is a reference time variable whose value is 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).

We illustrate the application of STNs to temporal planning using an example from (Cushing et al. 2007). Let a, b and c be temporal actions with durations $d(a) = 5$, $d(b) = 4$ and $d(c) = 1$, respectively. Assume that we are given an event sequence $\langle \mathsf{start}_a, \mathsf{start}_b, \mathsf{start}_c, \mathsf{end}_c, \mathsf{end}_a, \mathsf{end}_b \rangle$. Then there are three associated time variables τ_a, τ_b and τ_c; we designate τ_a as the reference time variable whose value is 0 since a is the temporal action that starts first. Given the above event sequence, the temporal constraints induced by consecutive events in the sequence are:

1. $\tau_a < \tau_b$,
2. $\tau_b < \tau_c$,
3. $\tau_c < \tau_c + d(c)$,
4. $\tau_c + d(c) < \tau_a + d(a)$,
5. $\tau_a + d(a) < \tau_b + d(b)$.

Since the temporal constraints of TEMPO are strict, we introduce a slack unit of time u and rewrite each constraint $\tau_j + x < \tau_i + y$ on the form $\tau_j - \tau_i \le y - x - u$ (this is possible since τ_i and x are non-negative). Figure 1a shows the temporal constraints rewritten this way. Note that constraint 5) subsumes constraint 1), and that constraint 3) is trivially satisfied whenever $u < 1$.

Figure 1b shows the associated STN after removing constraints 1) and 3). This STN has no negative cycles, and the range of feasible assignments to τ_b is given by $[-d_{ba}, d_{ab}] = [1 + u, 4 - 2u]$. Likewise, the range of feasible assignments to τ_c is given by $[-d_{ca}, d_{ac}] = [1 + 2u, 4 - u]$. This makes sense: a starts at time 0, so for b to end after a ends, b has to start after time 1. For c to end before a ends, c has to start before time 4. The remaining bounds are implied by the fact that c starts after b starts. Since one of the goals of temporal planning is to minimize makespan, i.e. the time until the last action of the temporal plan ends, we always select the smallest possible assignment to each time variable τ_i, i.e. $-d_{i0}$. In the example, this results in a temporal plan $\{(a, 0), (b, 1 + u), (c, 1 + 2u)\}$, illustrated in Figure 1c.

Theoretically, STNs can be modeled in PDDL, using fluents to represent the entries of a matrix and actions to simulate updates of Floyd-Warshall. However, each entry of the matrix can take on a range of values, and the size of the matrix is *not* bounded by the number of active actions, but rather the total number of temporal actions in the plan. In practice, the enormous number of necessary fluents and actions makes this approach unfeasible.

Jiménez, Jonsson, and Palacios (2015) proposed an alternative approach for incorporating STNs into temporal planning. They represent lifted temporal states as part of the search nodes of the Fast Downward planning system (Helmert 2006), which is where auxiliary information about a state is stored (e.g. the predecessor state). Specifically, to each search node they add an STN, a list of active actions and the latest event.

Each time a compiled action is applied (either start_a or end_a for some a), Fast Downward generates a successor state. To the search node associated with this state they add an STN which is a copy of the STN of its predecessor, but with a single new edge corresponding to the temporal constraint generated by the successor rule of TEMPO. They then recompute the shortest paths of the STN.

Given an STN (V, E) with accompanying shortest paths, Cesta and Oddi (1996) described an $O(|V||E|)$ algorithm for adding a single edge to the STN and recomputing the shortest paths. However, Jiménez, Jonsson, and Palacios take a different approach to updating the STN. Instead of explicitly representing the STN, they represent the STN implicitly using the matrix of shortest distances (as in Floyd Warshall's algorithm). When a new edge (τ_i, τ_j) is added to the STN, for each pair of nodes there is a single new candidate shortest path, namely that via (τ_i, τ_j). Since the shortest distances to τ_i and from τ_j are already represented, the update can be performed in time $O(|V|^2)$, which is typically much smaller than $O(|V||E|)$. In the modified version of Fast Downward, they prune a search node whenever the corresponding STN contains negative cycles, i.e. when the temporal actions cannot be scheduled in a way that coincides with the current event sequence. The temporal constraints can thus be viewed as an implicit precondition of actions which is invisible to the planner (e.g. when computing heuristics).

POPF (Coles et al. 2010) and OPTIC (Benton, Coles, and Coles 2012) also use STNs for solving temporal planning problems. POPF encodes the STN using linear programming, which allows it to compute plans with actions that cause continuous linear numeric changes. On the other hand, OPTIC encodes the STN as a mixed integer problem which additionally allows handling temporally dependent costs.

The STP Planner

In this section we describe STP (Simultaneous Temporal Planner). STP is built on top of the same framework as TP (Jiménez, Jonsson, and Palacios 2015), but incorporates additional machinery in order to handle simultaneous events. STP shares several characteristics with TP:

1. Both TP and STP apply a modified version of the Fast Downward (FD) planning system to generate temporal plans. The modified version of FD incorporates simple temporal networks or STNs to represent temporal constraints.

   There is a time variable τ_i for each temporal action a_i of a temporal plan, and a feasible assignment to τ_i corresponds to the time t_i when a_i should be scheduled to form an action-time pair (a_i, t_i). During the search process, a branch is pruned if the temporal constraints are violated. At the end of the section we describe the temporal constraints imposed by STP.

2. Both TP and STP impose a bound K on the number of *active* temporal actions, that started but did not end yet. Hence no more than K temporal actions can execute concurrently.

3. Both compilations are described for problems with fixed durations and no duration dependent effects.

Unlike TP, STP also defines a constant C that represents the maximum value of a cyclic counter that starts at 0, and resets to 0 after reaching C (more details later).

STP works by protecting the contexts of temporal actions in case a naïve execution of events using classical planning would produce inconsistent results. Our compilation divides each concurrent event into three phases:

1. End phase (immediately before the event). This is where active actions are scheduled to end, and in doing so, the corresponding counters of fluents in context are decremented.

2. Event phase (concurrent event itself). This is where simultaneous events take place, both ending and starting actions. Here we check preconditions and apply effects, and verify that the concurrent event is valid.

3. Start phase (immediately after the event). Here we check that the contexts of active actions that just started are satisfied (possibly as a result of being added during the concurrent event itself), and increment the corresponding counters of fluents in context.

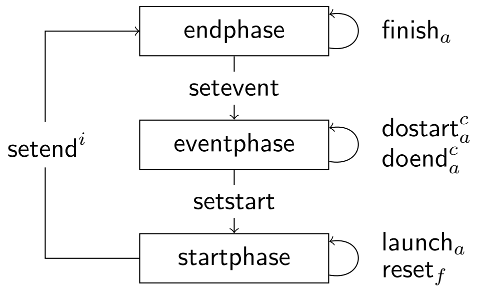


Figure 2: Interaction between the different actions introduced by the STP planner in the different phases.

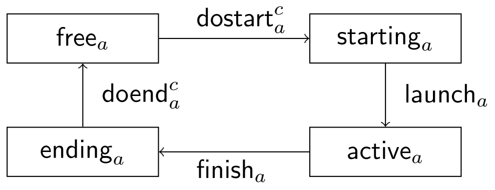


Figure 3: Fluents that are enabled each time an action in the compilation is executed.

Figure 2 shows the interconnection between the phases and actions introduced in the compilation. Actions setevent, setstart and setend^i change the current phase, whereas the other actions can only be applied (if their preconditions hold) in the corresponding phase. Actions dostart and doend correspond to the semantic events. Execution begins in the endphase, and ends in the startphase.

Figure 3 shows the cycle each action $a \in A$ passes through in the compilation. Between dostart$_a$ and doend$_a$, actions launch$_a$ and finish$_a$ execute the start phase and end phase, respectively. Each time we transition from one state to another, we delete the auxiliary fluent of the state, and add the next, thus obtaining a mutex invariant. We also use additional fluents nstarting$_a$ and nending$_a$, that have the opposite values of starting$_a$ and ending$_a$, respectively.

Now, we are ready to present the compilation itself. Let $P = \langle F, A, I, G\rangle$ be a temporal planning problem. Given constants K and C, STP compiles P into a classical planning problem $\Pi_{K,C} = \langle F_{K,C}, A_{K,C}, I_{K,C}, G_{K,C}\rangle$.

Fluents

To ensure that the contexts of temporal actions are not violated, STP follows the same scheme as TP: for each fluent $f \in F_o$ that appears in contexts, we introduce fluents count^c_f that model the number c of active temporal actions that have

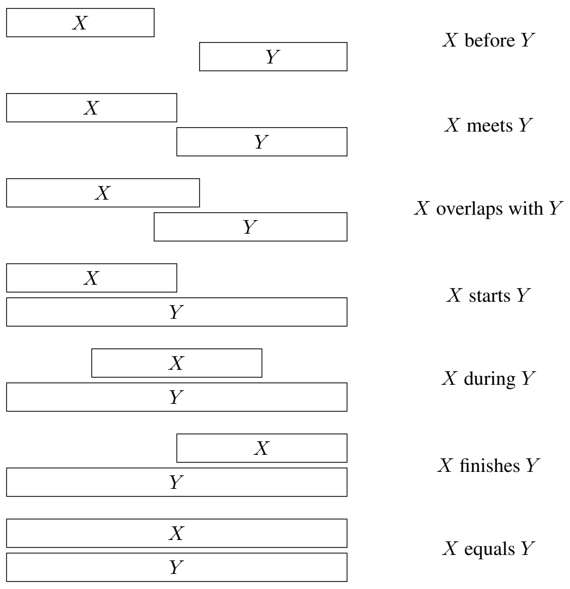


Figure 4: The seven relations on interval pairs (X, Y) in Allen's interval algebra.

f as context. For an event to delete a fluent $f \in F_o$, its count has to equal 0 (no active actions can have f as context).

STP also has to ensure that joint events are valid, and ensure that contexts are properly handled at the start and end of a temporal action. Recall that a fluent $f \in \mathsf{pre}_o(a)$ in the context of a temporal action a may be added by an event that is simultaneous with the start of a, and that f may be deleted by an event that is simultaneous with the end of a.

The set $F_{K,C} \supseteq F$ contains the following new fluents:

- For each $a \in A$, fluents free_a and active_a indicating that a is free (i.e. did not start) or active.
- For each $f \in F_o$ and each $c, 0 \le c \le K$, a fluent count_f^c indicating that c active actions have f as context.
- For each $c, 0 \le c \le K$, a fluent concur^c indicating that there are c concurrent active actions.
- Fluents $\mathsf{endphase}$, $\mathsf{eventphase}$ and $\mathsf{startphase}$ corresponding to the three phases described above.
- For each $a \in A$, fluents $\mathsf{starting}_a$, ending_a, $\mathsf{nstarting}_a$ and $\mathsf{nending}_a$ indicating that a is starting, ending, not starting and not ending, respectively.
- For each $f \in F$, fluents canpre_f and caneff_f indicating that we can use f as a precondition or effect.
- Fluents $\mathsf{endcount}^i, 0 \le i < C$, that model the number of times the end phase has occurred. This counter is cyclic, i.e. if $i = C - 1$, then $i + 1 = 0$.

To motivate the role of the end count, we consider an example instance of the Allen's Interval Algebra (AIA) domain (Jiménez, Jonsson, and Palacios 2015), where a set of time intervals must be scheduled such that they comply with a set of relations between them (see Figure 4). The example is the following: the start of i_1 and i_2 should be simultaneous, as should the end of i_2 and i_3. The durations of the intervals are 5 for i_1 and i_3, and 11 for i_2. Figure 5a shows the only possible solution for this problem, whereas Figure 5b shows two different intermediate solutions A and B that the planner might explore. The black dotted lines indicate end phases. In solution A, the start of i_3 is concurrent with the end of i_1, which is not the case in solution B. The solid red line indicates the time at which solutions A and B assign the same values to the fluents in $F_{K,C}$.

If no end counts are maintained and a planner first explores A, it will not find a solution since ending i_2 and i_3 simultaneously violates the temporal constraints. When the planner later explores B, it will find the state on fluents identical to A and prune this branch of search. As a result, the planner will report that no solution exists, even though the instance does have a valid solution, namely B. The end count allows the planner to distinguish between A and B.

Note that the end count is not infallible: since it is cyclic, state repetitions can still happen. The higher the end count is, the less likely it is that states are repeated. However, increasing the end count increases the complexity of the problem (higher number of fluents and actions), so it can be more difficult to obtain a solution.

Lemma 1. *The number of fluents of* $\Pi_{K,C}$ *is given by* $|F_{K,C}| = 3\,|F| + 6\,|A| + (K+1)\,(|F_o|+1) + C + 3$.

Proof. By inspection of the fluents in $F_{K,C}$. For each $f \in F$, $F_{K,C}$ contains three fluents f, canpre_f and caneff_f. For each $a \in A$, $F_{K,C}$ contains six fluents free_a, active_a, $\mathsf{starting}_a$, ending_a, $\mathsf{nstarting}_a$ and $\mathsf{nending}_a$. For each $f \in F_o$, $F_{K,C}$ contains $K+1$ fluents of type count_f^c, and there are $K+1$ fluents of type concur^c. Finally, there are C fluents of type $\mathsf{endcount}^i$, and three fluents $\mathsf{endphase}$, $\mathsf{eventphase}$ and $\mathsf{startphase}$. □

The initial state $I_{K,C}$ is defined as

$$\begin{aligned} I_{K,C} = I &\cup \{\mathsf{free}_a, \mathsf{nstarting}_a, \mathsf{nending}_a : a \in A\} \\ &\cup \{\mathsf{concur}^0, \mathsf{endphase}, \mathsf{endcount}^0\} \cup \{\mathsf{count}_f^0 : f \in F_o\} \\ &\cup \{\mathsf{canpre}_f, \mathsf{caneff}_f : f \in F\}, \end{aligned}$$

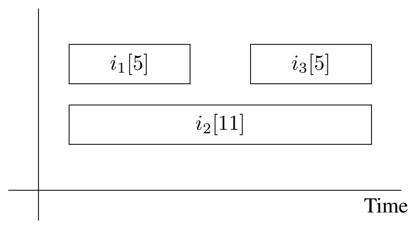
and the goal is $G_{K,C} = G \cup \{\mathsf{concur}^0\} \cup \{\mathsf{startphase}\}$.

Actions

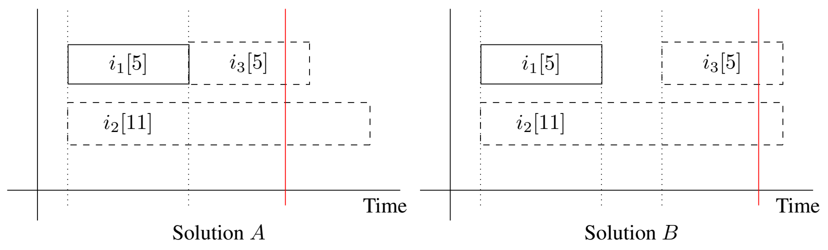
The action set $A_{K,C}$ contains several actions corresponding to each temporal action $a \in A$: $\mathsf{dostart}_a^c$ and launch_a for starting a, and doend_a^c and finish_a for ending a. For each c, $0 \le c < K$, action $\mathsf{dostart}_a^c$ is defined as

$$\begin{aligned} \mathsf{pre} = {} & \mathsf{pre}_s(a) \cup \{\mathsf{eventphase}, \mathsf{concur}^c, \mathsf{free}_a\} \\ & \cup \{\mathsf{count}_f^0 : f \in F_o \cap \mathsf{del}_s(a)\} \cup \{\mathsf{canpre}_f : f \in \mathsf{pre}_s(a)\} \\ & \cup \{\mathsf{caneff}_f : f \in \mathsf{add}_s(a) \cup \mathsf{del}_s(a)\}, \\ \mathsf{add} = {} & \mathsf{add}_s(a) \cup \{\mathsf{concur}^{c+1}, \mathsf{starting}_a\}, \\ \mathsf{del} = {} & \mathsf{del}_s(a) \cup \{\mathsf{concur}^c, \mathsf{free}_a, \mathsf{nstarting}_a\} \\ & \cup \{\mathsf{caneff}_f : f \in \mathsf{pre}_s(a)\} \\ & \cup \{\mathsf{canpre}_f, \mathsf{caneff}_f : f \in \mathsf{add}_s(a) \cup \mathsf{del}_s(a)\}. \end{aligned}$$

For a given $c < K$, we can only start a in the event phase if there are c active actions and a is free. All contexts deleted



(a) The only possible scheduling.



(b) Two possible intermediate solutions. With no end counter, B is not explored (see text).

Figure 5: Example instance of the AIA domain: i_1 and i_2 start at the same time, while i_2 and i_3 end at the same time.

at start of a need a count of 0, and all preconditions and effects at start of a have to be available. Starting a adds fluent $\mathsf{starting}_a$, deletes free_a and $\mathsf{nstarting}_a$, and increments the number of active actions. Moreover, deleting fluents of type canpre_f and caneff_f prevents invalid joint events: if f is a precondition at start of a, f can no longer be used as an effect in this event phase, and if f is an effect at start of a, f can no longer be used as a precondition or effect.

For each $c, 0 \leq c < K$, action doend_a^c is defined as

$$\begin{aligned}
\mathsf{pre} &= \mathsf{pre}_e(a) \cup \{\mathsf{eventphase}, \mathsf{concur}^{c+1}, \mathsf{ending}_a\} \\
&\cup \{\mathsf{count}_f^0 : f \in F_o \cap \mathsf{del}_e(a)\} \cup \{\mathsf{canpre}_f : f \in \mathsf{pre}_e(a)\} \\
&\cup \{\mathsf{caneff}_f : f \in \mathsf{add}_e(a) \cup \mathsf{del}_e(a)\}, \\
\mathsf{add} &= \mathsf{add}_e(a) \cup \{\mathsf{concur}^c, \mathsf{free}_a, \mathsf{nending}_a\}, \\
\mathsf{del} &= \mathsf{del}_e(a) \cup \{\mathsf{concur}^{c+1}, \mathsf{ending}_a\} \\
&\cup \{\mathsf{caneff}_f : f \in \mathsf{pre}_e(a)\} \\
&\cup \{\mathsf{canpre}_f, \mathsf{caneff}_f : f \in \mathsf{add}_e(a) \cup \mathsf{del}_e(a)\}\}.
\end{aligned}$$

For a given value of c, we can only end a in the event phase if there are $c+1$ active actions and a is already scheduled to end, represented by fluent ending_a. Ending a adds fluents free_a and $\mathsf{nending}_a$ and decrements the number of active actions. The remaining action definition is analogous to $\mathsf{dostart}_a^c$ and controls the validity of the joint event.

Action launch_a is responsible for completing the start of a during the start phase:

$$\begin{aligned}
\mathsf{pre} &= \mathsf{pre}_o(a) \cup \{\mathsf{startphase}, \mathsf{starting}_a\}, \\
\mathsf{add} &= \{\mathsf{active}_a, \mathsf{nstarting}_a\}, \\
\mathsf{del} &= \{\mathsf{starting}_a\}.
\end{aligned}$$

This is where we check that the contexts of a hold, and due to the precondition $\mathsf{starting}_a$ we can only launch a if a was started during the event phase. The result is adding active_a and $\mathsf{nstarting}_a$ and deleting $\mathsf{starting}_a$. In addition to the described effects, the action launch_a includes conditional effects $\langle\{\mathsf{count}_f^l\}, \{\mathsf{count}_f^{l+1}\}, \{\mathsf{count}_f^l\}\rangle$, $f \in \mathsf{pre}_o(a)$ and $0 \leq l < K$, incrementing the count of each $f \in \mathsf{pre}_o(a)$.

Finally, action finish_a is needed to schedule a for ending:

$$\begin{aligned}
\mathsf{pre} &= \{\mathsf{endphase}, \mathsf{active}_a\}, \\
\mathsf{add} &= \{\mathsf{ending}_a\}, \\
\mathsf{del} &= \{\mathsf{active}_a, \mathsf{nending}_a\}.
\end{aligned}$$

The result is adding ending_a and deleting active_a and $\mathsf{nending}_a$. In addition, finish_a includes conditional effects for decrementing the context counts of fluents in $\mathsf{pre}_o(a)$.

The action set $A_{K,C}$ also needs actions $\mathsf{setevent}$, $\mathsf{setstart}$ and setend^i, $0 \leq i < C$, whose purpose is to switch between phases. Action $\mathsf{setevent}$ is defined as

$$\begin{aligned}
\mathsf{pre} &= \{\mathsf{endphase}\}, \\
\mathsf{add} &= \{\mathsf{eventphase}\}, \\
\mathsf{del} &= \{\mathsf{endphase}\}.
\end{aligned}$$

Action $\mathsf{setstart}$ is defined as

$$\begin{aligned}
\mathsf{pre} &= \{\mathsf{eventphase}\} \cup \{\mathsf{nending}_a : a \in A\}, \\
\mathsf{add} &= \{\mathsf{startphase}\}, \\
\mathsf{del} &= \{\mathsf{eventphase}\}.
\end{aligned}$$

Note that we cannot leave the event phase unless all actions in the joint event have ended.

Action setend^i is defined as

$$\begin{aligned}
\mathsf{pre} &= \{\mathsf{startphase}, \mathsf{endcount}^i\} \cup \{\mathsf{nstarting}_a : a \in A\} \\
&\cup \{\mathsf{canpre}_f, \mathsf{caneff}_f : f \in F\}, \\
\mathsf{add} &= \{\mathsf{endphase}\} \cup \{\mathsf{endcount}^j : j = (i+1) \bmod C\}, \\
\mathsf{del} &= \{\mathsf{startphase}, \mathsf{endcount}^i\}.
\end{aligned}$$

Note that we cannot leave the start phase unless all actions in the joint event have started and all fluents are available as preconditions or effects. In addition, setend^i increments the end count.

The resulting action set, $A_{K,C}$, also contains a reset action reset_f for each $f \in F$:

$$\begin{aligned}
\mathsf{pre} &= \{\mathsf{startphase}\}, \\
\mathsf{add} &= \{\mathsf{canpre}_f, \mathsf{caneff}_f\}, \\
\mathsf{del} &= \emptyset.
\end{aligned}$$

These actions can only be applied in the start phase.

Lemma 2. *The number of actions of the classical planning problem $\Pi_{K,C}$ is $|A_{K,C}| = (2K+2)\,|A| + C + |F| + 2$.*

Proof. For each $a \in A$, $A_{K,C}$ contains $2K+2$ actions $\mathsf{dostart}_a^c$, doend_a^c, launch_a and finish_a, $0 \leq c < K$. $A_{K,C}$ also contains $C+2$ actions $\mathsf{setevent}$, $\mathsf{setstart}$ and setend^i, $0 \leq i < C$, and $|F|$ actions of type reset_f. □

In the modification of FD, we introduce temporal constraints every time we generate events. For a given event e, let $\tau_e = \tau_a$ if $e = \mathsf{start}_a$ for some temporal action a, and let $\tau_e = \tau_a + d(a)$ if $e = \mathsf{end}_a$. Let $\{e_1, \ldots, e_k\}$ be a concurrent event generated during the event phase of our compilation. To ensure that the events are scheduled at the same time, we introduce the temporal constraint $\tau_{e_j} \leq \tau_{e_{j+1}}$ for each j, $1 \leq j < k$, as well as the temporal constraint $\tau_{e_k} \leq \tau_{e_1}$. In addition, for each active action a' that started before the concurrent event, we introduce the temporal constraint $\tau_{e_j} + u \leq \tau_{a'} + d(a')$ for each j, $1 \leq j < k$, where u is a slack unit of time which ensures that the end of a' takes place strictly after e_j. This last constraint is redundant since the end of a' will eventually be scheduled after the given concurrent event, but in practice it helps ensure that unsound temporal plans are pruned as soon as possible.

For two consecutive concurrent events $\{e_1, \ldots, e_k\}$ and $\{e'_1, \ldots, e'_m\}$, we introduce the constraint $\tau_{e_k} + u \leq \tau_{e'_1}$. Our modification of FD maintains an STN that is updated each time a new temporal constraint is added, and prunes a search node as soon as the temporal constraints are impossible to satisfy, i.e. when the STN contains negative cycles.

Theorem 3 (Soundness)**.** *Let π' be a plan that solves the classical planning instance $\Pi_{K,C}$ by our modified version of FD. Given π', we can always construct a temporal plan π that solves the temporal planning instance Π.*

Proof. The system can only be in one phase at a time, and we can only cycle through phases in the order $\mathsf{endphase} \rightarrow \mathsf{eventphase} \rightarrow \mathsf{startphase} \rightarrow \mathsf{endphase}$ using actions $\mathsf{setevent}$, $\mathsf{setstart}$ and setend^i. The system is initially in the end phase, and the goal state requires us to be in the start phase with no actions active (due to goal condition concur^0).

A temporal action a can start in the event phase and launch in the start phase. Specifically, the fluent $\mathsf{nstarting}_a$ is deleted by $\mathsf{dostart}_a^c$ and added by launch_a. After starting a in the event phase, we cannot end a until another subsequent event phase since the precondition ending_a of action doend_a^c is only added by finish_a, which is only applicable in the end phase. Together with the fact that no action is active in the goal, starting a implies that we have to fully cycle through all the phases at least one more time. In turn, this requires us to apply action setend^i. Due to the precondition $\mathsf{nstarting}_a$ of setend^i, we cannot start a in the event phase without launching a in the very next start phase.

Likewise, a temporal action a can finish (i.e. be scheduled for ending) in the end phase, and end in the event phase. Specifically, the fluent $\mathsf{nending}_a$ is deleted by finish_a and added by doend_a^c. After ending a in the event phase, we have to apply action $\mathsf{setstart}$ at least one more time since the goal state requires us to be in the start phase. Due to the precondition $\mathsf{nending}_a$ of $\mathsf{setstart}$, we cannot finish a in the end phase without ending a in the very next event phase.

For each $f \in F$, fluents canpre_f and caneff_f are true each time we apply action $\mathsf{setevent}$, i.e. when entering the event phase. These fluents are true in the initial state, i.e. in the end phase, and are only deleted by actions of type $\mathsf{dostart}_a^c$ and doend_a^c, which are only applicable in the event phase. The precondition $\{\mathsf{canpre}_f, \mathsf{caneff}_f\}$ of action setend^i requires us to reset f in the start phase using action reset_f, and there are no actions that delete these fluents in the end phase.

A solution plan π' for $\Pi_{K,C}$ thus has the following form:

$$\langle \underline{\mathsf{setevent}}, \mathsf{dostart}_a, \underline{\mathsf{setstart}}, \mathsf{launch}_a, \mathsf{reset}_f, \underline{\mathsf{setend}}, \ldots,$$
$$\ldots, \underline{\mathsf{setend}}, \mathsf{finish}_a, \underline{\mathsf{setevent}}, \mathsf{doend}_a, \underline{\mathsf{setstart}} \rangle$$

For clarity, actions that alter the phases are underlined. We may, of course, start and launch multiple actions at once, as well as finish and end multiple actions at once. We may also start and end actions during the same event phase.

We show that each joint event induced by π' is valid. Each time a fluent f appears as an effect of an event, deleting fluents canpre_f and caneff_f prohibits f from appearing as a precondition or effect of another event in the same event phase (reset_f is not applicable until the following start phase). Likewise, each time f appears as a precondition, deleting caneff_f prohibits f from appearing as an effect of another event. Because of the mechanism for finishing and launching temporal actions, the context of a temporal action a may be added by an event simultaneous with starting a and deleted by an event simultaneous with ending a.

Since π' is reported as a solution plan for $\Pi_{K,C}$ by our modified version of FD, the resulting STN does not contain negative cycles, making it possible to satisfy all temporal constraints. Since the temporal constraints require all concurrent events to take place simultaneously and all subsequent events to take place after a given concurrent event, we can convert π' into a temporal plan π by assigning a starting time $t = -d_{i0}$ to each action a_i of π', where d_{i0} is the shortest distance in the graph from τ_i to τ_0, the temporal action that starts at time 0. This ensures that event sequence induced by π is identical to π'. Since π' solves $\Pi_{K,C}$ and ensures that no contexts of temporal actions are violated, this ensures that the goal condition G is satisfied after the execution of the temporal plan π, implying that π solves P. $\square$

Although STP can deal with many kinds of temporal problems (sequential, with single hard envelopes, simultaneous events, ...), it is not complete. The reason precisely depends on the parameters K and C. First, there can be problems for which a given K is not enough to solve the problem; for instance, STP will not solve a problem requiring 5 concurrent actions if $K < 5$. Second, the end count C is cyclic: it reduces the risk of ignoring propositionally equal but temporally different states, but it does not remove it.

Given the dependency on K and C, an appropriate strategy for trying to solve a temporal problem using STP could consist on starting from low values of K and C and increasing them while the solution is not found.

Results

We performed an evaluation in all 10 domains of the temporal track of IPC-2014. Moreover, we added the DRIVERLOGSHIFT (DLS) domain (Coles et al. 2009), the AIA domain (Jiménez, Jonsson, and Palacios 2015), and a domain based on an STN example by Cushing et al. (2007) (from now on, we refer to this domain as CUSHING)[2].

STP was executed for values of K in the range $1, \ldots, 4$ and with a fixed end count $C = 10$, which proved to work fine in AIA instances. We compared STP to several other planners that compile problems into classical planning: the TP planner using the same values of K, and the TPSHE planner using the LAMA-2011 setting of FD to solve the compiled instance. We also ran experiments for POPF2 (Coles et al. 2010) (the runner-up at IPC-2011), YAHSP3-MT (Vidal 2014) (the winner at IPC-2014), and ITSAT (Rankooh and Ghassem-Sani 2015).

Table 1 shows, for each planner, the IPC quality score and the coverage, i.e. the number of instances solved per domain. Experiments were executed on a Linux computer with Intel Core 2 Duo 2.66GHz processors. Each experiment had a cutoff of 10 minutes or 4GB of RAM memory.

The benchmark domains can be classified into four categories. Seven domains (DRIVERLOG, FLOORTILE, MAPANALYSER, PARKING, RTAM, SATELLITE and STORAGE) can be solved using sequences of temporal actions, i.e. there is no need for actions to execute concurrently. In these domains, TPSHE comes out on top, solving 98 instances with an IPC score of 76.44, followed by TP(1) and YAHSP3-MT. The latter can in fact *only* solve domains of this type. We remark that at IPC-2014, YAHSP3-MT solved 103 instances; one reason for this discrepancy is that we used a shorter cutoff, and YAHSP3-MT is also sensitive to input parameters.

All variants of STP performed particularly poorly in these seven domains, with the top performer being STP(1) with 34 instances solved. Compared to TP(1), which solved more than twice that number, STP(1) incorporates additional fluents and actions associated with the three phases used to simulate events. Since events are not concurrent, these additional fluents and action only hurt performance, resulting in a larger state space and an increased branching factor.

A further four domains (DLS, MATCHCELLAR, TMS and TURN&OPEN) can be solved using temporal plans that only incorporate concurrency in the form of *single hard envelopes* (Coles et al. 2009). ITSAT is the top performer in these domains, solving 60 instances with an IPC score of 56.97, followed by TPSHE with 59 instances solved. TP(1) and STP(1) cannot solve any instance since they do not allow for concurrency. TP(2) solves 40 instances, compared to 24 instances of STP(2), STP(3) and STP(4). Again, since simultaneous events are not needed, the increased size of the compilation in STP makes instances harder to solve.

The third category is represented by the CUSHING domain, which requires concurrency *not* in the form of single hard envelopes, but does not require events to take place simultaneously. In this domain, ITSAT and TPSHE cannot solve any instances, and the top performer is instead POPF2, which solves all instances. TP(3) and TP(4) also solve all instances but produce plans with longer makespan, and STP(3) and STP(4) solve 14 and 5 instances, respectively. Yet again the additional machinery of STP does not pay off. The fact that STP(4) performs much worse than STP(3) highlights the fact that the increased number of fluents and actions makes instances more difficult to solve.

[2]The code of the compilation and the domains are available at https://github.com/aig-upf/temporal-planning.

Domain	Compression	Domain	Compression
AIA	0.94	MATCHCELLAR	0.83
CUSHING	0.85	PARKING	0.66
DLS	0.66	SATELLITE	0.72
MAPANALYSER	0.72		

Table 2: Average level of compression of plans returned by STP(K) in relation to the best results of the other planners.

The only domain in which STP excels is AIA, in which many instances do require concurrency in the form of simultaneous events, representing the fourth and final category. STP(4) is the only planner that can solve all 25 instances. POPF2 and TP(4) solve all 10 instances that do not require simultaneous events, and all planners solve the 3 instances that can be solved using sequential temporal plans.

Although STP performs poorly in most domains, we still argue that the ability to deal with simultaneous events in a forward-search temporal planner is a contribution to the field of temporal planning, for two reasons. The benchmark domains commonly used in temporal planning present a clear bias towards domains that are challenging from a combinatorial perspective, but the temporal aspect is almost trivial, which is the reason that few temporal planners are able to handle concurrent temporal actions in a robust manner. The reason STP performs poorly is mostly because of the combinatorial aspect, since the ability to deal with simultaneous events comes with a price: an increased number of fluents and actions, resulting in a larger search space.

The second reason that STP may have an impact is if researchers develop a way to analyze temporal planning domains prior to solving them. If the analysis shows that a domain is temporally simple, then there is no need to run STP, since it will always perform worse than several other alternatives. Only when the analysis determines that more intricate forms of concurrency are required , STP will be executed.

Since STP can output plans with simultaneous events, we compare the number of events in its plans with the number of events in other planners. Table 2 shows how much STP(K) planners compress (on average) plans in comparison to the other planners (TPSHE, TP(K), POPF2, YAHSP3-MT and ITSAT). Given a problem, we compare the number of events in plans output by STP(K) against the number of events output by a planner from the other group. The comparison is done just if the problem is solved by both groups.

From the table, we observe that solutions returned by STP always contain less events than the other planners. In domains like DLS and PARKING, the solutions contain around 30% less events compared to the other planners.

Related Work

Several authors have provided theoretical justification for splitting durative actions into classical actions (Cooper, Maris, and Régnier 2013) and proposed a compilation for doing so. An early approach, LPGP (Long and Fox 2003), turned out to be unsound and incomplete since it failed to (1) ensure that temporal actions end before reaching the goal,

	TPSHE	TP(1)	TP(2)	TP(3)	TP(4)	STP(1)	STP(2)	STP(3)	STP(4)	POPF2	YAHSP3-MT	ITSAT
AIA[25]	3/3	3/3	6.5/8	7.5/9	8.5/10	3/3	17.17/22	19.51/24	**23.5/25**	10/10	3/3	3/3
CUSHING[20]	0/0	0/0	0/0	4.07/**20**	4.93/**20**	0/0	0/0	3.31/14	2.28/5	**20/20**	0/0	0/0
DRIVERLOG[20]	**14.78/15**	1.42/5	0.93/3	1.08/4	0.91/3	0/0	0/0	0/0	0/0	0/0	2.31/4	1/1
DLS[20]	9.37/11	0/0	10/10	7.7/9	8.06/9	0/0	3.78/4	3.9/4	3.49/4	7/7	0/0	**16.18/19**
FLOORTILE[20]	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	4.93/5	**19.7/20**
MAPANALYSER[20]	**17.38/20**	10.16/19	13.08/**20**	12.34/**20**	12.02/19	9.18/19	9.81/17	10.09/16	7.69/12	0/0	1/1	0/0
MATCHCELLAR[20]	15.72/**20**	0/0	15.71/**20**	15.71/**20**	15.71/**20**	0/0	15.71/**20**	15.71/**20**	15.71/**20**	**20/20**	0/0	18.91/19
PARKING[20]	6.73/**20**	5.59/19	5.79/17	5.67/17	5.33/16	1.67/6	1.79/6	1.93/6	1.93/6	12/13	**16.84/20**	0.96/6
RTAM[20]	**16/16**	4.91/11	2.45/6	2.73/6	2.79/6	0/0	0/0	0/0	0/0	0/0	0/0	0/0
SATELLITE[20]	**16.63**/18	7.99/19	4.97/13	5.04/13	4.67/12	2.31/6	0/0	0/0	0/0	2.92/3	13.82/**20**	1.68/7
STORAGE[20]	4.92/**9**	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	3.91/**9**	**9/9**
TMS[20]	0.06/9	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	**16/16**
TURN&OPEN[20]	**15.53/19**	0/0	5.05/10	5.03/10	5.19/10	0/0	0/0	0/0	0/0	7.31/8	0/0	5.88/6
Total	**120.12/160**	33.07/76	64.49/107	66.87/128	68.11/125	16.16/34	48.26/69	54.45/84	54.61/72	79.22/81	45.8/62	92.3/106

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

(2) ensure that the contexts of temporal actions are not violated, and (3) ensure that temporal constraints are preserved.

Rintanen (2007) proposed a 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 duration, potentially making the planner incomplete when events have to be scheduled fractions of time units apart and, as far as we know, this compilation has never been implemented as part of an actual planner.

The planners most similar to ours are TPSHE and TP (Jiménez, Jonsson, and Palacios 2015). Both planners are based on compiling temporal planning problems to classical planning problems. TPSHE only handles instances where required concurrency is in the form of single hard envelopes. In contrast, TP partially compiles temporal actions into classical planning and introduces an STN into the Fast Downward classical planner to enforce temporal constraints. POPF (Coles et al. 2010) and OPTIC (Benton, Coles, and Coles 2012) also use STNs. POPF encodes the STN using linear programming which allows it to compute plans with actions that cause continuous linear numeric changes. OPTIC encodes the STN as a mixed integer problem which additionally allows handling temporally dependent costs.

With respect to planners that perform explicit state-space search, an interesting direction is the exploitation of *l*andmarks. This group includes the TEMPLM planner (Marzal, Sebastia, and Onaindia 2014) that discovers classical landmarks from a temporal instance, and builds a landmark graph that expresses the temporal relations between these landmarks. This approach has proven useful to detect unsolvable instances under deadline constraints. However, in the absence of tightly-constrained dead-ends it does not yield significant benefits over classical causal landmarks. Karpas et al. (2015) do not rely on the presence of deadlines to discover landmarks that are not causal landmarks and define notions of temporal fact landmarks, which state that some fact must hold between two given time points, and temporal action landmarks, which state that the start or end of an action must occur at a given time point.

Satisfiability checking is also an important trend in temporal planning. Similarly to the SAT-based approaches for classical planning, temporal planning instances can be encoded as SAT problems. The SAT encoding for temporal planning instances is more elaborated since it involves choosing the start times of actions and verifying the temporal constraints between them. Moreover, PDDL induces temporal gaps between consecutive interdependent actions that effectively doubles the number of joint events required to solve a given temporal planning instance and hence affecting the performance of SAT-based search approaches. The ITSAT planner (Rankooh and Ghassem-Sani 2015) deals with this issue by abstracting out the duration of actions and separating action sequencing from scheduling. ITSAT assumes that actions can have arbitrary duration and encodes the abstract problem into a SAT formula to generate a causally valid plan without checking the existence of a valid schedule. To find a temporally valid plan, ITSAT then tries to schedule the causally valid plan solving the STN defined by the duration of the actions in the plan. If the STN can be solved, ITSAT returns a valid plan, but if not, ITSAT adds the sequence of events that led to the unsolvable STN as new blocking clauses in the SAT encoding. The process is repeated until a valid temporal plan is achieved. A different approach is producing a SAT encoding that integrates action sequencing and scheduling. Recently the modeling language NDL has been proposed as an alternative to PDDL with the aim of producing a SAT Modulo Theories encoding where action sequencing and scheduling are tightly integrated (Rintanen 2015a). Rintanen (2015b) showed that while PDDL forces temporal gaps in action scheduling (which have a performance penalty), NDL avoids such gaps using the notion of resources.

Conclusions

We introduced a compilation from temporal planing with simultaneous events to classical planning, and proved it returns sound plans if used in a forward-search planner that maintains STNs for checking temporal consistency. We showed that our approach performs well in domains requiring simultaneous events, although it is not competitive in domains requiring simpler forms of concurrency. We only use the actual durations of actions in the STN to verify consistency, making our compilation closer to the state-based definition of temporal planning (Rintanen 2007), where durations did not appear. Moreover, we avoid the proposed counters on the remaining duration of actions, instead relying on temporal constraints to enforce soundness.

Solving rich planning problems using classical planning usually involves modeling trajectories that can be translated to rich plans, and additional constraints to enforce the classical planning trajectories correspond to sound rich plans (Baier, Bacchus, and McIlraith 2009; Palacios and Geffner 2009). These mechanisms could be challenging for state-of-the-art planners, developed around handmade benchmarks.

Regarding the cost of compilation, the increase in the number of fluents and actions is polynomial, ensuring that the existence of a classical plan remains in PSPACE. The STN, used by STP to verify the consistency of temporal constraints, grows linearly with the length of the plan, as grows the plan representation built by *state-of-the-art* planners.

Acknowledgments

This work has been supported by the Maria de Maeztu Units of Excellence Programme (MDM-2015-0502).

References

Allen, J. F. 1983. Maintaining Knowledge about Temporal Intervals. *Commun. ACM* 26(11):832–843.

Baier, J. A.; Bacchus, F.; and McIlraith, S. A. 2009. A heuristic search approach to planning with temporally extended preferences. *Artificial Intelligence* 173(5-6):593–618.

Benton, J.; Coles, A. J.; and Coles, A. 2012. Temporal Planning with Preferences and Time-Dependent Continuous Costs. In *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling, ICAPS 2012, Atibaia, São Paulo, Brazil, June 25-19, 2012*.

Brafman, R. I., and Domshlak, C. 2008. From one to many: Planning for loosely coupled multi-agent systems. In *ICAPS*, 28–35.

Cesta, A., and Oddi, A. 1996. Gaining Efficiency and Flexibility in the Simple Temporal Problem. In *Proceedings of the Third International Workshop on Temporal Representation and Reasoning, TIME-96, Key West, Florida, USA, May 19-20, 1996*, 45–50.

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

Coles, A. J.; Coles, A.; Fox, M.; and Long, D. 2010. Forward-Chaining Partial-Order Planning. In *Proceedings of the 20th International Conference on Automated Planning and Scheduling, ICAPS 2010, Toronto, Ontario, Canada, May 12-16, 2010*, 42–49.

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

Cushing, W.; Kambhampati, S.; Mausam; and Weld, D. S. 2007. When is Temporal Planning Really Temporal? In *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, 1852–1859.

Dechter, R.; Meiri, I.; and Pearl, J. 1991. Temporal Constraint Networks. *Artif. Intell.* 49(1-3):61–95.

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

Helmert, M. 2006. The Fast Downward Planning System. *J. Artif. Intell. Res. (JAIR)* 26:191–246.

Howey, R.; Long, D.; and Fox, M. 2004. VAL: Automatic Plan Validation, Continuous Effects and Mixed Initiative Planning Using PDDL. In *16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2004), 15-17 November 2004, Boca Raton, FL, USA*, 294–301.

Jiménez, S.; Jonsson, A.; and Palacios, H. 2015. Temporal Planning With Required Concurrency Using Classical Planning. In *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling, ICAPS 2015, Jerusalem, Israel, June 7-11, 2015.*, 129–137.

Karpas, E.; Wang, D.; Williams, B. C.; and Haslum, P. 2015. Temporal Landmarks: What Must Happen, and When. In *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling, ICAPS 2015, Jerusalem, Israel, June 7-11, 2015.*, 138–146.

Long, D., and Fox, M. 2003. Exploiting a Graphplan Framework in Temporal Planning. In *Proceedings of the Thirteenth International Conference on Automated Planning and Scheduling (ICAPS 2003), June 9-13, 2003, Trento, Italy*, 52–61.

Marzal, E.; Sebastia, L.; and Onaindia, E. 2014. On the Use of Temporal Landmarks for Planning with Deadlines. In *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling, ICAPS 2014, Portsmouth, New Hampshire, USA, June 21-26, 2014*.

Palacios, H., and Geffner, H. 2009. Compiling uncertainty away in conformant planning problems with bounded width. *Journal of Artificial Intelligence Research* 35:623–675.

Rankooh, M. F., and Ghassem-Sani, G. 2015. ITSAT: An Efficient SAT-Based Temporal Planner. *J. Artif. Intell. Res.* 53:541–632.

Rintanen, J. 2007. Complexity of concurrent temporal planning. In *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling, ICAPS 2007, Providence, Rhode Island, USA, September 22-26, 2007*, 280–287.

Rintanen, J. 2015a. Discretization of Temporal Models with Application to Planning with SMT. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA.*, 3349–3355.

Rintanen, J. 2015b. Models of Action Concurrency in Temporal Planning. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, 1659–1665.

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