Temporal Planning With Required Concurrency Using Classical Planning

Abstract

In this paper we describe a temporal planner that can handle required concurrency as long as time events are asynchronous. The planner is an implementation of the TEMPO algorithm, which compiles each temporal action into two STRIPS actions corresponding to the start and end of the action. Temporal constraints on actions are handled through a modification of the classical planner Fast Downward. All other required mechanisms are coded directly into STRIPS. The paper also presents a novel compilation from temporal planning to STRIPS in the case where required concurrency only appears in the form of single hard envelopes. Compiling a temporal instance into STRIPS gives a lot of freedom in terms of the classical planner or heuristic used to solve the instance, and experimental results are encouraging.

Introduction

While the performance of classical planners dramatically improved in the last decade, their application to real-world problems is still limited. 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 is very expressive, being able to represent many features of real-world problems 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 at the last International Planning Competitions (IPCs) 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 YAHSP2 (Vidal 2011), winner at IPC-2011 and IPC-2014.

In this work we describe a general temporal planner based on the TEMPO algorithm (Cushing, Kambhampati, and Weld 2007), which is able to handle any form of concurrency as long as time events are asynchronous. This planner follows an hybrid approach: on one hand it compiles every temporal action into two STRIPS actions that correspond to the start and the end of the action. On the other it modifies the classical planner Fast Downward (Helmert 2006) to keep track of the temporal constraints on the actions. In addition we present a compilation from temporal planning to STRIPS that handles the particular class of temporal planning in which required concurrency appears only as single hard envelopes (Coles et al. 2009).

The paper is organized as follows. We first introduce planning models and notation, and then present our implementation of TEMPO. Next, we describe our compilation of temporal problems with single hard envelopes. Following the description of the two approaches we present results from experiments, and conclude with a discussion about related work and possible extensions to our approaches. Our planners share many features with CRIKEY (Coles et al. 2009), and we explain exactly how the two differ.

Background

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

Classical Planning

A STRIPS planning instance is a tuple $P = \langle F, A, I, G \rangle$, where F is 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 $\mathsf{pre}(a) \subseteq s$, 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. $G \subseteq I \oplus a_1 \oplus \cdots \oplus a_n$.

A STRIPS planning domain is an abstraction that defines an entire family of STRIPS instances. A planning domain consists of a set of parameterized predicates and action templates. Given a planning domain, an instance P is defined by a set of objects, an initial state I, and a goal condition G. The set of fluents F and set of actions A are implicitly defined by assigning objects to the parameters of predicates and action templates, respectively.

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 STRIPS planning. However, each $a \in A$ is a temporal action composed as follows:

- $d(a)$: integer 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.

The precondition over all $\mathsf{pre}_o(a)$ is also known as the *invariant* of a. Just as for STRIPS planning, we can define temporal planning domains that consist of parameterized predicates and temporal action templates.

In temporal planning the application of actions depends not only on the current state but also on whether other actions are being applied. Therefore a plan Π is not a sequence of actions but a set of action-time pairs (a_i, t_i) such that action a_i starts at time t_i and ends at time $t_i + d(a_i)$. Π is a solution to P when the goal conditions are met after simulating Π starting from I. In a temporal plan Π, we refer to starting or ending a temporal action as an *event*. Simulating Π is then given by applying the actions effects starting from the initial state I and following the ordered sequence of events t_i and $t_i + d(a_i)$. The simulation fails when a precondition does not hold, in that case Π is considered an invalid plan (Howey, Long, and Fox 2004).

In this paper, we assume that the time of each event is unique, i.e. we do not allow actions to start and/or end simultaneously. This is a common assumption made by many temporal planners; nevertheless, in the conclusion we discuss how to relax this assumption.

Inherently Sequential Temporal Planning

A temporal planning instance P is *inherently sequential* if all its solutions can be expressed as sequential plans. Cushing, Kambhampati, and Weld (2007) identify several classes of temporal planning instances that are inherently sequential. In these instances every temporal action can be mapped into a STRIPS action, called *compressed action* (Coles et al. 2009), that simulates the application of an entire temporal action at once. Formally a compressed action is a STRIPS action c_a built from a temporal action $a \in A$ 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 a compressed action is the union of the precondition at start with the preconditions over all and at end not achieved by the effect at start of the temporal action. The effect of a compressed action is the effect at start of the temporal action followed immediately by its effect at end.

A common approach to solving an inherently sequential temporal planning instance comprises tree steps: 1) compiling the temporal instance P into a STRIPS instance $P' = \langle F, A', I, G \rangle$ in which A' is the action set resulting from compressing the actions in P; 2) finding a solution Π' to the new instance P' using a classical planner; and 3) *scheduling* plan Π' to reduce the makespan of the solution, e.g., extracting a partial order from the sequential plan (Veloso, Perez, and Carbonell 1990). Of course temporal planners following this approach are incomplete since they cannot solve, nor indeed represent, instances that require concurrency given that handling concurrency with compressed actions is limited to rescheduling the actions of the sequential plan. This approach is also unsound because the compression of temporal actions may discard fluents relevant for checking whether two actions can overlap.

Planning With Required Concurrency

A temporal planning instance requires concurrency when all its solutions contain two or more actions that need to execute in parallel to achieve the goals. This section describes our novel temporal planner, which is an implementation of the TEMPO algorithm (Cushing, Kambhampati, and Weld 2007), that can solve temporal planning instances with any form of required concurrency, as long as events are asynchronous.

The TEMPO Algorithm

In TEMPO, each copy of a temporal action a has an associated *time variable* τ_a representing the starting time of a. The current situation is represented by a *lifted temporal state*, i.e. a tuple $\mathcal{N} = \langle s, \mathcal{A}, \tau_e, \mathcal{T} \rangle$, where s is a current state on fluents as in classical planning, $\mathcal{A}$ is a set of *active* temporal actions, i.e. that started but not yet ended, τ_e is the time variable associated with the last event (i.e. $\tau_e = \tau_a$ or $\tau_e = \tau_a + d(a)$ for some temporal action a, depending on whether the last event was starting or ending a), and $\mathcal{T}$ is a set of *temporal constraints* on time variables.

To plan, TEMPO considers two successor rules: one for starting an action, called *Fattening*, and one for ending it, called *Advancing Time*. Each time an action a is started in lifted temporal state $\mathcal{N}$, the *Fattening* rule generates a successor $\mathcal{N}' = \langle s', \mathcal{A} \cup \{a\}, \tau_a, \mathcal{T} \cup \{\tau_e < \tau_a\} \rangle$, where $s' = (s \setminus \mathsf{del}_s(a)) \cup \mathsf{add}_s(a)$ is the new state on fluents. A new constraint $\tau_e < \tau_a$ is added to the set of temporal constraints, forcing a to start after the last event e. The *Fattening* rule is applicable in $\mathcal{N}$ if the preconditions at start and over all of a hold in s and starting a does not delete an invariant of another active action.

Likewise, each time an action a is ended in lifted temporal state $\mathcal{N}$, the *Advancing Time* rule generates a successor $\mathcal{N}' = \langle s', \mathcal{A} \setminus \{a\}, \tau_a + d(a), \mathcal{T} \cup \{\tau_e < \tau_a + d(a)\} \rangle$, where $s' = (s \setminus \mathsf{del}_e(a)) \cup \mathsf{add}_e(a)$. The rule removes a from the set of active actions and adds a new constraint to the set of temporal constraints, forcing a to end after the last event e. The *Advancing Time* rule is applicable in $\mathcal{N}$ if $a \in \mathcal{A}$, the precondition at end of a holds in s, and ending a does not delete an invariant of another active action.

TEMPO plans by repeatedly starting and ending actions, updating the lifted temporal state as needed. Once a sequential plan is computed, TEMPO uses the temporal constraints to schedule the starting time of each copy of temporal action a, i.e. assign a value to τ_a. The problem is that this scheduling might fail: 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 makes a classical planer do this work.

Compiling Temporal Actions

We compile each temporal action a into two STRIPS actions, one for starting a and one for ending it, corresponding to the two successor rules of TEMPO. Early temporal planners like LPGP (Long and Fox 2003) already explored splitting temporal actions into STRIPS actions. Coles et al. (2009) identified three key challenges associated with this approach:

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

Our compilation addresses the first two challenges by defining extra preconditions and effects on the resulting STRIPS actions. To reduce the branching factor we 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 classical planner Fast Downward, as explained in the next section.

Consider a temporal instance $P = \langle F, A, I, G\rangle$, 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 invariants. We compile P into a STRIPS 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 $\mathsf{count}_f(l)$ indicating that l active actions have f as an invariant.

As a result, the number of fluents of the STRIPS 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 $I_K = I \cup \{\mathsf{count}_f(0) : f \in F_o\} \cup \{\mathsf{free}_a : a \in A\}$ and $G_K = G \cup \{\mathsf{free}_a : a \in A\}$. In other words, no active actions initially require invariants, and temporal actions are free both initially and in the goal. The new set of actions A_K contains two STRIPS actions, start_a and end_a, for each temporal action $a \in A$. Action start_a is defined as:

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

Likewise, action end_a is defined as:

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

In addition to these effects, action start_a increments $\{\mathsf{count}_f\}_{f \in \mathsf{pre}_o(a)}$, i.e. deletes $\mathsf{count}_f(l)$ and adds $\mathsf{count}_f(l+1)$ for some l (for simplicity, we implement this as a conditional effect, which is technically not STRIPS but can easily be compiled to STRIPS). Likewise, action end_a decrements $\{\mathsf{count}_f\}_{f \in \mathsf{pre}_o(a)}$.

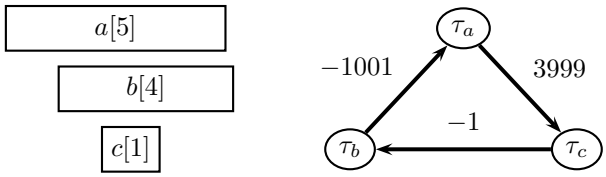

Figure 1: Example event sequence and associated STN.

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 cost c represents a constraint $\tau_j - \tau_i \le c$. Dechter, Meiri, and Pearl (1991) show that scheduling the time variables fails if and only if an STN contains negative cycles. Otherwise, the earliest feasible assignment of each time variable τ_i is given by $-d_{i0}$, where d_{i0} is 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).

Figure 1 shows three overlapping actions a, b, and c with duration 5, 4, and 1, respectively, and the STN associated to their events (the example is taken from Cushing, Kambhampati, and Weld, 2007). We express the time of events in thousandths of time units and require each pair of events to be separated by at least 1 thousandth. The set of constraints imposed by pairs of consecutive events are:

1. $\tau_a + 1 \le \tau_b$,
2. $\tau_b + 1 \le \tau_c$,
3. $\tau_c + 1 \le \tau_c + 1000$,
4. $\tau_c + 1000 + 1 \le \tau_a + 5000$,
5. $\tau_a + 5000 + 1 \le \tau_b + 4000$.

Note that the fifth constraint subsumes the first, and that the third is trivially satisfied. There are 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 action a starts first.

Ideally our compilation should represent temporal constraints in PDDL and use STRIPS actions to simulate Floyd-Warshall's algorithm. However, this approach appears doomed to fail: the number of fluents and actions quickly becomes unmanageable. Instead, our approach implements the scheduling mechanism as part of the Fast Downward planner (Helmert 2006). Apart from the current state, search nodes in Fast Downward contain additional information such as the parent node in the search graph. Our modification extends the information stored in the search nodes of Fast Downward to fully represent a current lifted temporal state of the TEMPO algorithm, including a list of active actions and an STN representing the temporal constraints. Moreover, each time a compiled action is applied, either a start or an end action, 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 potentially incomplete for two other reasons: 1) the bound K on the number of concurrent active actions; and 2) the fact that the planner only tracks the state on fluents, potentially causing lifted temporal states to be incorrectly pruned. Since the state on fluents does not include the STN, the planner may decide not to open a lifted temporal state that would have led to a solution. The first problem can be addressed by iteratively increasing the bound K, while the second problem does not seem to occur much in practice.

Planning With Single Hard Envelopes

There is a particular subclass of temporal planning with required concurrency for which the preservation of temporal constraints can be encoded in the STRIPS actions. This section describes a compilation from temporal planning to STRIPS for temporal planning in which required concurrency appears only in the form of *single hard envelopes* (Coles et al. 2009). A single hard envelope arises when a single action (the *envelope*) creates a window of opportunity within which another action or set of actions (the *content*) must start *and* end. Single hard envelopes are able to model unary resources available over a time window and, interestingly, include all instances from the IPC that require concurrency, i.e. instances from the MATCHCELLAR, TURN&OPEN and TMS domains.

The concurrency graph

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 to pairs of temporal actions: in order for a to be a single hard envelope (or envelope for short), there has to exist a temporal action b that has f as invariant and duration shorter than a, i.e. a is an envelope for b if and only if $\mathsf{add}_s(a) \cap \mathsf{del}_e(a) \cap \mathsf{pre}_o(b) \neq \emptyset$ and $d(b) < d(a)$. This definition allow us to construct a *concurrency graph* with the temporal action templates of the domain as nodes. There is an edge between action templates a and b if and only if there exists an assignment of parameters to a and b (inducing temporal actions a' and b') such that a' is an envelope for b'. 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.

We use the concurrency graph to infer which actions can be envelopes and on what level each action should appear. Let D be the depth of the concurrency graph, i.e. the longest directed path between any pair of nodes. For each action $a \in A$, let $0 \leq l(a) \leq D$ be its level, i.e. the length of incoming directed paths to the associated action template. In this paper we assume that the level of each temporal action is unique, but relaxing this assumption is relatively straightforward. In Figure 2, action template fire-kiln1 has level 0, treat-ceramic2 has level 2, and make-structure has level 0.

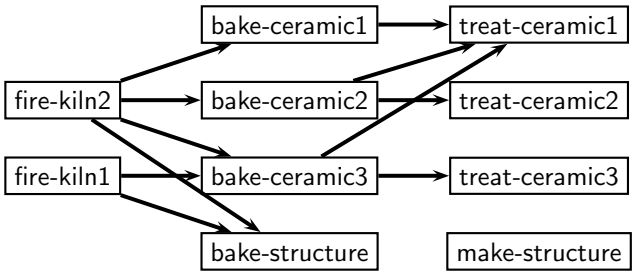

Figure 2: The concurrency graph of the `tms` domain.

Compiling Single Hard Envelopes into STRIPS

Our compilation models this particular case of required concurrency using a stack of active actions, i.e. actions that started but not yet ended. Only envelope actions and their remaining duration are stacked; all other actions are compiled into compressed actions.

Formally, a temporal instance $P = \langle F, A, I, G \rangle$ is compiled into a STRIPS instance $P' = \langle F', A', I', G' \rangle$. The set F' contains all fluents in F_D as defined in the previous section, i.e. fluents of type free_a and active_a for temporal actions and $\mathsf{count}_f(l)$ for invariants and levels, with the bound on active actions given by the depth D of the concurrency graph. In addition, F' includes additional fluents to preserve the temporal constraints produced by single hard envelopes, where T is the maximum duration of envelopes:

- For each $0 \leq l \leq D$, a fluent $\mathsf{stack}(l)$ indicating that the stack contains l active actions.
- For each $0 \leq l < D$ and $1 \leq t \leq T$, a fluent $\mathsf{remain}(l, t)$ indicating that t time units remain of the execution of the active action on level l of the stack.
- For each pair of integers (t, u) such that $1 \leq u < t \leq T$, a static fluent $\mathsf{sub}(t, u, t - u)$ representing subtraction.

We remark that stack levels and time units could be implemented using numeric fluents, but in experiments we obtained the best results with planners that do not support numeric fluents. The total number of fluents is given by $|F'| = |F| + 2|A| + (D + 1)(|F_o| + 1) + DT + \frac{T(T-1)}{2}$.

The initial state represents an empty stack and initializes the subtractions of durations, i.e. $I' = I_D \cup \{\mathsf{stack}(0)\} \cup \{\mathsf{sub}(t, u, t - u) : 1 \leq u < t \leq T\}$, with I_D defined as in the previous section, replacing the bound K with D. The goal condition verifies that goals are achieved and that the stack is empty, i.e. $G' = G_D \cup \{\mathsf{stack}(0)\}$.

For each envelope $a \in A$ and each pair of time units (t, u), A' contains two STRIPS actions, $\mathsf{start}_a(t, u)$ and $\mathsf{end}_a(t)$, that correspond to starting and ending action a. Action $\mathsf{start}_a(t, u)$ applies all effects at start of a and pushes

action a onto the stack:

$$\begin{aligned}\mathsf{pre}(\mathsf{start}_a(t,u)) &= \mathsf{pre}(\mathsf{start}_a) \cup \{\mathsf{stack}(l(a))\} \\ &\cup \{\mathsf{remain}(l(a)-1,t), \mathsf{sub}(t,u,d(a))\}, \\ \mathsf{add}(\mathsf{start}_a(t,u)) &= \mathsf{add}(\mathsf{start}_a) \cup \{\mathsf{stack}(l(a)+1)\} \\ &\cup \{\mathsf{remain}(l(a),d(a)), \mathsf{remain}(l(a)-1,u)\}, \\ \mathsf{del}(\mathsf{start}_a(t,u)) &= \mathsf{del}(\mathsf{start}_a) \cup \{\mathsf{stack}(l(a))\} \cup \\ &\cup \{\mathsf{remain}(l(a)-1,t)\}.\end{aligned}$$

Here, start_a is the compiled action from the previous section. In case another action is on top of the stack (i.e. $l(a) > 0$), t and u are used to update its remaining time, subtracting the duration of a. If $l(a) = 0$, parameters t and u and the associated preconditions and effects are not needed and can be removed. The stack size is also incremented and a is added to level $l(a)$ of the stack, setting its remaining time to $d(a)$.

Action $\mathsf{end}_a(t)$ applies all effects at end of a and pops action a from the stack:

$$\begin{aligned}\mathsf{pre}(\mathsf{end}_a(t)) &= \mathsf{pre}(\mathsf{end}_a) \cup \{\mathsf{stack}(l(a)+1)\} \\ &\cup \{\mathsf{remain}(l(a),t)\}, \\ \mathsf{add}(\mathsf{end}_a(t)) &= \mathsf{add}(\mathsf{end}_a) \cup \{\mathsf{stack}(l(a))\}, \\ \mathsf{del}(\mathsf{end}_a(t)) &= \mathsf{del}(\mathsf{end}_a) \cup \{\mathsf{stack}(l(a)+1)\} \\ &\cup \{\mathsf{remain}(l(a),t)\}.\end{aligned}$$

Popping a from the stack is only possible if a is active and on top of the stack, and the result is decrementing the stack size and making a free, erasing its remaining time.

For each action $a \in A$ that is not an envelope and each pair of time units (t,u), the set A' contains a compressed action $c_a(t,u)$ defined as follows:

$$\begin{aligned}\mathsf{pre}(c_a(t,u)) &= \mathsf{pre}(c_a) \cup \{\mathsf{stack}(l(a))\} \cup \\ &\cup \{\mathsf{remain}(l(a)-1,t), \mathsf{sub}(t,u,d(a))\}, \\ \mathsf{add}(c_a(t,u)) &= \mathsf{add}(c_a) \cup \{\mathsf{remain}(l(a)-1,u)\}, \\ \mathsf{del}(c_a(t,u)) &= \mathsf{del}(c_a) \cup \{\mathsf{remain}(l(a)-1,t)\}.\end{aligned}$$

Here, c_a is the compressed action of temporal action a. As before, in case $l(a) > 0$ the remaining duration of the action on top of the stack is updated using t and u.

There are two cases for which our compilation is incomplete: 1) the temporal planning problem contains other sources of required concurrency different from single hard envelopes; and 2) there exist actions with two or more invariants, provided by different envelopes (the stack only admits one envelope at a time on each level). If neither condition presents itself, the compilation is both sound and complete.

Theorem 0.1 *The compilation is sound and complete if envelopes are the only sources of required concurrency and no actions require two or more envelopes.*

Proof sketch Soundness: Given a plan for the STRIPS planning instance P', we first schedule all actions at level 0 of the stack in order, which does not require concurrency. 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 remaining duration of an envelope cannot fall below 1.

Completeness: Since there are no other sources of required concurrency, we can always rearrange the plan of a temporal planning instance with single hard envelopes such that actions are either applied in sequence or nested inside envelopes. If such a plan exists, our compilation will find it (assuming the planner used to solve P' is complete).

Results

The evaluation is carried out in all domains of the temporal track of IPC-2014 except MAPANALYSER and RTAM. The latter two were excluded since they contain actions with variable duration, something not handled in the current version of our planner. However, both domains are inherently sequential and variable action duration is thus only needed during scheduling, which should not be hard to implement.

In addition, we added the *DriverLog Shift* (DLS) domain (Coles et al. 2009) which requires concurrency in the form of single hard envelopes, a domain based on Allen's interval algebra (AIA) (Allen 1983) that models possible relations between time intervals, and a domain based on the example in Figure 1 (EXAMPLE). As a result, 5 domains are inherently sequential (DRIVERLOG, FLOORTILE, PARKING, SATELLITE, STORAGE), 4 domains can be solved using simple hard envelopes (TURN&OPEN, TMS, MATCHCELLAR, DLS), while 2 domains involve general required concurrency (AIA and EXAMPLE).

Our temporal planner (TP) detects required concurrency in the form of single hard envelopes, and only applies the general compilation in the remaining two domains. Our compilation for single hard envelopes (TP') was solved using the LAMA-2011 setting of Fast Downward. 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 inherently sequential domains. We also included POPF2, the runner-up at IPC-2011. Table 1 shows, for each planner, accumulated make-span (in seconds) and number of problem solved per domain.

When looking at coverage, our planners show good performance both in inherently sequential domains and in domains that require concurrency. This stable behavior across domains is not observed in the other planners. YAHSP3-MT performs well in inherently sequential domains but cannot handle domains with required concurrency. In contrast, ITSAT solves instances in all domains that require concurrency but performs poorly in the remaining domains.

Our best planner (TP') has higher coverage (111) than YAHSP3-MT (the winner) in domains from IPC-2014, even though we excluded two domains. On the other hand, YAHSP3-MT apparently comes with a better scheduler since the makespan tends to be shorter. Interestingly, YAHSP3-MT performs worse in our comparison than at IPC-2014 since we included more domains that require concurrency (only 3 were included at IPC-2014). The reason our planners perform poorly in TMS is that the preprocessing of Fast Downward runs out of memory due to large instances; in smaller instances from IPC-2011 search is actually quite fast once preprocessing is done.

	TP	TP'	ITSAT	TFD	POPF2	YAHSP3-MT
DRIVERLOG	47689/20	11646/20	140/1	-	-	786/3
FLOORTILE	533/4	327/5	1447/20	-	-	251/5
PARKING	357/20	256/20	75/6	101/20	60/14	66/20
SATELLITE	7723/15	4699/18	-	3478/17	316/3	4765/20
STORAGE	-	386/9	-	-	-	317/9
TURN&OPEN	3385/14	2932/19	264/7	1859/18	316/7	-
TMS	-	-	316/16	-	-	-
MATCHCELLAR	1451/20	1450/20	1143/20	1145/20	109/3	-
DLS	34/1	596/12	4724/20	83/1	186/2	-
AIA	16/6	3/3	132/6	42/6	52/6	-
EXAMPLE	41/1	-	-	20/1	-	-
	61209/101	22298/126	8223/96	6710/83	1039/35	6185/57

Table 1: Accumulated make-span (in seconds) / number of problems solved per domain.

Related work

Several authors have proposed compiling temporal planning actions into multiple STRIPS 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 earlier. As mentioned, our planner is more or less a direct implementation of TEMPO (Cushing, Kambhampati, and Weld 2007).

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). Apart from the state on fluents, CRIKEY maintains a set of *envelopes* (each of which may involve multiple envelope actions, not just a single envelope). Each time a temporal action is started or ended, CRIKEY updates the set of envelopes (potentially adding new envelopes). An action is only included in an envelope if its interaction with other actions in the envelope induces a temporal constraint. Just like our planner, CRIKEY uses STNs to detect unsatisfiable temporal constraints, but unlike our planner, CRIKEY maintains multiple STNs, one per envelope. Another difference is that several of CRIKEY's mechanisms are not compiled into STRIPS, but checked externally on envelopes.

We believe that our approach has two advantages compared to CRIKEY. First, compiling as much as possible of the temporal planning instance into STRIPS potentially enables a forward-search planner to compute better heuristic estimates regarding the distance-to-go to the goal. In addition, implementing the scheduling mechanism as part of Fast Downward makes it easy to experiment with different heuristics or combinations of heuristics. Second, it appears simpler to base temporal constraints on the event order and let interactions be checked automatically by the planner, as opposed to maintaining multiple STNs and explicitly detect interactions among actions.

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 STRIPS, using the precondition $\{\mathsf{remain}(l(a)-1,t), \mathsf{sub}(t,u,d(a))\}$ to test whether the remaining duration t of an envelope a' is sufficiently large to accommodate an action a with duration $d(a)$ inside a'. Compiling the entire temporal instance into STRIPS makes it possible to use any classical planner to solve the instance.

Rintanen (2007) proposed another compilation from temporal planning to STRIPS that explicitly represents time units as objects. The compilation includes STRIPS actions that start temporal actions, and keeps track of the time elapsed in order to determine when temporal actions should end. Time is not divided into parts, 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 two contributions to temporal planning. First we implement the TEMPO algorithm introduced by Cushing, Kambhampati, and Weld (2007). The result is a general temporal planner able to handle any form of concurrency as long as time events are asynchronous. This planner follows an hybrid approach: on one hand it compiles every temporal action into two STRIPS actions that correspond to the start and the end of the temporal action. On the other it modifies the classical planner Fast Downward (Helmert 2006) to keep track of the temporal constraints on actions. The second contribution is a compilation from temporal planning to STRIPS that handles the particular class of temporal planning in which required concurrency appears only in the form of single hard envelopes.

Our approach assumes that time points are unique, i.e. that actions do not start and/or end simultaneously. The problem with relaxing this assumption is that simultaneous effects might be conflicting, which must be tested during planning. One way to do so is to separate action selection from action execution during planning: first select the actions to be started and/or ended, and then apply the effect of these actions at once, simultaneously checking whether these effects are consistent. This mechanism requires several duplicates of each fluent and is likely to slow down planning; nevertheless it is something that we want to explore in the future.

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.; and s. Sannerand S. Yoon. 2012. A survey of the seventh international planning competition. *AI Magazine* 33(1):83–88.

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

Howey, R.; Long, D.; and Fox, M. 2004. VAL: Automatic Plan Validation, Continuous Effects and Mixed Initiative Planning using PDDL. In *The 16th IEEE International Conference on Tools with Artificial Intelligence*, 294–301.

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. 2011. YAHSP2: Keep It Simple, Stupid. In *Proceedings of the 7th International Planning Competition (IPC-2011)*, 83–90.