Master in Intelligent Interactive Systems
Universitat Pompeu Fabra

Resolution of Concurrent Planning Problems using Classical Planning

Daniel Furelos Blanco

Supervisor: Anders Jonsson

September 2017

Master in Intelligent Interactive Systems
Universitat Pompeu Fabra

Resolution of Concurrent Planning Problems using Classical Planning

Daniel Furelos Blanco

Supervisor: Anders Jonsson

September 2017

Contents

Acknowledgement

I would like to express my sincere gratitude to my supervisor, Anders Jonsson, for his guidance, patience and giving me the opportunity to join the Artificial Intelligence & Machine Learning Research Group for some months.

I am very grateful to my parents and my brother, who have always supported me.

Abstract

In this work, we present new approaches for solving multiagent planning and temporal planning problems. These planning forms are two types of concurrent planning, where actions occur in parallel. The methods we propose rely on a compilation to classical planning problems that can be solved using an off-the-shelf classical planner. Then, the solutions can be converted back into multiagent or temporal solutions.

Our compilation for multiagent planning is able to generate concurrent actions that satisfy a set of concurrency constraints. Furthermore, it avoids the exponential blowup associated with concurrent actions, a problem that many multiagent planners are facing nowadays. Incorporating similar ideas in temporal planning enables us to generate temporal plans with simultaneous events, which most state-of-the-art temporal planners cannot do.

In experiments, we compare our approaches to other approaches. We show that the methods using transformations to classical planning are able to get better results than state-of-the-art approaches for complex problems. In contrast, we also highlight some of the drawbacks that this kind of methods have for both multiagent and temporal planning.

We also illustrate how these methods can be applied to real world domains like the smart mobility domain. In this domain, a group of vehicles and passengers must self-adapt in order to reach their target positions. The adaptation process consists in running a concurrent planning algorithm. The behavior of the approach is then evaluated.

Keywords: Classical planning; Concurrent planning; Multiagent planning; Temporal planning

Chapter 1

Introduction

In this chapter, the research concerning this thesis is contextualized and motivated. Finally, the structure of the report is described.

1.1 Context

Concurrent planning is one of the most promising planning forms. Solutions to concurrent planning problems consist of sequences of joint actions. Each joint action is formed by atomic actions that are simultaneously performed. Restrictions on concurrent actions, called *concurrency constraints*, can be imposed to ensure they are well-formed, i.e. that there are not inconsistencies between them.

There are many real world applications where concurrency is required. For example, RoboCup [1] is a competition where two teams of robots play football. The robots act simultaneously in order to score a goal. Another simple example could consist of two robots that want to move a table from one room to another. To do so, they must lift the table at the same time; otherwise, the table would be tipped and the objects on it would fall.

Concurrency is inherent to certain forms of planning like *temporal planning*. Moreover, it can be also seen as an extension of other planning forms like *multiagent planning*.

Research in multiagent planning has seen a lot of progress in recent years, in part due to the first competition of distributed and multiagent planners, CoDMAP 2015 [2]. Many recent multiagent planners are based on the MA-STRIPS formalism [3], and can be loosely classified into one of two categories: centralized and distributed. In CoDMAP 2015, the most successful centralized planners were ADP [4], MAP-LAPKT [5] and CMAP [6], while prominent distributed planners included PSM [7], MAPlan [8] and MH-FMAP [9].

Although concurrency is a major problem in real world applications, none of the domains in CoDMAP has concurrency constraints. Besides, the solutions produced by most of the above planners are *sequential*, i.e. at each time step, a single agent takes an action. Actually, compared to other settings, *concurrent multiagent planning* has been less studied in the literature. There are several approaches that allow for concurrent actions [9, 10], but few that are capable of reliably generating plans that solve complex concurrent multiagent problems. A notable exception is the work by Crosby *et al.* [11], who associate concurrency constraints with the *objects* of a multiagent planning problem and transform the problem into a sequential, single-agent problem that can be solved using an off-the-shelf classical planner.

In the case of temporal planning, the International Planning Competition (IPC) has included a temporal track for many years. Thus, the work on this form of planning is more advanced than for concurrent multiagent planning. However, although concurrency is inherent to it, no temporal planner can currently deal with simultaneous events, i.e. events that occur exactly at the same time.

1.2 Problem

As explained in Section 1.1, both temporal planning and multiagent planning can be related to concurrent planning. We provide definitions for concurrent planning that can be used to define both temporal and multiagent planning problems.

As it has not been deeply studied so far, we focus on the problem of concurrent multiagent planning, where several agents can act at each time step. This problem

is challenging for different reasons: the number of concurrent actions is worst-case exponential in the number of agents, and we must deal with an explicit way to specify concurrency constraints [12, 13, 14] to guarantee that concurrent actions are well-formed. To do so, we propose a method for compiling the concurrent multiagent planning problem into a classical planning problem. In the new problem, the number of actions to choose from at each step is polynomial, and can be solved using any off-the-shelf classical planner.

Compilation from temporal planning to classical planning has already been used in the past [15]. We propose a new compilation that supports arbitrary concurrency and simultaneous events.

To sum up, the goal of this thesis is to provide methods for converting concurrent planning problems into classical problems while respecting concurrency constraints.

1.3 Structure of the Report

In Chapter 2, we present some background regarding different forms of planning and the relationship between them.

Chapter 3 introduces a compilation for converting concurrent multiagent problems into classical problems while avoiding the exponential blowup in the total number of joint actions. A temporal planner that is able to deal with simultaneous events is presented in Chapter 4. This planner also relies in a compilation to classical planning. Chapter 5 shows a real application of concurrent planning: the smart mobility domain.

Approaches related to the works explained in the previous chapters are introduced in Chapter 6. Finally, conclusions and future work are detailed in Chapter 7.

Chapter 2

Background

Planning is the *model-based* approach to autonomous behavior where the agent selects the action to do next using a model of how actions and sensors work, what is the current situation, and what is the goal to be achieved. The main challenge in this kind of approach consists in dealing with problems that can be computationally intractable (even the simplest ones) [16].

In the following subsections, we will introduce four different forms of planning: classical planning, concurrent planning, concurrent multiagent planning and temporal planning.

2.1 Classical Planning

Given a set of propositional variables or *fluents* F, a *literal* l is a valuation of a fluent in F, i.e. $l = f$ or $l = \neg f$ for some $f \in F$. A set of literals L represents a partial assignment of values to fluents in F; we assume that literal sets do not assign conflicting values to any fluent. Given L, let $\neg L = \{\neg l : l \in L\}$ be the complement of L. Finally, we define $L(F)$ as the set of all literals that can be formed from a set of fluents F, i.e. $L(F) = \{f, \neg f : f \in F\}$. We abuse notation and assume that no subset of $L(F)$ contains conflicting literals, i.e. for a given $f \in F$, they do not contain both f and $\neg f$.

A *state* $s \subseteq L(F)$ is a subset of literals. We abuse notation and define states only in terms of the fluents that are true. A subset of literals $L(F)$ *holds* in state s if and only if $L(F) \subseteq s$.

A *classical planning* problem is a tuple $\Pi = \langle F, A, I, G \rangle$, where F is a set of fluents, A is a set of actions, $I \subseteq L(F)$ is an initial state, and $G \subseteq L(F)$ is a goal condition (usually satisfied by multiple states). Each action $a \in A$ has a set of preconditions $\mathsf{pre}(a) \subseteq L(F)$ and a set of positive and negative effects $\mathsf{eff}(a) \subseteq L(F)$, each a subset of literals. Action a is *applicable* in state $s \subseteq F$ if and only if $\mathsf{pre}(a)$ holds in s, and applying a in s results in a new state $s \oplus a = (s \setminus \neg\mathsf{eff}(a)) \cup \mathsf{eff}(a)$.

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

We sometimes define classical planning problem whose actions include *conditional effects*. For the notation to be consistent, we include conditional effects as part of the effects of an action, i.e. each action $a \in A$ has a set of conditional effects $\mathsf{cond}(a)$ and, as before, a set of preconditions $\mathsf{pre}(a)$. Each conditional effect $C \rhd E \in \mathsf{cond}(a)$ consists of a condition $C \subseteq L(F)$ and an effect $E \subseteq L(F)$[1]. As before, action a is applicable in state s if and only if the precondition holds in s, i.e. $\mathsf{pre}(a) \subseteq s$. The actual effect $\mathsf{eff}(s, a)$ of a is conditional on s, and is composed as

$$\mathsf{eff}(s, a) = \bigcup_{C \rhd E \in \mathsf{cond}(a),\, C \subseteq s} E,$$

i.e. effects whose condition holds in s. As before, the result of applying a in s is a new state $s \oplus a = (s \setminus \neg\mathsf{eff}(s, a)) \cup \mathsf{eff}(s, a)$. Although an action with conditional effects can be compiled into multiple actions without conditional effects, it may cause an exponential increase in the number of actions [17].

[1]Note that those effects that are not conditioned to anything, can be expressed as conditional effects by setting C to the empty set $\emptyset$.

2.2 Concurrent Planning

Concurrent planning is the extension of classical planning where each action is a combination of atomic actions that are performed simultaneously. Actions in concurrent planning are named *joint actions* or *concurrent actions*.

Given a set of atomic actions B, the size of the set of actions A in a concurrent planning problem will be the number subsets of B, i.e. $|A| = 2^{|B|}$. Therefore, we can define a concurrent planning problem as a classical planning problem $\Pi = \langle F, A, I, G \rangle$ where $A = 2^B$ is the power set of B, i.e. the set of all subsets of B.

Given a concurrent action $a = \left(a^1, \ldots, a^k\right)$, we define the precondition and the effect of a as the union of preconditions and effects of the constituent atomic actions[2]:

$$\mathsf{pre}(a) = \bigcup_{j=1}^{k} \mathsf{pre}(a^j),\ \mathsf{eff}(s, a) = \bigcup_{j=1}^{k} \mathsf{eff}(s, a^j)$$

Concurrent actions can be ill-defined. For example, two atomic actions of a joint action a could have incompatible effects, i.e. one of them adds literal l, and the other one deletes l. To make sure that concurrent actions are well-defined, *concurrency constraints* are used. Constraints can be defined on pairs of atomic actions, and they can be of two types:

- *Positive concurrency constraints* state that two atomic actions *must* be done at the same time.
- *Negative concurrency constraints* state that two atomic actions *cannot* be done at the same time.

If none of the previous constraints is specified for a given pair of actions, there is no restriction on their concurrency.

[2]Note that we use the notation of effects that depend on the current state s introduced in Section 2.1.

By using concurrency constraints, the set A of possible concurrent actions is a subset of 2^B.

In Section 2.3, we introduce state-of-the-art notation of concurrency constraints in multiagent planning problems.

2.3 Concurrent Multiagent Planning

In this section, we introduce the main concepts concerning multiagent planning. We concretely focus on centralized multiagent planning: the agents share a common goal and planning consists in achieving this goal.

2.3.1 Definition

A *multiagent planning* problem (MAP) is a tuple $\Pi = \left\langle N, F, \{A^i\}_{i=1}^n, I, G \right\rangle$, where $N = \{1, \ldots, n\}$, and A^i is the action set of agent $i \in N$. The set of fluents F, the initial state I and the goal condition G are defined as for classical planning.

Concurrent multiagent planning can be viewed as a special case of concurrent planning (see Section 2.2) where the set of atomic actions is partitioned as $B = A^1 \cup \ldots \cup A^n$.

A concurrent multiagent planning problem can be represented as a classical planning problem $\Pi = \langle F, A, I, G \rangle$ where $A \subseteq A^1 \times \cdots \times A^n$ is the set of concurrent actions satisfying the concurrency constraints.

2.3.2 Concurrency Constraints

As stated in Section 2.2, concurrency constrains are usually introduced for resolving conflicting effects. In this section, we introduce different existing approaches for modeling these constraints.

Boutilier and Brafman (2001) extended the STRIPS language to make the definition of MAPs (including concurrency constraints) possible [12]. The first parameter always corresponds to the associated agent. Then, they introduced the notion of

concurrent action list. These lists are attached to each actions' preconditions or to conditional effects, and they contain both state variables and references to concurrently executed actions. If an action a^i appears in the list of action a^j, then the two actions must be concurrently performed (positive concurrency constraint). If an action a^i appears negated in the list of action a^j, then they cannot be concurrently performed (negative concurrency constraint).

Although Boutilier and Brafman's method is a natural way of modeling concurrent actions, they did not use explicit quantifiers (e.g. for all, exists) for many variables which appear out of scope. For this reason, Kovacs (2012) extended the PDDL language to propose an standard definition of MAPs [13]. In this approach, concurrent actions are directly referenced in actions' preconditions and conditional effects instead of using an explicit concurrency action list. Moreover, as stated before, no variable is out of scope unlike in the original approach.

To give an example of the notation proposed by Kovacs, we use the TableMover domain created by Boutilier and Brafman (2001) [12][3]. In this domain, two agents move blocks between rooms. There are two possible strategies:

1. They move blocks one by one using their arms.

2. They put all the blocks on a table and move the table from one room to another. Both agents must lift the table at the same time; otherwise, the table will be tipped and the blocks will fall. However, in case they want to quickly leave the blocks on the floor, the table can be tipped by lowering only one side of it.

The lift-side action in Figure 1 models an agent ?a who wants to lift one side ?s of the table. To do the action, ?a has to be at side ?s of the table, the side ?s must be down (i.e. on the floor) and the agent cannot be holding anything. Moreover, the concurrent action list indicates that another agent ?a2 cannot lower side ?s2 at the same time. When the action is applied, ?s is not longer touching the floor (i.e. it is

[3]See Appendix A for a complete specification of the TableMover domain.

up), and ?a is busy lifting ?s. The action also has a conditional effect (represented by the when clause): if no agent ?a2 lifts side ?s2 of the table, then all blocks on the table will fall to the floor.

Note that we have used forall quantifiers in the concurrency constraints. It might look unnatural that we do not explicitly state, for example, that ?a2 and ?s2 must be different from ?a and ?s respectively. Writing such constraints usually requires complex formulations. However, in this case it is not necessary to impose such restrictions because (1) if nobody is lowering a side of the table, we can lift ?s, and (2) if nobody apart from ?a is lifting a side of the table that was originally on the floor, then the blocks will fall.

Crosby *et al.* (2014) also extended the PDDL language to define MAPs with concurrency constraints [11]. The authors use the MAZE domain [18] to show their approach. This domain consists of a grid of interconnected locations. Each agent must move from an initial location to a target location. The connection between two adjacent locations can be one of the following:

- Door: can only be traversed by one agent at a time. Some are initially locked. They are unlocked by pushing a specific switch, placed anywhere in the maze.
- Bridge: can be crossed by multiple agents at once, but it is destroyed after being crossed.
- Boat: can only be used by two or more agents in the same direction.

The authors do not use concurrent action lists as concurrency constraints. Instead, these constraints are defined as affordances on subsets of objects. Affordances are defined as *intervals*, e.g. the affordance on the subset of objects {location, boat} in the MAZE domain is $[2, \infty]$, representing that at least two agents have to row the boat between the same two locations at once (since the boat always travels between the same two locations, only one location is needed to indicate the direction of movement), while the affordance on {door} is $[1, 1]$, representing that at most one agent can traverse the door at once.

```
(:action lift-side
    :agent ?a - agent
    :parameters (?s - side)
    :precondition (and
                      (at-side ?a ?s)
                      (down ?s)
                      (handempty ?a)
                      (forall
                             (?a2 - agent ?s2 - side)
                             (not (lower-side ?a2 ?s2))
                      )
                 )
    :effect (and
                (not (down ?s))
                (up ?s)
                (lifting ?a ?s)
                (not (handempty ?a ?s))
                (forall
                       (?b - block ?r - room ?s2 - side)
                       (when
                            (and
                                 (inroom Table ?r)
                                 (on-table ?b)
                                 (down ?s2)
                                 (forall
                                        (?a2 - agent)
                                        (not (lift-side ?a2 ?s2))
                                 )
                            )
                            (and
                                 (on-floor ?b)
                                 (inroom ?b ?r)
                                 (not (on-table ?b))
                            )
                       )
                )
           )
)
```

Figure 1: Definition of the TableMover's action lift-side using Kovac's (2012) notation.

```
(:action row
    :agent ?a - agent
    :parameters (?b - boat ?x - location ?y - location)
    :precondition (and
                      (at ?a ?x)
                      (has-boat ?b ?x ?y)
                  )
    :effect (and
                (at ?a ?y)
                (not (at ?a ?x))
            )
)

(:concurrency-constraint v2
    :parameters (?bb - boat ?xx - location)
    :bounds (2 inf)
    :actions ( (row 1 2) )
)
```

Figure 2: Definition of the MAZE's cross action using Crosby *et al.* (2014) notation.

The row action in Figure 2 models an agent ?a that wants to go from location ?x to location ?y using boat ?b[4]. To do the action, agent ?a must be at ?x and the boat ?b must connect ?x and ?y.

Note that the concurrency constraint is given a name, v2. They specify the list of parameters on which the affordance is defined (a boat ?bb and a location ?xx), as well as the bounds of the constraint which are $[2, \infty]$. Finally, a list of actions is given. In this case, only the row action is associated to this constraint. The numbers next to the action name represent the parameter number in the original action (?a $\rightarrow 0$, ?b $\rightarrow 1$, ?x $\rightarrow 2$ and ?y $\rightarrow 3$) and their association with the constraint parameters, i.e. ?bb is associated to parameter 1 in row, while ?xx is associated to parameter 2.

Kovacs's approach is more expressive than Crosby's *et al.* because of the following reasons:

1. In Kovacs's, actions can be used in conditional effects, while Crosby's *et al.* cannot.

[4]See Appendix B for a complete specification of the MAZE domain.

```
(:action row
    :agent ?a - agent
    :parameters (?b - boat ?x - location ?y - location)
    :precondition (and
                      (at ?a ?x)
                      (has-boat ?b ?x ?y)
                      (exists
                             (?a2 - agent)
                             (and
                                  (not (= ?a ?a2))
                                  (row ?a2 ?b ?x ?y)
                             )
                      )
                      (forall
                             (?a2 - agent)
                             (not (row ?a2 ?b ?y ?x))
                      )
                 )
    :effect (and
                (at ?a ?y)
                (not (at ?a ?x))
            )
)
```

Figure 3: Definition of the MAZE's cross action using Kovacs (2012) notation.

2. Crosby *et al.* can represent concurrency constraints on multiple action templates as long as they are defined on the same subset of objects. In contrast, in Kovacs's, concurrency constraints can be specified between any pair of actions.

For these reasons, Crosby's *et al.* approach cannot model the TABLEMOVER domain, while the Kovacs's can. On the other hand, Kovacs's can model the MAZE domain as shown in Figure 3.

2.4 Temporal Planning

A *temporal planning* problem is also a tuple $\Pi = \langle F, A, I, G \rangle$, where F, I, and G are defined as for classical planning. However, each element $a \in A$ is a *temporal action* composed of:

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

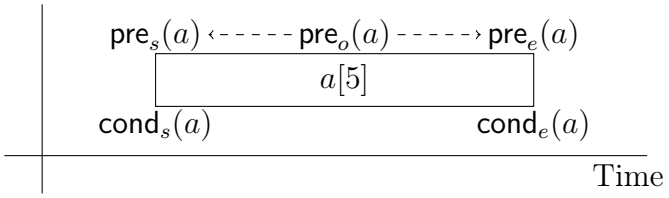


Figure 4: Example temporal action a with duration $d(a) = 5$.

spectively.

- $\mathsf{cond}_s(a), \mathsf{cond}_e(a)$: conditional effects of a at start and at end. Note that they respectively replace $\mathsf{eff}_s(a)$ and $\mathsf{eff}_e(a)$ as we did in classical planning (see Section 2.1).

Although a has a duration, its effects $\mathsf{cond}_s(a)$ and $\mathsf{cond}_e(a)$ apply instantaneously at the start and at the end respectively. The preconditions $\mathsf{pre}_s(a)$ and $\mathsf{pre}_e(a)$ are also checked instantaneously, but the precondition $\mathsf{pre}_o(a)$ has to hold for the entire duration of a. Figure 4 shows a graphical representation of a temporal action a, with preconditions appearing above a, and effects below.

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 as follows [19]:

- $\langle \mathsf{pre}(\mathsf{start}_a) = \mathsf{pre}_s(a), \mathsf{cond}(\mathsf{start}_a) = \mathsf{cond}_s(a) \rangle$
- $\langle \mathsf{pre}(\mathsf{end}_a) = \mathsf{pre}_e(a), \mathsf{cond}(\mathsf{end}_a) = \mathsf{cond}_e(a) \rangle$

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. We use the term *context* to refer to a precondition over all, while F_o is the set of fluents used in contexts.

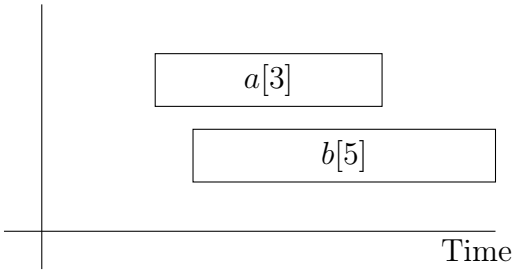


Figure 5: Temporal plan with concurrent actions a and b, action a has duration 3 while action b has duration 5.

A temporal planning problem can be viewed as a concurrent planning problem where the set of atomic actions is $B = \{\mathsf{start}(a_1), \mathsf{end}(a_1), \mathsf{start}(a_2), \mathsf{end}(a_2), \ldots\}$. Specifically, a temporal plan induces a concurrent plan over the actions in B.

Since temporal actions have durations and can overlap, in general we cannot define a *temporal plan* as a simple sequence of actions. Instead, a plan for Π is a set of action-time pairs $\pi = \langle (a_1, t_1), \ldots, (a_k, t_k) \rangle$. 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. We say that π has *concurrent actions* if there exist two pairs (a_i, t_i) and (a_j, t_j) in π such that $t_i < t_j < t_i + d(a_i)$, i.e. a_j starts after a_i starts but before a_i ends. Figure 5 shows an example of a temporal plan with two concurrent actions a and b, action a has duration 3 while action b has duration 5.

Each action-time pair (a, t) induces two events: start_a, with associated time t, and end_a, with associated time $t + d(a)$. If we order the $2k$ events induced from $\pi = \langle (a_1, t_1), \ldots, (a_k, t_k) \rangle$ by their associated times, we obtain an *event sequence* $\pi_E = \langle E_1, \ldots, E_m \rangle$ satisfying $1 \leq m \leq 2k$. Each E_j, $1 \leq j \leq m$, is a *joint event* composed of one or more individual events of π that all have the same associated time. Note that joint events can be seen as the joint actions we defined for the concurrent planning formalism (see Section 2.2) since, as explained before, events represent atomic actions.

For each j, $1 < j \leq m$, the individual events of a joint event E_{j-1} are scheduled to occur *before* the events of E_j. We say that π has *simultaneous events* if $m < 2k$, i.e. if at least one joint event is composed of multiple individual events. By imposing

the restriction $m = 2k$, we disallow simultaneous events, i.e. no pair of events of π can have the same associated time. Figure 6 shows an example temporal plan with simultaneous events end_a and end_b. Note that in this case actions a and b are also concurrent.

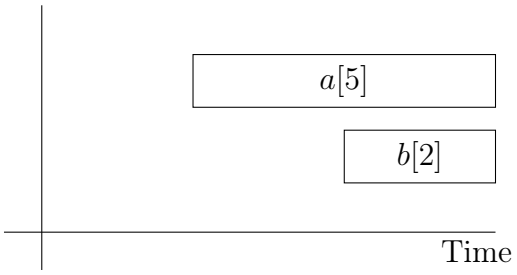


Figure 6: Temporal plan with simultaneous events end_a and end_b.

When multiple events occur simultaneously, it is necessary to verify that the resulting joint event is *valid*, i.e. that individual events do not interact in undesired ways. We adopt the definition of valid joint events from PDDL 2.1 [19]:

Definition 2.1. *Let $E = \{e^1, \ldots, e^k\}$ be a joint event, i.e. a set of simultaneous events. E is* valid *if and only if there exists no event $e \in E$ with an effect on a fluent that is mentioned by another event in E.*

As a consequence of Definition 2.1, given an individual event e, no effect of e can be mentioned by another event simultaneous with e, not even as a precondition. The only way that a fluent f can be mentioned by two simultaneous events e and e' is if f is a precondition of both e and e'. Although it is a rather strong limitation, this definition is commonly accepted in temporal planning and, crucially, is implemented as part of VAL [20], the program used at the IPC (and by us) to validate temporal plans.

The above conditions do not apply to contexts, which only have to hold from immediately after the start of a temporal action until immediately before its end. If the start of a temporal action a is part of a joint event E, it is safe for another event in E to add a context of a. Likewise, if the end of a is part of a joint event E, it is safe for another event in E to delete a context of a.

We are now able to define when a temporal plan π solves a temporal planning instance $\Pi = \langle F, A, I, G \rangle$. Let $\pi_E = \langle E_1, \ldots, E_m \rangle$ be the sequence of joint events induced by π, and let $E = \{e^1, \ldots, e^k\}$ be a joint event in π_E. If E is invalid, then so is π. In addition, π_E has to respect the contexts of temporal actions in π. Specifically, for each $(a, t) \in \pi$, let E_i be the joint event that includes start_a and let E_j be the joint event that includes end_a. The context $\mathsf{pre}_o(a)$ of a has to hold in each intermediate state s_k, $i \leq k < j$.

The quality of a temporal plan is given by its makespan, i.e. the temporal duration from the the start of the first action execution to the end of the last action execution. Formally, the makespan of a temporal plan π is defined as $\max_{(a,t)\in\pi}(t + d(a))$. The first action is assumed to start a time 0, i.e. $\min_{(a,t)\in\pi} t = 0$.

One of the most common forms of required concurrency is given by *single hard envelopes* [15]. A single hard envelope is a temporal action a that adds a fluent $f \in F, f \notin I$ at start and deletes it at end, i.e. $f \in \mathsf{cond}_s(a), \neg f \in \mathsf{cond}_e(a)$. Besides, there has to exist another action b with shorter duration than a that has f as context, i.e. $d(b) < d(a)$ and $f \in \mathsf{pre}_o(b)$.

Chapter 3

Compilation of Multiagent Planning to Classical Planning

In this chapter, we describe our approach for transforming a multiagent planning problem (MAP) $\Pi = \langle N, F, \{A^i\}_{i=1}^n, I, G\rangle$ into a classical planning problem $\Pi' = \langle F', A', I', G'\rangle$.

As described in Section 2.3.1, we can naively convert a MAP into a classical planning problem making A' equal to all valid concurrent actions of the MAP, i.e. $A' \subseteq A^1 \times \cdots \times A^n$. Therefore, the *purpose* of our work is not so much to compile the problem into classical planning. Instead, our aim is to transform the MAP into an alternative classical planning problem whose action set A' is much smaller than $A^1 \times \cdots \times A^n$.

3.1 Description

Our compilation assumes that if a concurrent action $a = (a^1, \ldots, a^k)$ satisfies all concurrency constraints, then a is well-formed, i.e. does not introduce conflicting effects. Moreover, we also make the following assumptions:

1. The input MAP is specified using Kovacs (2012) notation (see Section 2.3.2).

2. Each agent performs at most once in a joint action, i.e. an agent cannot perform two actions simultaneously.

   Note that this assumption determines the kind of solutions that the classical planner will provide. Nevertheless, if we want an agent to do two actions simultaneously, we can derive two subagents, each of them doing one of the actions.

In any planning problem, each step is divided in two phases: (1) the selection of an action, and (2) the application of the selected action. Any planner must be able to do these steps in the resulting classical planning problem. However, we have to take into account that we are not selecting and applying atomic actions, instead we are selecting and applying joint actions.

The compilation introduces mechanisms (fluents and actions) that allow to select and apply joint actions while respecting concurrency constraints. Besides, given the addition of new fluents, a third step is required to clean these fluents after the selection and execution of each joint action. The details of each step are described below and illustrated in Figure 7:

1. Each agent optionally looks for an atomic action to select (i.e. it is not required that all agents choose an action). The preconditions of the atomic actions must be satisfied in order to be selected. At this point, the negative concurrency constraints must also be respected.

   The result of this selection is a joint action $a = \left(a^1, \ldots, a^k\right)$ corresponding to a subset of agents.

2. The effects of the atomic actions in a are applied. At this point, if an agent has selected an action b that has positive concurrency with c, then this agent can check whether c is in a. On the other hand, it is also possible to check whether actions used in conditional effects are in a.

3. The actions in a reset auxiliary fluents (later introduced) so that another joint action can be started.

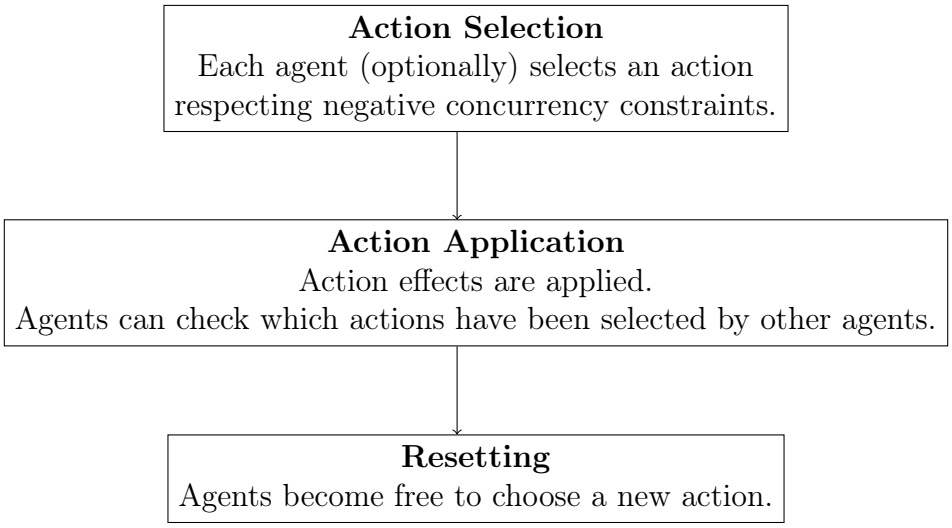


Figure 7: Process of selection and application of a joint action.

In the next sections we describe the new fluents and actions of the classical problem Π'[1]. Moreover, we also prove that the compilation is sound and complete.

3.1.1 Fluents

We describe the fluents in PDDL format, i.e. each fluent is associated with a predicate.

The set of fluents F' includes all fluents in F. Besides, F' also includes different fluents corresponding to the previously described steps/phases:

- A fluent free indicating that a joint action can be started, i.e. we are not currently composing a joint action.
- A fluent selecting indicating that each agent is (optionally) selecting an atomic action to do (i.e. the joint action is being formed).
- A fluent applying indicating that those agents which selected actions are applying them.
- A fluent resetting indicating that actions have been applied. Some fluents are being cleaned up to make another concurrent action possible.

[1]The code of the compilation is available in the following repository: `https://github.com/aig-upf/universal-pddl-parser-multiagent`.

There are also fluents that model the state of an agent $i \in N$:

- A fluent $\mathsf{free\text{-}agent}(i)$ indicating that agent i is free to select an atomic action.
- A fluent $\mathsf{busy\text{-}agent}(i)$ indicating that agent i has selected an atomic action to do.
- A fluent $\mathsf{done\text{-}agent}(i)$ indicating that agent i has applied its selected atomic action.

Finally, for each action $a^i \in A^i, i \in N$, the following fluents are added:

- A fluent $\mathsf{active}\text{-}a^i$ indicating that the atomic action a^i has been selected. These fluents are helpful to:
  1. Implicitly represent the joint action a (all those actions set to active form the joint action).
  2. Avoid selecting an atomic action b^j that has negative concurrency with a^i.
  3. Check if an atomic action b^j that has positive concurrency with a^i has been selected.
  4. Check if an atomic action b^j in a conditional effect of a^i has been selected.
- A fluent $\mathsf{req\text{-}neg}\text{-}a^i$ indicating that the atomic action a^i cannot be selected. This kind of fluents complement $\mathsf{active}\text{-}a^i$ fluents to help satisfy negative concurrency constraints.

Therefore, the resulting set of fluents F' can be expressed as follows:

$$\begin{aligned} F' = F \cup & \left\{\mathsf{active}\text{-}a^i, \mathsf{req\text{-}neg}\text{-}a^i : a^i \in A^i, i \in N\right\} \\ & \cup \left\{\mathsf{free\text{-}agent}(i), \mathsf{busy\text{-}agent}(i), \mathsf{done\text{-}agent}(i) : i \in N\right\} \\ & \cup \left\{\mathsf{free}, \mathsf{selecting}, \mathsf{applying}, \mathsf{resetting}\right\}. \end{aligned} \tag{3.1}$$

Lemma 3.1. *The resulting number of fluents in the single-agent planning problem* Π' *is*

$$|F'| = |F| + 2\sum_{i \in N} \left|A^i\right| + 3n + 4.$$

Proof. By equation 3.1, the set of fluents F' in the single-agent problem Π' contains all the fluents F in the MAP Π. Besides, we add two new fluents active-a^i and req-neg-a^i for each agent action $a^i \in A^i, i \in N$. Thus, the size of F' is increased by $2\sum_{i \in N} |A^i|$. Then, three more fluents (free-agent(i), busy-agent(i), done-agent(i)) are added for each agent $i \in N$, so the size is increased by $3n$. Finally, four more fluents are introduced: free, selecting, applying and resetting. □

Note that Lemma 3.1 states which is the number of fluents taking into account the assumption presented at the beginning of the section: each agent performs at most once in a joint action. We can overcome this problem if for each agent $i \in N$ we derive $|A^i|$ subagents (i.e. we get as many subagents as actions the agent can do). In the worst case, we will have to add $\max_i \left|A^i\right|, i \in N$ agents; that is, we want all agents to be capable of doing all their actions at the same time. Therefore, the size of F' can be bounded by:

$$\begin{aligned} |F'| &= |F| + 2nm + 3nm + 4 \\ &= |F| + 5nm + 4, \\ m &= \max_i \left|A^i\right|, i \in N. \end{aligned}$$

Take into account that the maximum number of actions that can be taken by an agent (m) will typically be much higher than the number of agents (n). Thus, we can say that $|F'|$ is bounded by the maximum number of actions an agent can do and the size of the original set of fluents F. Note that this number of fluents is still polynomial, as the one in Lemma 3.1.

The initial state I' is defined as

$$I' = I \cup \{\mathsf{free}\} \cup \{\mathsf{free\text{-}agent}(i) : 1 \leq i \leq n\}.$$

That is, we are initially free to start a concurrent action, and all agents are free to select an action. On the other hand, the goal condition is defined as $G' = G \cup \{\mathsf{free}\}$.

3.1.2 Actions

The first kind of actions in the new action set A' are those which allow us to switch from one phase to another during the execution of a joint action. They are defined as follows:

start: $\text{pre} = \{\mathsf{free}\}$,
$\text{cond} = \{\emptyset \rhd \{\neg\mathsf{free}, \mathsf{selecting}\}\}$.

apply: $\text{pre} = \{\mathsf{selecting}\}$,
$\text{cond} = \{\emptyset \rhd \{\neg\mathsf{selecting}, \mathsf{applying}\}\}$.

reset: $\text{pre} = \{\mathsf{applying}\}$,
$\text{cond} = \{\emptyset \rhd \{\neg\mathsf{applying}, \mathsf{resetting}\}\}$.

finish: $\text{pre} = \{\mathsf{resetting}, \mathsf{free\text{-}agent}(i) : 1 \leq i \leq n\}$,
$\text{cond} = \{\emptyset \rhd \{\neg\mathsf{resetting}, \mathsf{free}\}\}$.

For each agent action $a^i \in A^i, i \in N$, we create three new actions: select-a^i, do-a^i and end-a^i. These actions represent each of the three steps that an agent must perform during each of the three phases.

The precondition of a select-a^i action checks whether a^i is selectable by an agent that has not chosen any action, i.e. it respects the original preconditions of a^i and

its associated negative concurrency constraints. It is specified as follows:

$$\begin{aligned}
\text{pre} &= \{\textsf{selecting}, \textsf{free-agent}(i), \neg\textsf{req-neg-}a^i\} \\
&\cup \{l, \neg l : l, \neg l \in \textsf{pre}(a^i), l, \neg l \in L(F)\} \\
&\cup \{\neg\textsf{active-}b^j : \neg b^j \in \textsf{pre}(a^i), b^j \in A^j\}, \\
\text{cond} &= \{\emptyset \rhd \{\textsf{busy-agent}(i), \neg\textsf{free-agent}(i), \textsf{active-}a^i\}\} \\
&\cup \{\emptyset \rhd \{\textsf{req-neg-}b^j : \neg b^j \in \textsf{pre}(a^i), b^j \in A^j\}\}.
\end{aligned}$$

Figure 8 shows the compilation from the original lift-side action described in Figure 1 to a select-a^i action.

Lemma 3.2. *The selected a^i actions (i.e. those actions for which* select-a^i *is done) satisfy all negative concurrency preconditions.*

Proof. Because of the precondition $\neg$active-b^j, a^i is not selected if there is an action b^j with which it has negative concurrency. On the other hand, by adding req-neg-b^j we prevent b^j from being selected later. □

The do-a^i actions are used for applying the effects of a selected action a^i; therefore, they are the only ones that change the values of fluents in F (the set of fluents of the MAP). At this point, the set of selected agent actions is fixed, each represented by a fluent of type active. Thus, we can use the active fluents to verify (1) the positive concurrency constraints to apply the action, and (2) whether some conditional effects should also be applied. Their specification is the following:

$$\begin{aligned}
\text{pre} &= \{\textsf{applying}, \textsf{busy-agent}(i), \textsf{active-}a^i\} \\
&\cup \{\textsf{active-}b^j : b^j \in \textsf{pre}(a^i), b^j \in A^j\}, \\
\text{cond} &= \{\emptyset \rhd \{\textsf{done-agent}(i), \neg\textsf{busy-agent}(i)\}\} \\
&\cup \{C' \rhd E : C \rhd E \in \textsf{cond}(a^i)\},
\end{aligned}$$

where C' is the set of literals C with all $b^j \in A^j$ replaced by active-b^j if $b^j \in C$, and $\neg$active-b^j if $\neg b^j \in C$.

```
(:action select-lift-side
    :parameters (?a - agent ?s - side)
    :precondition (and
                      (selecting)
                      (free-agent ?a)
                      (not (req-neg-lift-side ?a ?s))
                      (down ?s)
                      (at-side ?a ?s)
                      (handempty ?a)
                      (forall
                             (?a2 - agent ?s2 - side)
                             (not (active-lower-side ?a2 ?s2))
                      )
                  )
    :effect (and
                (not (free-agent ?a))
                (busy-agent ?a)
                (active-lift-side ?a ?s)
                (forall
                       (?a2 - agent ?s2 - side)
                       (req-neg-lower-side ?a2 ?s2)
                )
            )
)
```

Figure 8: Compilation of TableMover's lift-side into a classical planning select-a^i action.

The resulting compilation from the original lift-side action to a do-a^i one is shown in Figure 9. In this case, observe that the active fluent is being used to determine if the conditional effect should be applied: if any other agent is lifting the other side of the table, the blocks fall to the ground.

The actions end-a^i are needed to reset auxiliary fluents to their original values. They can be understood as cleanup actions. They have the following preconditions and conditional effects:

$$\begin{aligned} \text{pre} &= \{\text{resetting}, \text{done-agent}(i), \text{active-}a^i\}, \\ \text{cond} &= \{\emptyset \rhd \{\text{free-agent}(i), \neg\text{done-agent}(i), \neg\text{active-}a^i\}\} \\ &\cup \{\emptyset \rhd \{\neg\text{req-neg-}b^j : \neg b^j \in \text{pre}(a^i), b^j \in A^j\}\}. \end{aligned}$$

By applying this action, the agent i becomes free again and the action a^i is not longer active. Furthermore, those actions b^j that were marked as incompatible in select-a^i become selectable again (i.e. the req-neg-b^j is reset to false). Figure 10 shows the resulting compilation of the lift-side action into an end-a^i action.

The resulting action set of the single-agent planning problem Π' is

$$\begin{aligned} A' = &\left\{\text{select-}a^i, \text{do-}a^i, \text{end-}a^i : a^i \in A^i, i \in N\right\} \\ &\cup \{\text{start}, \text{apply}, \text{reset}, \text{finish}\}. \end{aligned} \tag{3.2}$$

Lemma 3.3. *The resulting number of actions of the single-agent planning problem* Π' *is*

$$|A'| = 3\sum_{i \in N} \left|A^i\right| + 4.$$

Proof. By equation 3.2, we know that the action set A' in the single-agent problem Π' contains four parameter-free actions: start, apply, reset, finish. Thus, A' has base size of 4. Then, we add three actions (select-a^i, do-a^i, end-a^i) for each agent action $a^i \in A^i, i \in N$. For this reason, the size of A' is increased by $3\sum_{i \in N} |A^i|$. □

```
(:action do-lift-side
    :parameters (?a - agent ?s - side)
    :precondition (and
                      (applying)
                      (busy-agent ?a)
                      (active-lift-side ?a ?s)
                  )
    :effect (and
                (not (busy-agent ?a))
                (done-agent ?a)
                (not (down ?s))
                (up ?s)
                (lifting ?a ?s)
                (not (handempty ?a))
                (forall
                       (?b - block ?r - room ?s2 - side)
                       (when
                            (and
                                 (inroom Table ?r)
                                 (on-table ?b)
                                 (down ?s2)
                                 (forall
                                        (?a2 - agent)
                                        (not (active-lift-side ?a2 ?s2))
                                 )
                            )
                            (and
                                 (on-floor ?b)
                                 (inroom ?b ?r)
                                 (not (on-table ?b))
                            )
                       )
                )
           )
)
```

Figure 9: Compilation of TableMover's lift-side into a classical planning do-a^i action.

```
(:action end-lift-side
    :parameters (?a - agent ?s - side)
    :precondition (and
                      (resetting)
                      (done-agent ?a)
                      (active-lift-side ?a ?s)
                  )
    :effect (and
                (not (done-agent ?a))
                (free-agent ?a)
                (not (active-lift-side ?a ?s))
                (forall
                       (?a2 - agent ?s2 - side)
                       (not (req-neg-lower-side ?a2 ?s2))
                )
            )
)
```

Figure 10: Compilation of TableMover's lift-side into a classical planning end-a^i action.

Note that the number of actions $|A'|$ grows linearly in the description of the MAP. In contrast, in the worst case, the number of concurrent actions grows exponentially in the number n of agents:

$$\prod_{i \in N} |A^i| = O(\mathcal{A}^n),$$

where $\mathcal{A} = \max_i |A^i|$ is the size of the largest action set.

Finally, we provide some figures to show and exemplify which is the flow of actions in a plan. Figure 11 shows which is the order in which single-agent actions are selected to form a joint action. Joint actions begin with a start action and end with a finish action. When the finish action ends, a new joint action can be started. The select actions are executed t times, where t is the number of agents that have applied an action of this kind. Those same t agents will also do a do and an end action.

To give an actual plan example, we use a simple instance of the TableMover domain. The following lines show a possible plan for the example in Figure 12. In this problem, block b_1 must be moved from room r_1 to room r_2.

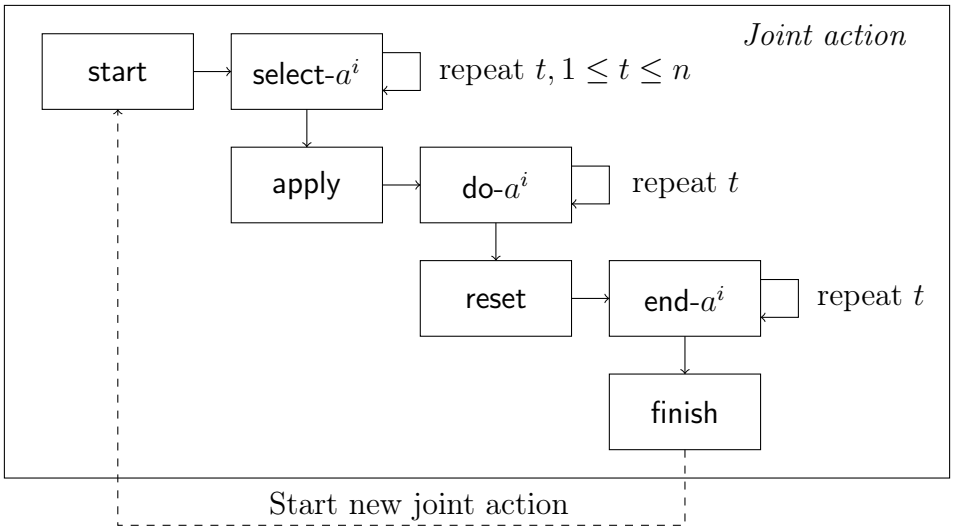


Figure 11: Actions in the single-agent problem that represent the selection of a joint action of the MAP.

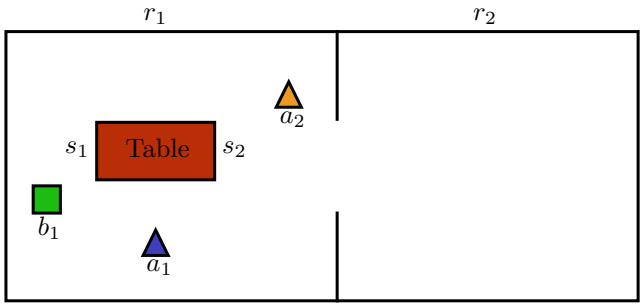


Figure 12: Initial state of a simple TABLEMOVER instance.

```
(start )
(select-to-table a1 r1 s2)
(select-pickup-floor a2 b1 r1)
(apply )
(do-to-table a1 r1 s2)
(do-pickup-floor a2 b1 r1)
(reset )
(end-to-table a1 r1 s2)
(end-pickup-floor a2 b1 r1)
(finish )
(start )
(select-putdown-table a2 b1 r1)
(apply )
(do-putdown-table a2 b1 r1)
(reset )
(end-putdown-table a2 b1 r1)
(finish )
(start )
(select-to-table a2 r1 s1)
(apply )
(do-to-table a2 r1 s1)
(reset )
(end-to-table a2 r1 s1)
(finish )
(start )
(select-lift-side a1 s2)
(select-lift-side a2 s1)
(apply )
(do-lift-side a1 s2)
(do-lift-side a2 s1)
(reset )
(end-lift-side a1 s2)
(end-lift-side a2 s1)
(finish )
(start )
(select-move-table a1 r1 r2 s2)
(select-move-table a2 r1 r2 s1)
(apply )
(do-move-table a2 r1 r2 s1)
(do-move-table a1 r1 r2 s2)
(reset )
(end-move-table a1 r1 r2 s2)
```

```
43 (end-move-table a2 r1 r2 s1)
44 (finish )
45 (start )
46 (select-lower-side a1 s2)
47 (apply )
48 (do-lower-side a1 s2)
49 (reset )
50 (end-lower-side a1 s2)
51 (finish )
```

As each joint action begins with an start action and ends with a finish action, this plan contains 6 joint actions. Each of the joint actions goes through the three phases previously explained: action selection, action application and resetting.

The actions being selected are those that compound the actual joint action in the MAP. Therefore, if we transform the single-agent plan into a multiagent plan, there would be the following actions:

```
1 (to-table a1 r1 s2), (pickup-floor a2 b1 r1)
2 (putdown-table a2 b1 r1)
3 (to-table a2 r1 s1)
4 (lift-side a1 s2), (lift-side a2 s1)
5 (move-table a1 r1 r2 s2), (move-table a2 r1 r2 s1)
6 (lower-side a1 s2)
```

3.1.3 Properties

We show that the compilation is both sound and complete.

Lemma 3.4. *When applying action* start*, fluent* free-agent(i) *is true for each agent* $i \in N$.

Proof. The start action can only be done if the fluent free is true. This happens in two situations:

1. In the initial state I, where the fluent free-agent(i) is also true for each agent $i \in N$. Then, these fluents are true when applying the start action.

2. After doing a finish action. This action requires free-agent(i) fluents to be true for $i \in N$. Then, these fluents are still true when applying the start action.

□

Lemma 3.5. *When applying action* apply*, fluent* busy-agent$(i), i \in N$ *is true for a subset of agents, and in this case,* active-a^i *is true for some action* $a^i \in A^i$*.*

Proof. The only precondition of the apply action is that the selecting fluent is true. While the selecting fluent is true, we can only apply actions of type select-a^i or the apply action to change phase. If a select-a^i action is applied, the fluents busy-agent(i) and active-a^i corresponding to agent i and the original action $a^i \in A^i$ become true. At most N select-a^i actions can be applied (it depends on the negative concurrency constraints and the original preconditions of the action). Therefore, before doing the apply action, we will have a set of agents with busy-agent(i) and active-a^i set to true for their associated action. □

Lemma 3.6. *When applying action* reset*, fluent* done-agent(i) *is true for the same subset of agents, and* active-a^i *is still true if positive concurrency constraints hold.*

Proof. For an agent $i \in N$ to have fluent done-agent(i), it is required it applies action do-a^i for the action $a^i \in A^i$ that was previously selected (i.e. select-a^i action). Moreover, to apply a do-a^i action, the fluent busy-agent(i) has to be true. Therefore, only those agents whose fluent busy-agent is true will be able to apply the do-a action and make their corresponding done-agent fluent true. This will only happen if positive concurrency constraints (if any) hold.

Although the reset action does not indicate it explicitly, all agents with busy-agent fluent set to true will do a do-a action (again, if positive concurrency constraints hold). To explain the reason, we have to go through the whole cycle for choosing a joint action. To end a joint action, the fluent free-agent must be true for all agents (precondition of the finish action). To make free-agent true, all those agents having busy-agent true have to end the whole cycle, i.e. first become done-agent and then free-agent. This forces them to perform the do-a action and, thus, the done-agent fluent will be true for all agents that were busy-agent before applying the reset action.

On the other hand, the fluents active-a are not changed by actions do-a (they only use them as preconditions). Thus, their value for actions chosen by agents being

busy-agent are that moment is still true. □

Lemma 3.7. *When applying action* finish*, fluent* free-agent(i) *is again true for each agent, and all* active-a^i *fluents become false.*

Proof. In Lemma 3.6, we showed that before doing a reset action, the set of agents that are performing the joint action have their done-agent fluent set to true. The finish action requires that all agents have their corresponding free-agent fluent set to true. The agents become free by applying the reset-a^i actions, where a^i is the atomic action agent $i \in N$ chose in the selection phase. The corresponding active-a^i fluents become false as an effect of reset-a^i actions. □

Theorem 3.8 (Soundness)**.** *A sequential plan π' that solves Π' can be transformed into a concurrent plan π that solves Π.*

Proof. The start action represents the beginning of a joint action. In Lemma 3.4, we showed that any agent is free to start a joint action. All agents doing an select-a^i action indicate that they participate in the joint action using atomic action a^i. Because of Lemma 3.5, in the moment of doing the apply action, a subset of agents will have chosen atomic actions to do (one each); besides, all selected atomic actions satisfy all negative concurrency constraints, as Lemma 3.2 states.

In Lemma 3.6, we demonstrated that at the end of the application phase (if positive concurrency constraints are respected), each of the agents have applied the effects of their chosen atomic actions a^i by using the corresponding do-a^i actions. Hence, the joint action satisfies all concurrency constraints (positive and negative). By assumption, all such joint actions are well-formed, i.e. do not introduce conflicting effects.

Finally, because of Lemma 3.7, when the joint action finishes, all auxiliary fluents reset to its previous state for either (1) do a new joint action, or (2) end the process. Since π' solves Π', the sequence π of concurrent actions induced by the plan π' achieves the goal condition $G' = G \cup \{\text{free}\}$, implying that π solves Π.

□

Theorem 3.9 (Completeness)**.** *A concurrent plan π that solves Π can be transformed into a classical plan π' that solves Π'.*

Proof. For each concurrent action a_j of the plan π, we form a sequence of actions of Π' that emulates a_j. An start action is used to represent the beginning of a concurrent action in Π'. After doing this action, we are in the select phase, where the atomic actions forming the concurrent actions are chosen. Thus, for each of the constituent actions a^i of the concurrent action a_j, we do select-a^i in π'. Since by definition a_j is well-formed, the order in which we do select-a^i does not matter: all the original preconditions in Π and the negative concurrency constraints will be satisfied.

Once all the required select-a^i actions have been done, the apply action is performed for changing from the selection to the application phase. Analogously to the selection phase, a do-a^i action is applied for each atomic action a^i in a_j. Again, since a_j is well-defined, the order of the do-a^i actions does not matter: the positive concurrency constraints will be satisfied. Since the effect of a_j equals the union of the individual effects, applying all do actions results in the same new state on fluents in F.

After doing the do-a^i actions, the reset action is applied for changing from the application to the resetting phase. In this phase, an end-a^i action is applied for each atomic action a^i in joint action a_j. These actions do not modify any of the original fluents in F; therefore, the state on fluents in F after doing the end-a^i actions is the same than the one we had before starting the reset phase (i.e. only some of the auxiliary fluents in Π' have been modified).

The emulated joint action ends with the finish action, which makes the emulation of another joint action possible. Since π achieves the goal G on F, π' achieves the goal $G' = G \cup \{\text{free}\}$ since the fluent free is true following the last concurrent action, which is a finish action. □

In the following lines, we want to emphasize the utility of this approach. Although the search space is the same, the number of actions has been reduced from expo-

nential to linear. This relies on having a classical planner that exploits heuristics to avoid having to search the entire search space of solutions (planners have a small representation of the planning problem and can often solve such problems even when the search space is exponential).

3.2 Results

Experiments have been performed in three domains: TableMover, Maze and Workshop. To solve single-agent problems we used the classical planner Fast Downward [21] in the LAMA setting [22], with a timeout of 3,600 seconds.

The domains and their respective results are described in the following subsections. The results are summarized in one table per domain. For each problem size we generated 5 random instances and we report three parameters:

- n_a: the average number of actions of the single-agent plan π'.
- n_{se}: the average number of select actions of the plan π', which indicates how many individual actions are applied.
- n_{st}: the average number of start actions of the plan π', which indicates the number of concurrent actions.

The cells containing a dash represent cases for which a minimum number of instances has not been solved after one hour. Thus number varies between domains: in TableMover and Workshop we required all 5 instances to be solved, while we required 3 instances to be solved in Maze.

3.2.1 TableMover Domain

The TableMover domain, described in Section 2.3.2, is used with some modifications: there is more than one table and more than two agents. To promote concurrent plans, agents can only move between rooms if they are carrying a table.

Table 1: Results for the TABLEMOVER domain (a = agents, r = rooms, b = blocks).

		$a=2$			$a=4$			$a=8$		
		n_a	n_{se}	n_{st}	n_a	n_{se}	n_{st}	n_a	n_{se}	n_{st}
$r=2$	$b=1$	41	7	5	52	9	6	64	11	8
	$b=2$	78	14	9	112	19	14	-	-	-
	$b=4$	97	18	11	211	37	25	-	-	-
	$b=8$	110	21	12	-	-	-	-	-	-
$r=8$	$b=1$	93	17	11	172	31	20	-	-	-
	$b=2$	168	30	19	-	-	-	-	-	-
	$b=4$	212	39	24	-	-	-	-	-	-
	$b=8$	1106	196	129	-	-	-	-	-	-
$r=20$	$b=1$	190	36	21	179	33	20	-	-	-
	$b=2$	172	33	18	-	-	-	-	-	-
	$b=4$	834	155	93	-	-	-	-	-	-
	$b=8$	1067	196	120	-	-	-	-	-	-

We have generated 5 instances for different combinations of agents (2, 4, 8), rooms (2, 8, 20) and blocks (1, 2, 4, 8). The instances are characterized by:

- Rooms are linked such that they form a tree structure.
- Blocks and tables are randomly placed in rooms.
- For each table, two agents start in the same room.
- All blocks must be moved to the same random room.

Table 1 shows the results for TABLEMOVER. The minimum number of instances to be solved is 5 (i.e. all of them).

We observe that the higher the parameter values are, the longer plans become. There are many cases that cannot be solved in the given time, concretely those with many agents. Even in simple cases with 2 rooms, 2 blocks and 8 agents, the planner does not generate any solution.

In large instances, agents tend to prefer moving few blocks using the table. In some instances, the agents may leave a room without collecting the blocks there. This strategy is inefficient since they eventually have to return to that room.

Table 2: Results for the MAZE domain (a = agents, n = rows and columns of the $n \times n$ grid).

	$a=5$			$a=10$			$a=15$			$a=20$		
	n_a	n_{se}	n_{st}	n_a	n_{se}	n_{st}	n_a	n_{se}	n_{st}	n_a	n_{se}	n_{st}
$n=4$	81	18	7	117	28	8	165	43	9	238	56	18
$n=8$	163	33	16	308	70	25	-	-	-	-	-	-
$n=12$	-	-	-	-	-	-	-	-	-	-	-	-
$n=16$	456	83	52	510	124	34	-	-	-	-	-	-
$n=20$	475	88	53	-	-	-	-	-	-	-	-	-

We experimented with a cost function that made the start action more expensive. The objective was to minimize the number of start actions, thus increasing the number of blocks the agents would move simultaneously on the table. However, it barely had any effect, no matter the amplitude of the cost.

3.2.2 MAZE Domain

In the case of the MAZE domain (described in Section 2.3.2), we randomly generated 5 instances for grids of size 4x4, 8x8, 12x12, 16x16 and 20x20, and for 5, 10, 15 and 20 agents.

Table 2 shows the results for this domain. The minimum number of instances to be solved is 3 since it was more difficult to obtain solved instances than in TABLEMOVER. Part of the explanation may be that we did not explicitly test whether a given randomly generated instance was solvable; moving between rooms sometimes introduces intricate interactions, since agents have to collaborate by pushing switches that unlocks certain doors, or rowing a boat together. As in TABLEMOVER, longer plans are generated for larger parameter values. Again, the number of agents seems to be the main bottleneck that prevents solving more complex problems.

The number of solved cases is less than that reported in the experiments by Crosby *et al.* (2014) using their MAP transformation algorithm [11]. Table 3 reports these results, where L' is the average length of the single-agent plan, and L is the average length of the compressed plan (i.e. number of joint actions). We conjecture that when applicable, their transformation is slightly more efficient; this could be attributed to the fact that our concurrency constraints are more expressive, and hence the

Table 3: Results for the MAZE domain using the Crosby *et al.* (2014) multiagent to single-agent compilation.

	$a = 5$		$a = 10$		$a = 15$		$a = 20$	
	L'	L	L'	L	L'	L	L'	L
$n = 4$	8	5	28	16	39	24	43	21
$n = 8$	29	24	52	41	56	39	88	60
$n = 12$	38	31	43	31	-	-	97	49
$n = 16$	59	50	56	38	153	133	127	98
$n = 20$	55	47	-	-	81	63	132	104

resulting problems may be more difficult to solve.

3.2.3 WORKSHOP Domain

We introduce a new domain called WORKSHOP, in which the objective is to do inventory in a high-security storage facility. The actions and their constraints are the following:

- To open a door, one agent has to press a switch while another agent simultaneously turns a key.
- To do inventory on a pallet, one agent has to use a forklift to lift the pallet while another agent examines it (for security reasons, labels are located underneath pallets).
- There are also actions for picking up a key, entering or exiting a forklift, moving an agent, and driving a forklift.

The instances we generated have the following features:

- There are connected sections forming a tree. Sections are linked through locked security doors.
- Each section has a number of subsections (i.e. rooms) which are connected via unlocked doors, forming a tree.
- In each section there is a switch and a key for each of the security doors it is connected to. The key and the switch are randomly placed in one of its rooms.

- Pallets and forklifts are randomly placed in rooms.
- Two agents are placed in the same room as a forklift.

We generated 5 random instances for certain combinations of agents (2, 4, 8), pallets (1, 4, 8) and rooms (2, 16, 32, 64). The number of rooms represents the product between the number of sections and the number of subsections.

Table 4 shows the experiments for this domain. The minimum number of instances to be solved is 5. Almost all cases have been solved for this domain. Only those with a large number of parameters are not solved. However, note that unlike the number of rooms and the number of pallets, increasing the number of agents results in shorter plans. This happens because it is more likely that a pallet is near to two agents when there are more agents; therefore, the agents do not run many actions for moving or driving a forklift between rooms. For example, if we look at the average number of move and drive actions with $r = 64$ and $p = 1$, we get the following:

- When $a = 2$, the agents move 33.6 times on average.
- When $a = 4$, the agents move 21.6 times on average.
- When $a = 8$, the agents move 15.8 times on average.

Thus, it is reasonable to think that the fact that smaller plans are obtained is due to shorter distances between the agents and the pallets.

Table 4: Results for the WORKSHOP domain (a = agents, r = rooms, p = pallets).

		$a=2$			$a=4$			$a=8$		
		n_a	n_{se}	n_{st}	n_a	n_{se}	n_{st}	n_a	n_{se}	n_{st}
$r=2$	$p=1$	24	4	3	20	3	2	17	3	2
	$p=4$	63	12	6	50	11	4	47	10	4
	$p=8$	100	20	10	82	19	6	91	18	9
$r=16$	$p=1$	99	18	11	81	16	9	50	9	6
	$p=4$	242	46	26	206	40	21	155	30	16
	$p=8$	364	68	40	287	58	28	230	42	26
$r=32$	$p=1$	160	30	18	108	20	12	58	11	6
	$p=4$	389	72	44	312	59	34	169	30	19
	$p=8$	517	95	58	515	94	58	-	-	-
$r=64$	$p=1$	262	48	29	168	29	20	124	23	14
	$p=4$	601	108	69	-	-	-	-	-	-
	$p=8$	946	172	107	-	-	-	-	-	-

Chapter 4

STP: Simultaneous Temporal Planner

In this chapter, we introduce a temporal planner that is capable to handle simultaneous events, i.e. events that occur at exactly the same time. Firstly, we justify the need for this kind of planner by introducing the AIA domain, which uses simultaneous events. Then, we explain how the planner works. Finally, we show the results of some experiments using this planner and compare them to other state-of-the-art planners.

4.1 The AIA Domain

Allen's interval algebra [23] is a calculus for temporal reasoning in logic that defines possible relations between time intervals. Specifically, there are seven possible relations on interval pairs (X, Y), illustrated in Figure 13. The first six relations also have an inverse, for a total of 13 unidirectional relations.

In this section we describe the AIA domain, a novel domain for temporal planning based on Allen's interval algebra. Specifically, the domain was designed with two goals in mind: 1) incorporate diverse forms of required concurrency, not only in the form of single hard envelopes (see Section 2.4); and 2) include temporal planning

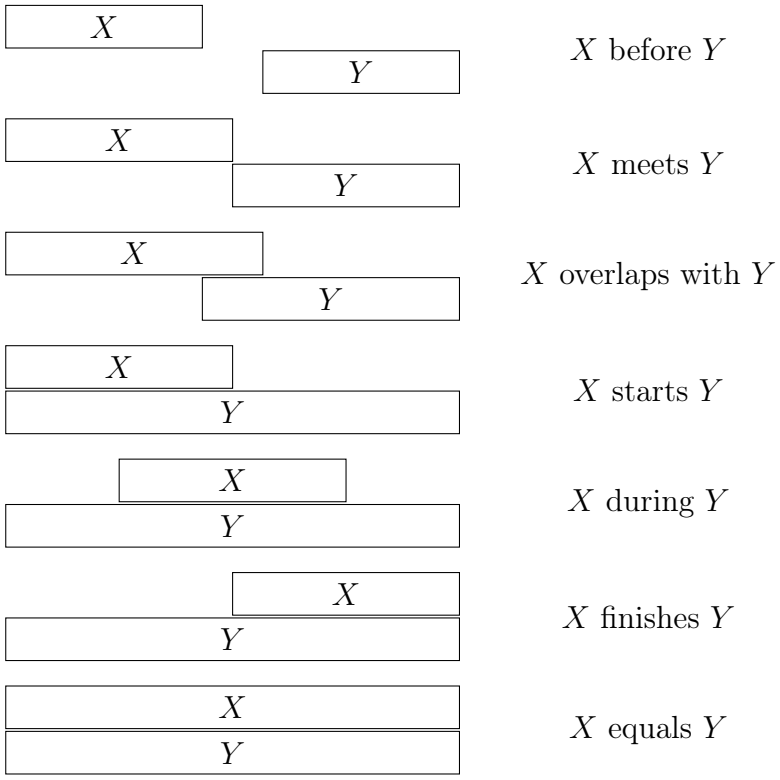


Figure 13: The seven relations on interval pairs (X, Y) in Allen's interval algebra.

instances that require simultaneous events.

To model Allen's interval algebra in PDDL, we define a single type interval and a function length on intervals that represent their duration. We also define predicates started and ended on intervals to indicate that a given interval has started or ended, predicates nstarted and nended to preserve positive preconditions, and seven predicates on interval pairs corresponding to the seven relations in Figure 13. For example, the binary predicate `(before ?i1 ?i2 - interval)` is used to represent the first relation in Figure 13.

The domain has a single action template apply-interval with a single argument that is an interval. The action a_X = apply-interval(X) associated with interval X has

Relation	$\mathsf{pre}_o(a_X)$	$\mathsf{pre}_s(a_Y)$	$\mathsf{pre}_o(a_Y)$	$\mathsf{pre}_e(a_Y)$
$\mathsf{before}(X,Y)$	-	$\{\mathsf{ended}(X)\}$	-	-
$\mathsf{meets}(X,Y)$	-	-	$\{\mathsf{ended}(X)\}$	-
$\mathsf{overlaps}(X,Y)$	-	$\{\mathsf{started}(X), \mathsf{nended}(X)\}$	-	$\{\mathsf{ended}(X)\}$
$\mathsf{starts}(X,Y)$	$\{\mathsf{started}(Y)\}$	-	$\{\mathsf{started}(X)\}$	$\{\mathsf{ended}(X)\}$
$\mathsf{during}(Y,X)$	-	$\{\mathsf{started}(X)\}$	-	$\{\mathsf{nended}(X)\}$
$\mathsf{finishes}(Y,X)$	$\{\mathsf{nended}(Y)\}$	$\{\mathsf{started}(X)\}$	$\{\mathsf{nended}(X)\}$	-
$\mathsf{equal}(X,Y)$	$\{\mathsf{started}(Y), \mathsf{nended}(Y)\}$	-	$\{\mathsf{started}(X), \mathsf{nended}(X)\}$	-

Table 5: Modifications to actions a_X and a_Y as a result of a desired relation on X and Y.

duration $d(a_X) = \mathsf{length}(X)$ and is defined as follows:

$$\begin{aligned}
\mathsf{pre}_s(a_X) &= \{\mathsf{nstarted}(X)\},\\
\mathsf{pre}_o(a_X) &= \mathsf{pre}_e(a_X) = \emptyset,\\
\mathsf{cond}_s(a_X) &= \{\emptyset \rhd \{\mathsf{started}(X), \neg\mathsf{nstarted}(X)\}\},\\
\mathsf{cond}_e(a_X) &= \{\emptyset \rhd \{\mathsf{ended}(X), \neg\mathsf{nended}(X)\}\}.
\end{aligned}$$

To make sure that a_X is only applied once, precondition $\mathsf{nstarted}(X)$ ensures that a_X has not previously been started. The effect at start is to add $\mathsf{started}(X)$ and delete $\mathsf{nstarted}(X)$, and the effect at end is to add $\mathsf{ended}(X)$ and delete $\mathsf{nended}(X)$.

We now define instances of the domain. Each instance consists of intervals $X_1, \ldots, X_m$, each with a given duration. Each interval X_i, $1 \leq i \leq m$, is initially marked as not started and not ended. The goal state is a series of relations on interval pairs expressed in Allen's interval algebra that we want to achieve. For example, $\mathsf{overlaps}(X_1, X_2) \wedge \mathsf{overlaps}(X_2, X_3) \wedge \mathsf{overlaps}(X_3, X_4)$.

Given an instance of AIA, we compile the domain and instance into new PDDL domain and instance files. The reason is that we want to modify the individual action $a_X = \mathsf{apply\text{-}interval}(X)$ of each interval X depending on the desired relations in the goal state. Table 5 lists the modifications to actions a_X and a_Y as a result of a desired relation on intervals X and Y, in terms of additional preconditions on a_X and a_Y. We explain these modifications below.

- $\mathsf{before}(X,Y)$: Action a_X has to end before a_Y starts.

- $\mathsf{meets}(X, Y)$: Action a_X has to end exactly when a_Y starts (possibly simultaneously).
- $\mathsf{overlaps}(X, Y)$: Action a_X has to start before a_Y starts, end after a_Y starts, and end before a_Y ends.
- $\mathsf{starts}(X, Y)$: Actions a_X and a_Y have to start simultaneously (represented by contexts $\mathsf{started}(Y)$ of a_X and $\mathsf{started}(X)$ of a_Y), and a_X has to end before a_Y ends.
- $\mathsf{during}(Y, X)$: Action a_X has to start before a_Y starts and end after a_Y ends.
- $\mathsf{finishes}(Y, X)$: Actions a_X and a_Y have to end simultaneously (represented by contexts $\mathsf{nended}(Y)$ of a_X and $\mathsf{nended}(X)$ of a_Y), and a_X has to start before a_Y starts.
- $\mathsf{equal}(X, Y)$: Actions a_X and a_Y have to start and end simultaneously.

With these modifications, the temporal planning instance has a plan if and only if all relations on interval pairs are satisfied, with one exception: we cannot model the relation $\mathsf{meets}(X, Y)$ such that Y is forced to start at the same time as X ends, and Y starting after X ends also satisfies the precondition over all $\mathsf{ended}(X)$ of a_Y. To model the relation $\mathsf{meets}(X, Y)$, we use an auxiliary interval Z satisfying $\mathsf{length}(Z) = \mathsf{length}(X) + \mathsf{length}(Y)$, $\mathsf{starts}(X, Z)$ and $\mathsf{finishes}(Y, Z)$. An example of the PDDL domain and instance representation is shown in Appendix C.

4.2 Description

In this section, we describe an extension of the TP planner [15], that we call Simultaneous TP (STP), to handle simultaneous events. Recall that simultaneous events are those that occur at exactly the same time, and that we have to check whether or not the resulting joint event is valid.

Just as for TP, we use a modified version of Fast Downward [21] in order to get temporal solutions. Simple Temporal Networks (STN) [24] are used to represent

temporal constraints. 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 $\tau_j - \tau_i \leq c$. 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. In TP (and STP), there is a time variable for each temporal action. During the search process, a branch is pruned if the temporal constraints are violated. At the end of the section, we explain the additional modifications we introduced in Fast Downward to support STP.

Like TP, we impose a bound K on the number of active actions. Besides, we add the maximum value C of a cyclic counter (i.e. a counter that starts from 1 and resets to 1 after reaching the maximum value C). This cyclic counter increases each time an specific phase in the resulting compilation, called ending phase, is reached (detailed later).

Let $\Pi = \langle F, A, I, G \rangle$ be a temporal instance. The compilation of the STP(K, C) planner is a classical instance $\Pi_{K,C} = \langle F_{K,C}, A_{K,C}, I_{K,C}, G_{K,C} \rangle$. The new compilation has to ensure that joint events are valid, taking special care of the contexts of temporal actions. Recall that a context f 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. To properly handle contexts, our compilation divides each joint event into three phases:

1. End phase (immediately before the event). This is where active actions are scheduled to end during the joint event. In doing so, the counts of the corresponding contexts are decremented (else a context could not be deleted in the joint event itself).

2. Event phase (joint event itself). This is where the simultaneous events take place, both ending and starting actions. Thus, we check that preconditions hold and apply effects, simultaneously verifying that we do not violate the validity of the joint event.

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 joint event) and increment the context counts.

In the next sections we describe the new fluents and actions of the classical problem $\Pi_{K,C}$.

4.2.1 Fluents

The set of fluents $F_{K,C}$ extends F with 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 $c, 0 \leq c \leq K$, a fluent count_f^c indicating that c active actions have f as context.
- For each $c, 0 \leq c \leq K$, 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 $\mathsf{endingcount}^i, 1 \leq i \leq C$ indicating the state of the ending counter. This counter works cyclically, i.e. if $i = C$, then $i + 1 = 1$. These fluents help to distinguish between different states that share the same fluents but have different temporal meanings.

  As it may not be clear why $\mathsf{endingcount}$ fluents are needed, we give an example using the AIA domain (see Figure 14). In this example, the planner must find

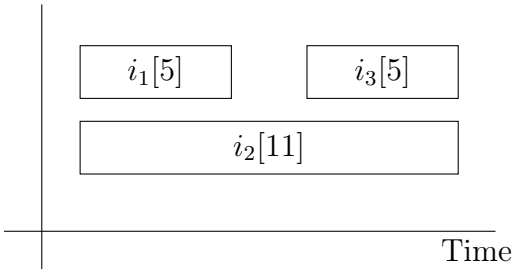
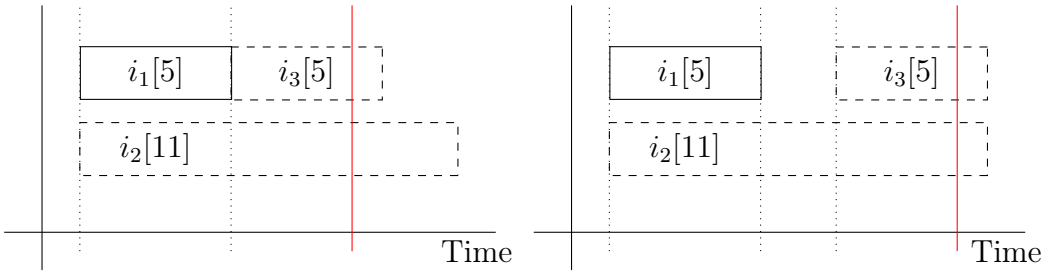


Figure 14: Example of the AIA domain: i_1 and i_2 start at the same time, while i_2 and i_3 end at the same time.

a solution such that i_1 and i_2 start simultaneously, while i_2 and i_3 end simultaneously. Figure 15 shows two different intermediate solutions the planner might visit. The black dotted lines indicate the execution of an end phase. The solid red line indicates the time at which A and B are considered equal to the planner if we did not take the number of ending phases into account. Although the time at which i_3 starts is different for each solution, the classical planner does not have this temporal information: it just knows that i_2 and i_3 are executing simultaneously.

Assuming no counters for ending phases are maintained, if the classical planner first visits A, it will mark A as visited and it will not find the solution (attempting to end i_3 at the same time as i_2 will violate the temporal constraints). Therefore, if the planner reaches B, then it will detect that it is in the visited list, and will skip it! Consequently, the planner will state that no solution could be found although there was one. If we add the ending counters, the planner can distinguish between A and B since they have run 2 and 3 ending phases respectively.

The resulting set of fluents $F_{K,C}$ can be expressed as follows:



Solution A Solution B

Figure 15: Possible intermediate solutions the planner can find for the example in Figure 14 (see text for more information).

$$
\begin{aligned}
F_{K,C} = F &\cup \{\mathsf{free}_a, \mathsf{active}_a : a \in A\} \\
&\cup \{\mathsf{count}_f^c : f \in F_o, 0 \leq c \leq K\} \\
&\cup \{\mathsf{concur}^c : 0 \leq c \leq K\} \\
&\cup \{\mathsf{endphase}, \mathsf{eventphase}, \mathsf{startphase}\} \\
&\cup \{\mathsf{starting}_a, \mathsf{ending}_a, \mathsf{nstarting}_a, \mathsf{nending}_a : a \in A\} \\
&\cup \{\mathsf{canpre}_f, \mathsf{caneff}_f : f \in F\} \\
&\cup \{\mathsf{endingcount}^i : 1 \leq i \leq C\}.
\end{aligned} \tag{4.1}
$$

Lemma 4.1. *The resulting number of fluents in the classical planning problem* $\Pi_{K,C}$ *is*

$$
\begin{aligned}
|F_{K,C}| &= |F| + 2\,|A| + (K+1)\,|F_o| + (K+1) + 3 + 4\,|A| + 2\,|F| + C \\
&= 3\,|F| + 6\,|A| + (K+1)\,(|F_o| + 1) + C + 3.
\end{aligned}
$$

Proof. By equation 4.1, the set of fluents $F_{K,C}$ in the classical problem $\Pi_{K,C}$ contains all the fluents F in the temporal planning problem Π. Besides, we add two new fluents (canpre_f, caneff_f) for each fluent $f \in F$. This results in $3\,|F|$ fluents.

We add 6 new fluents for each action (free_a, active_a, $\mathsf{starting}_a$, ending_a, $\mathsf{nstarting}_a$, $\mathsf{nending}_a$); thus, the size of the set increases $6\,|A|$. Then, we add $(K+1)\,|F_o|$ count_f^c fluents, and $(K+1)$ concur^c fluents; this results in an increase of $(K+1)\,(|F_o|+1)$ fluents.

We add C endingcount fluents, and three more fluents: endphase, eventphase and startphase. □

The initial state is defined as follows:

$$\begin{aligned} I_{K,C} = I &\cup \{\mathsf{free}_a : a \in A\} \cup \{\mathsf{count}_f^0 : f \in F_o\} \cup \{\mathsf{concur}^0\} \\ &\cup \{\mathsf{endphase}, \mathsf{endingcount}^1\} \\ &\cup \{\mathsf{nstarting}_a, \mathsf{nending}_a : a \in A\} \cup \{\mathsf{canpre}_f, \mathsf{caneff}_f : f \in F\}. \end{aligned}$$

The goal condition is:

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

4.2.2 Actions

In the STP(K, C) encoding, 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, $0 \leq c < K$, action $\mathsf{dostart}_a^c$ is defined as

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

For a given value of $c < K$, we can only start a in the event phase if there are c active actions and a is free. All contexts deleted 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. Specifically, if f is a precondition at start of a, f can no longer be used as an effect in this event phase. Likewise, if f is an effect at start of a, f can no longer be used neither as a precondition nor as an effect.

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

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

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 ending, 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:

$$\begin{aligned}
\text{pre} &= \mathsf{pre}_o(a) \\
&\quad \cup \{\mathsf{startphase}, \mathsf{starting}_a\}, \\
\text{cond} &= \{\emptyset \rhd \{\mathsf{active}_a, \mathsf{nstarting}_a, \neg\mathsf{starting}_a\}\} \\
&\quad \cup \{\mathsf{count}_f^c \rhd \mathsf{count}_f^{c+1}, f \in \mathsf{pre}_o(a) \wedge 0 \leq c < K\} \\
&\quad \cup \{\mathsf{count}_f^c \rhd \neg\mathsf{count}_f^c, f \in \mathsf{pre}_o(a) \wedge 0 \leq c < K\}.
\end{aligned}$$

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 $\mathsf{starting}_a$. Moreover, the context counts are incremented.

Finally, action finish_a is needed to schedule a for ending during the end phase:

$$\begin{aligned}
\text{pre} &= \{\mathsf{endphase}, \mathsf{active}_a\}, \\
\text{cond} &= \{\emptyset \rhd \{\mathsf{ending}_a, \neg\mathsf{active}_a, \neg\mathsf{nending}_a\}\} \\
&\quad \cup \{\mathsf{count}_f^{c+1} \rhd \mathsf{count}_f^c, f \in \mathsf{pre}_o(a) \wedge 0 \leq c < K\} \\
&\quad \cup \{\mathsf{count}_f^{c+1} \rhd \neg\mathsf{count}_f^{c+1}, f \in \mathsf{pre}_o(a) \wedge 0 \leq c < K\}.
\end{aligned}$$

The result is adding ending_a and deleting active_a and $\mathsf{nending}_a$. Context counts are also decremented.

The action set $A_{K,C}$ also needs three actions $\mathsf{setevent}$, $\mathsf{setstart}$ and setend for switching between phases. Action $\mathsf{setevent}$ is defined as

$$\begin{aligned}
\text{pre} &= \{\mathsf{endphase}\}, \\
\text{cond} &= \{\emptyset \rhd \{\mathsf{eventphase}, \neg\mathsf{endphase}\}\}.
\end{aligned}$$

Action setstart is defined as

$$\begin{aligned} \text{pre} &= \{\mathsf{eventphase}\} \cup \{\mathsf{nending}_a : a \in A\}, \\ \text{cond} &= \{\emptyset \rhd \{\mathsf{startphase}, \neg\mathsf{eventphase}\}\}. \end{aligned}$$

Note that we cannot leave the event phase unless all actions have ended. Action setend is defined as

$$\begin{aligned} \text{pre} &= \{\mathsf{startphase}\} \cup \{\mathsf{nstarting}_a : a \in A\} \cup \{\mathsf{canpre}_f, \mathsf{caneff}_f : f \in F\}, \\ \text{cond} &= \{\emptyset \rhd \{\mathsf{endphase}, \neg\mathsf{startphase}\}\} \\ &\quad \cup \{\mathsf{endingcount}^i \rhd \mathsf{endingcount}^j, 1 \leq i \leq C, j = (i+1) \mod C\} \\ &\quad \cup \{\mathsf{endingcount}^i \rhd \neg\mathsf{endingcount}^i, 1 \leq i \leq C\}. \end{aligned}$$

Note that we cannot leave the start phase unless all actions have started and all fluents are available as preconditions or effects. Moreover, the counter of end phases updates.

Finally, $A_{K,C}$ includes a reset action reset_f for each fluent $f \in F$, defined as

$$\begin{aligned} \text{pre} &= \{\mathsf{startphase}\}, \\ \text{cond} &= \{\emptyset \rhd \{\mathsf{canpre}_f, \mathsf{caneff}_f\}\}. \end{aligned}$$

These can only be applied in the start phase.

Figure 16 shows the interconnection between the different phases and the actions introduced in the compilation. Some actions allow to change the current phase, whereas others can be repeatedly applied (if the preconditions hold) while their associated phase is active. The execution begins with the endphase, as explained before.

Figure 17 focuses on the cycle each of the actions $a \in A$ passes through in the compilation. Note that after applying an action, the previous fluent is deleted,

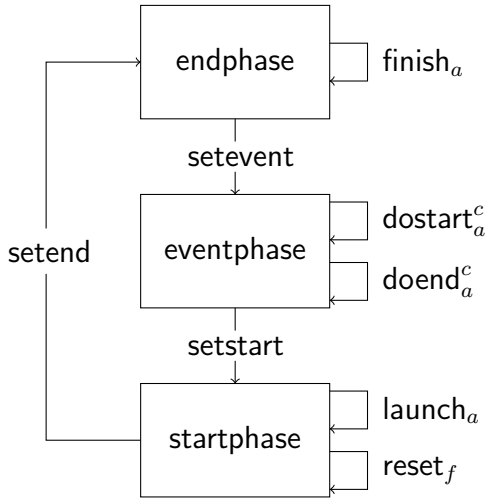


Figure 16: Interaction between the different actions introduced by the STP planner in the different phases.

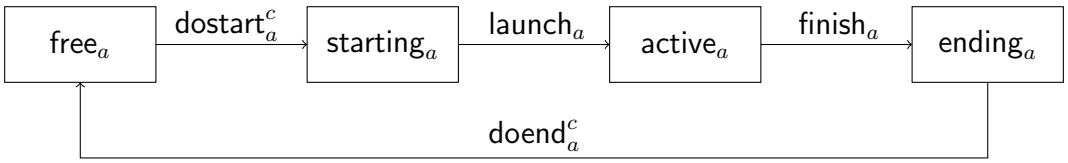


Figure 17: Fluents that are enabled each time an action in the compilation is executed.

e.g. when $\mathsf{dostart}^c_a$ is applied, free_a is deleted and $\mathsf{starting}_a$ is added. Besides, take into account that, at any time, the predicates $\mathsf{nstarting}_a$ and $\mathsf{nending}_a$ have the opposite value than $\mathsf{starting}_a$ and ending_a respectively.

The action set of the resulting classical planning problem $\Pi_{K,C}$ is

$$\begin{aligned} A_{K,C} = &\{\mathsf{dostart}^c_a, \mathsf{doend}^c_a : a \in A, 0 \leq c < K\} \\ &\cup \{\mathsf{launch}_a, \mathsf{finish}_a, \mathsf{setevent}_a, \mathsf{setstart}_a, \mathsf{setend}_a : a \in A\} \\ &\cup \{\mathsf{reset}_f : f \in F\}. \end{aligned} \tag{4.2}$$

Lemma 4.2. *The resulting number of actions of the classical planning problem* $\Pi_{K,C}$ *is*

$$\begin{aligned} |A_{K,C}| &= 2K\,|A| + 4\,|A| + |F| \\ &= (2K+4)\,|A| + |F|\,. \end{aligned}$$

Proof. By equation 4.2, the set of actions $A_{K,C}$ in the classical problem $\Pi_{K,C}$ contains two actions for each pair of $a \in A$ and $0 \leq 0 < K$ ($\mathsf{dostart}_a^c$ and doend_a^c). This results in $2K\,|A|$ actions. Then, for each $a \in A$, we add four actions: $\mathsf{launch_a}$, finish_a, $\mathsf{setevent}_a$, $\mathsf{setstart}_a$ and setend_a; thus, we add $4\,|A|$ actions more. Finally, we have a reset_f action for each $f \in F$, so we add $|F|$ actions to the actions set. □

Theorem 4.3 (Soundness)**.** *Let π' be a plan that solves the classical planning instance $\Pi_{K,C}$. Given π', we can always construct a temporal plan π that solves the temporal planning instance Π.*

Proof. Clearly, the system can only be in one of the three phases at once, and we can only cycle through the phases in the order $\mathsf{endphase} \rightarrow \mathsf{eventphase} \rightarrow \mathsf{startphase} \rightarrow \mathsf{endphase}$ using actions $\mathsf{setevent}$, $\mathsf{setstart}$ and setend. 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 the fluent concur^0 of $G_{K,C}$).

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. Due to the precondition $\mathsf{nstarting}_a$ of setend, 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 fluent $f \in F$, we show that the fluents canpre_f and caneff_f are true each

time we apply action setevent, i.e. when we enter the event phase. The fluents in question are true in the initial state, i.e. in the end phase, and are only deleted by actions of type dostart and doend, which are only applicable in the event phase. However, the precondition $\{\mathsf{canpre}_f, \mathsf{caneff}_f\}$ of action setend requires us to reset f in the start phase using action reset_f, and there are no actions that delete these fluents that are applicable in the end phase.

We have thus shown that any solution plan π' for $\Pi_{K,C}$ has the following form:

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

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 in the event phase, deleting fluents canpre_f and caneff_f prohibits f from appearing as a precondition or effect of another event in the same event phase (note that reset_f is not applicable until the following start phase). Likewise, each time a fluent f appears as a precondition, deleting fluent caneff_f prohibits f from appearing as an effect of another event. Because of the mechanism for finishing and launching temporal actions, however, 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.

To obtain a temporal plan π, we associate each temporal action a that occurs in π' with the starting time $\tau_a = -d_{a0}$ given by the STN. The modified time constraints are defined such that all events that are part of a joint event are scheduled to occur at the same time. Hence the events of the induced event sequence π_E occur exactly in the same order as the events in π'. Since π' solves $\Pi_{K,C}$, π_E solves Π with respect to fluents in Π, and contexts are respected because of the count mechanism. □

We have introduced some additional modifications to support STP. In addition to

the latest event e, we also keep track of the first event e^* to take place in the current event phase. Unlike TEMPO, while we remain in the same event phase the imposed constraint $\tau_e \leq \tau_a$ or $\tau_e \leq \tau_a + d(a)$ is not strict, since e is intended to be simultaneous with starting or ending a. When we apply the action setstart, we impose the additional constraint $\tau_e \leq \tau_{e^*}$, thus ensuring that all events of the current event phase are scheduled exactly at the same time. When we reach the next event phase, we reset the first event e^* and impose a strict constraint $\tau_e < \tau_{e^*}$ to ensure that non-simultaneous events are spaced apart.

4.3 Results

We performed an evaluation in all 10 domains of the temporal track of IPC-2014. Moreover, we added the DRIVERLOGSHIFT (DLS) domain [25], the AIA domain (see Section 4.1), and a domain based on an STN example introduced by Cushing *et al.* (2007) [26] (from now on, this domain is referred as the CUSHING domain).

The STP planner was executed for values of K in the range $1, \ldots, 4$ and with a fixed $C = 10$. We compared STP with other planners that compile temporal planning problems into classical planning ones. Firstly, we ran the TP planner using the same values of K than STP. Secondly, the TPSHE planner used the LAMA-2011 setting of Fast Downward to solve the compiled instance. We also ran experiments for POPF2 [27] (the runner-up at IPC-2011), YAHSP3-MT [28] (the winner at IPC-2014), and ITSAT [29] (a SAT-based temporal planner).

Table 6 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 TPSHE planner solved at least one instance for all domains except for CUSHING and FLOORTILE. Besides, it obtained the highest IPC score and coverage.

The only domains with required concurrency not in the form of single hard envelopes are AIA and CUSHING. TP and TPSHE cannot handle simultaneous events. Thus,

Table 6: Comparison among presented planners. IPC quality score / coverage per domain for each planner. Total number of instances of each domain between brackets.

	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	2.51/6	**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]	6.19/**20**	5.36/18	5.79/17	5.31/16	5.33/16	2.52/9	2.83/9	2.83/9	2.59/8	12/13	**16.84**/**20**	0.96/6
RTAM[20]	**16**/**16**	4.62/9	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	7/18	4.94/13	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]	0.05/8	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	**119.58**/**159**	31.55/72	64.46/107	66.51/127	68.12/125	17.01/37	49.3/72	55.36/87	55.49/75	79.22/81	45.8/62	92.3/106

they only could solve few the AIA instances. In the case of CUSHING, TPSHE, TP(1) and TP(2) could not solve any instance, while TP with higher values of K (TP(3), TP(4)) could solve them all.

In the case of TP, note that if K is increased, more instances are generally solved. For example, TP(3) solves 20 more instances than TP(2). However, TP(4) solves two less instances than TP(3). This might happen because of the additional mechanisms that TP needs for higher values of K. Therefore, some instances become harder to solve.

STP is the only algorithm that is able to solve all instances of the AIA domain (given its support for simultaneous events), specifically with $K = 4$. As in TP, although higher values of K generally provide better score and coverage, STP(4) solves twelve less instances than STP(3). However, TP scales better than STP, since the decrease in the number of solved instances is not as high as in STP. On the other hand, TP solves many more instances than STP. Since the STP compilation has a higher number of fluents and actions than TP, it is more difficult for the planner to find solutions, so it is a possible reason for which STP does not solve many instances in domains like RTAM or SATELLITE.

TPSHE and ITSAT are the planners that return solutions with higher quality, i.e. shorter makespans. Nevertheless, they behave differently for different domains. For example, they are the only algorithms providing solutions for the TMS domain, but TPSHE solutions are much longer than ITSAT's. This happens because TPSHE

does not take the duration of actions into account and does not exploit parallelism.

The makespan of plans returned by STP are not as good as those of the other planners, so the IPC scores are lower. It is just comparable to YAHSP3-MT if $K \geq 2$. We want to highlight that YAHSP3-MT solved all instances in IPC-2014 in the MapAnalyser and RTAM domains. As shown in the table, we have not been able to reproduce these results (maybe due to the provided time limit).

Chapter 5

Smart Mobility using Temporal Planning

In this chapter, we introduce the SmartMobility domain. Firstly, we give a general description of the concepts concerning this domain. Then, we show how we can model this problem as a temporal planning problem using PDDL. Finally, we describe the approach we have followed for implementing the overall system and how we evaluated it.

5.1 Motivation

Collective Adaptive Systems (CAS) consist of diverse heterogeneous agents composing a socio-technical system [30, 31]. The agents that belong to these systems self-adapt in order to leverage other agents' resources and capabilities to perform their task more efficiently or effectively. Adaptation is done in a collective way: the agents must be capable of automatically and simultaneously self-adapt while preserving the collaboration and benefits of the system. The goals of each of the agents should always be fulfilled after an adaptation.

Agents are organized in *ensembles*. They can be created spontaneously and change over time: different agents may decide to joint and/or leave an existing ensemble

dynamically and autonomously. Additionally, adaptation in these systems is triggered by the run-time occurrence of an unwanted circumstance, called *issue*. An issue is handled by an *issue resolution* process that involves agents, affected by the issue, to collaboratively adapt with minimal impact on their own preferences.

The SmartMobility domain is a promising area for CAS. Organizing and managing mobility services within a city, meeting travelers expectations and properly exploiting the available transport resources, is becoming a more and more complex task.

Traditional transportation models are currently being substituted by other more social models aiming to provide a more flexible, customized and collective way of organizing transport.. For example, carpooling is a service that provides procedures for offering resources (cars) and to ask for them (i.e. searching a ride). Thus, this service offers a way to organize a team of citizens that need to reach equal or closed destinations starting also from different locations.

In this new kind of models, passengers select the resources they want to use. However, coordination between the participants is needed to reach each destination, preferably in time. Indeed, depending on their location, their route and additional activities (like refueling), they coordinate their departure times by communicating with each other. By sharing a resource, people save gas and money, as well as reduce auto emissions, pollution, etc. Even if services as carpooling look very promising and more sustainable, they have limits on how the resolution of unwanted situations are managed.

In the following sections, we propose a method for resolving issues in this domain. Besides, we will demonstrate the benefits of adapting collectively instead of selfishly (e.g. all agents use their own car).

5.2 Problem Modeling

In this section we model the problem of smart mobility using temporal planning. Even though temporal planning was not specifically invented with multiple agents in

mind, temporal actions are concurrent and have variable duration. Besides, temporal planning allows modeling complex features such as deadlines, conditions during the application of actions, and effects occurring at arbitrary time points.

The smart mobility domain that we consider consists of a set of agents formed by passengers and carpools. Each agent can move between different locations specified by a map. Each agent starts from a specific location, and the goal is to reach a target location. To simplify the model we impose the following restrictions:

- Carpools have infinite capacity.
- Each link between two locations has a fixed distance.
- Each link is a footpath, a street, or both. Footpaths are used by passengers, while streets are used by carpools.
- Passengers move uniformly at 1 m/s. Carpools move uniformly at a speed that depends on the speed limit.
- A passenger can embark a carpool only if they are at the same location. This action takes 1 time unit.
- A passenger can debark a carpool at any location reached by the carpool. This action takes 1 time unit.

Figure 18 shows a simple instance of this domain that will be used for illustration. The notation is as follows:

- Locations are denoted by l_i.
- Footpaths are represented by dashed lines, while streets are represented by solid lines.
- A label d, s indicates that the distance between locations is d and that the speed limit is s (for carpools only).

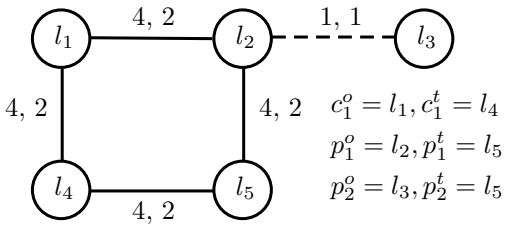


Figure 18: Example of the smart mobility domain.

- Carpools are denoted by c, while passengers are denoted by p. They are identified by subindices.
- The origin location of an agent a is denoted by a^o, while its target location is denoted by a^t.

To define a temporal planning problem $\Pi = \langle F, A, I, G \rangle$, we first define the fluents in F as follows:

- For each agent a and location l, a fluent $\mathsf{at}(a, l)$ indicating whether a is at l.
- For each passenger p and carpool c, a fluent $\mathsf{in}(p, c)$ indicating whether p is inside c.
- For each passenger p and location pair l_1, l_2, a fluent $\mathsf{has\text{-}footpath}(p, l_1, l_2)$ indicating whether there is a footpath for p between l_1 and l_2.
- For each location pair l_1, l_2, a fluent $\mathsf{has\text{-}street}(l_1, l_2)$ indicating whether there is a street between l_1 and l_2.

The actions in the set A are defined as follows:

- For each passenger p, carpool c and location l, an action $\mathsf{embark}(p, c, l)$ that makes p embark c at l.
- For each passenger p, carpool c and location l, an action $\mathsf{debark}(p, c, l)$ that makes p debark c at l.

- For each passenger p and pair of locations l_1, l_2, an action $\mathsf{walk}(p, l_1, l_2)$ that makes p walk from l_1 to l_2.
- For each carpool c and pair of locations l_1, l_2, an action $\mathsf{travel}(c, l_1, l_2)$ that makes c travel from l_1 to l_2.

Figure 19 defines the four types of actions in the graphical representation previously introduced. For example, the action $\mathsf{travel}(c, l_1, l_2)$ has duration $dist(l_1, l_2)/speed(l_1, l_2)$, which depends on the distance and speed limit between l_1 and l_2. The precondition at start is that c is at l_1, and the effect at start is that c is no longer at l_1 (since c is now traveling towards l_2). The effect at end is that c arrives at l_2. The precondition over all requires there to be a street between l_1 and l_2.

Finally, the initial state I and goal condition G encode origin and target locations of agents, as well as footpaths and streets. A temporal plan for solving the example in Figure 18 is:

start time	action	duration
0.0000	$\mathsf{travel}(c_1, l_1, l_2)$	2.0000
0.0000	$\mathsf{walk}(p_2, l_3, l_2)$	1.0000
2.0002	$\mathsf{embark}(p_1, c_1, l_2)$	1.0000
2.0002	$\mathsf{embark}(p_2, c_1, l_2)$	1.0000
3.0004	$\mathsf{travel}(c_1, l_2, l_5)$	2.0000
5.0006	$\mathsf{debark}(p_2, c_1, l_5)$	1.0000
5.0006	$\mathsf{debark}(p_1, c_1, l_5)$	1.0000
6.0008	$\mathsf{travel}(c_1, l_5, l_4)$	2.0000

Each action a_i is labeled with its starting time t_i and duration $d(a_i)$. Passenger p_1 and carpool c_1 both travel to location l_2, where both passengers embark the carpool. The carpool then travels to the target location l_5 for both passengers, where they debark. Finally, the carpool travels to its target location l_4. Note that some actions occur in parallel.

Appendix D contains a full example of this domain for instance in Figure 18 using

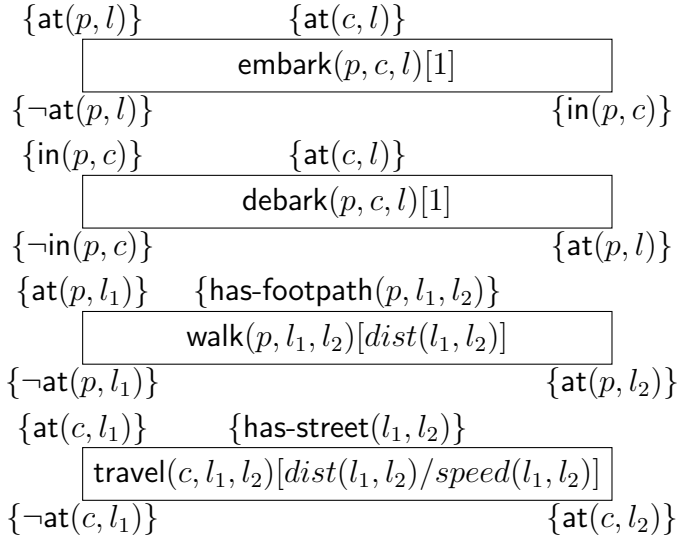


Figure 19: Definition of temporal actions for smart mobility.

PDDL.

5.3 Evaluation

In this section, to show our approach in action, we have implemented a demonstrator of the carpooling scenario[1]. The, by using this tool, we provide some results showing the importance of doing collective adaptation instead of selfish.

5.3.1 Carpooling Demonstrator

The demonstrator has been implemented as an extension of the *Collective Adaptation Engine (CAE)* [32, 33] and manages different modules for computing adaptation solutions and statistics. These new modules are: (i) the *Scenario Builder*, (ii) the *Concurrent Planner* and, (iii) the *Scenario Viewer*.

The *Scenario Builder* builds the initial state of the carpooling application given:

- The number of passengers and the number of carpools.
- Two latitude-longitude pairs to form the boundaries of the map area.

[1]We have made the *Smart Carpooling Demo* available at the following link: `https://github.com/aig-upf/smart-carpooling-demo`

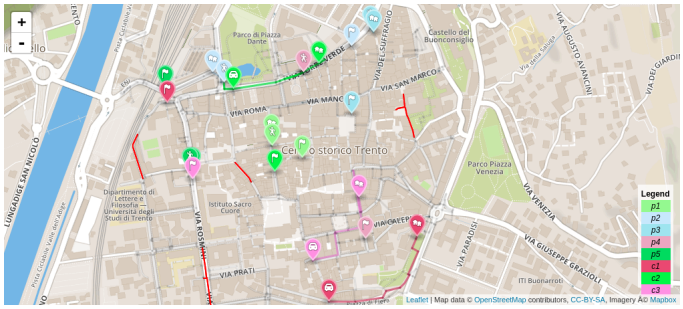


Figure 20: Screenshot of the Scenario Viewer.

- The minimum and maximum walking ranges of the passengers. These quantities express the preference for how far a passenger is willing to walk from/to their origin/target positions.

The demonstrator uses a real map (localized in the Trento city area) obtained from OpenStreetMap (OSM)[2]. OSM maps contain a list of the locations in the map and the links between them. The latitude and longitude is given for each location. On the other hand, each link between two locations has a maximum speed limit and a list of intermediate locations. Random initial and target positions inside the boundaries are assigned to both passengers and carpools. Moreover, each passenger has a random walking range between the minimum and maximum ranges specified. To build an initial random scenario, the OSM parser is used. The *OSM parser* is the responsible for parsing the input OSM maps. Thus, it eases the access to the information of the map by other modules. For example, it allows to get the set of nodes inside an specific area of the map bounded by two latitude-longitude pairs.

The resulting initial state is processed by the CAE to create the first set of ensembles (i.e. carpool rides) that are instantiated and executed in the demonstrator. The instantiation of each ensemble is done using the *Concurrent Planner*, which is responsible for finding an initial plan for all involved agents in the various ensembles. Once the concurrent plan is found, all the agents (i.e. drivers and passengers) start their execution.

The planning algorithm we use is called TPSHE [15]. The reason we choose this al-

[2] `http://www.openstreetmap.org/`.

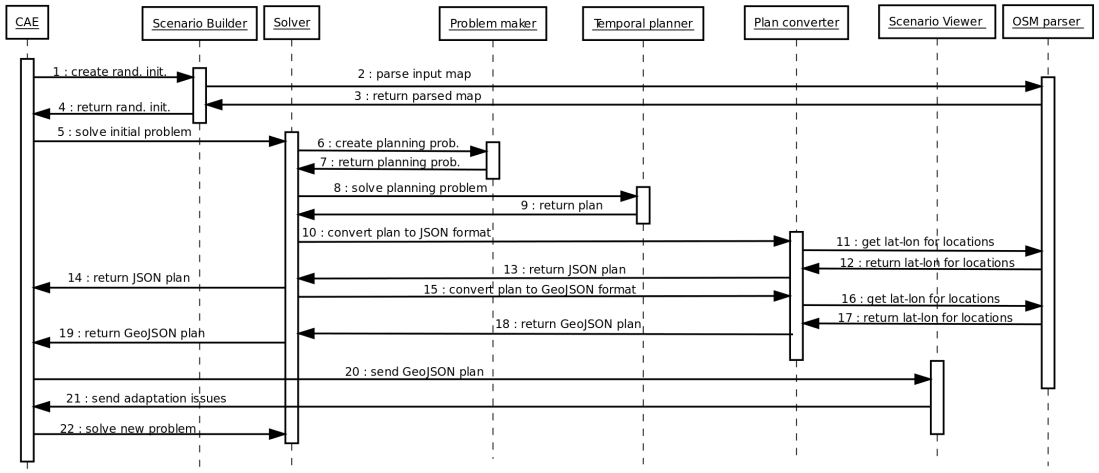


Figure 21: Sequence diagram showing the communication between the different implemented modules.

gorithm is that if performs well in the International Planning Competition (IPC) [34] domains that require concurrent actions. Since on criterion of smart mobility is to bring each agent to their goal in the shortest time possible, we want to exploit the ability of temporal planning to express concurrent plans. We remark that TPSHE is not an optimal algorithm, although it attempts to minimize the total duration (i.e. makespan) of the temporal plan.

During a normal execution of the ensembles, the demonstrator provides a way to generate adaptation issues (e.g. blocked streets), and starts the execution of the collective adaptation algorithm implemented in the CAE. To compare collective adaptation with a selfish approach to resolving adaptation issues, the CAE also includes a module able to run the two respective techniques (collective and selfish) per each adaptation issue and memorize relevant statistics.

Just as for the initial plan, every time that an adaptation issue is triggered, the *Concurrent Planner* component is invoked and executes the following steps:

1. It converts the input problem into a temporal planning problem following the specification described in Section 2.4. To introduce blocked streets in the planning problem, the corresponding $\mathsf{has\text{-}street}(l_1, l_2)$ fluents have to be removed from the initial state.

2. It executes a temporal planner (TPSHE) to compute a solution. There are two types of solutions:

   - Collective: Agents interact to reach their target locations (e.g. a carpool picking up a passenger).
   - Selfish: Agents do not interact, i.e. they try to reach their target locations by themselves.

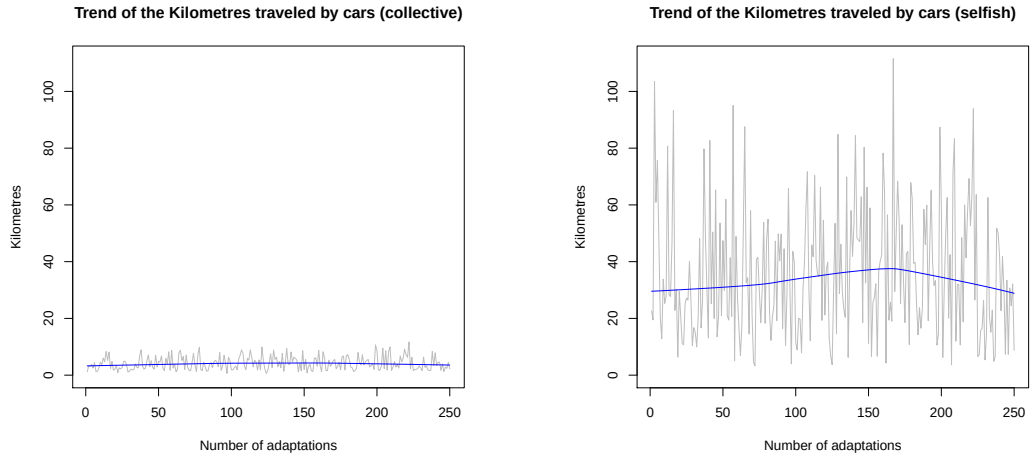
3. It converts the solution into a more user-friendly format. The OSM parser is used in this step to get the latitude-longitude coordinates of each location in the plan. JSON is used by the CAE to compute statistics, while GeoJSON is used by the Scenario Viewer.

The *Scenario Viewer* provides a graphical representation of the carpooling application through a map showing all the agents of the domain (cars and passengers) in action executing their plans (see Figure 20). Moreover, there is one chart for each type of solution indicating the traveled distance by each agent. Thus, it is easy to find out if the distance traveled by cars in the collective solution is lower than in the selfish solution.

The overall plan can be navigated step by step, i.e. it shows where each agent is at a given intermediate time point. Besides, it is also possible to block streets by clicking on the street.

An adaptation issue can be introduced at any timestamp using the Scenario Viewer and start the execution of the resolution algorithm. When a solution is found, the viewer refreshes automatically and displays the new plan assigned to each agent.

The sequence diagram depicted in Figure 21 shows the communication between the modules that have been previously explained when a new ensemble is created and when an intra-ensemble adaptation issue must be resolved.



(a) Collective Adaptation (b) Selfish Adaptation

Figure 22: Kilometers traveled by cars.

5.3.2 Results

To evaluate the performance of the adaptation approach proposed in this paper, we ran the carpooling demonstrator in continuous mode and collected information on 250 ensembles with adaptation activated. The run was performed on a Windows laptop with dual-core 2.8GHz CPU with 8GB of RAM.

Each time we computed a collective plan, we measured a number of indicators that characterized the complexity of the problem and the timing. We then collected these indicators and organized them into charts in order to draw conclusions about the performance and scalability of our approach.

Figure 22 shows the number of kilometers traveled by cars in each of the 250 instances of adaptation. We compare, on one hand, the outcome of collective adaptation, and on the other hand, the outcome of selfish adaptation. In the case of selfish adaptations, we assume that all passengers use their own car. It is clear that the number of kilometers traveled by cars is significantly smaller for collective adaptation, in spite of the fact that the collaborative plans that we compute are not optimal (since TPSHE is not an optimal temporal planner). This illustrates the importance of collective adaptation in general, and for smart mobility in particular.

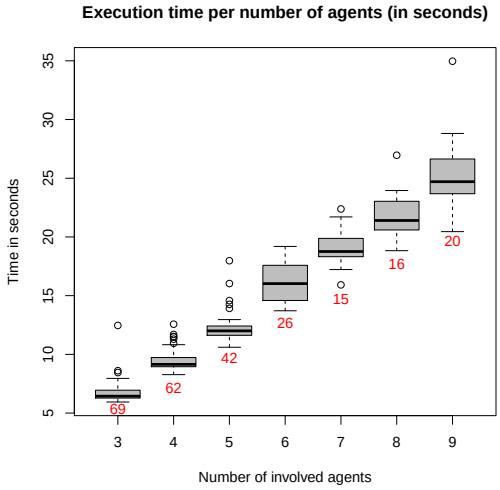


Figure 23: Execution time per number of agents (in seconds), with number of adaptation instances in red.

Figure 23 shows the average execution time of collective adaptation as a function of the number of agents. Numbers in red indicate the number of adaptation instances per number of agents (for a total of 250). We can see that execution time is approximately linear in the number of agents, despite the fact that the search space increases exponentially with the number of agents (since each additional propositional variable doubles the total number of possible states). This indicates that our approach scales reasonably well to large numbers of agents.

Chapter 6

Related Work

In this section, we briefly explain some related work to the content introduced in Chapters 3, 4 and 5. The title of each section corresponds is the same than its corresponding chapter's title.

6.1 Compilation of Multiagent Planning to Classical Planning

Several other authors consider the problem of concurrent multiagent planning. Boutilier and Brafman (2001) [12] describe a partial-order planning algorithm for solving MAPs with concurrent actions, based on their formulation of concurrency constraints, but do not present any experimental results. Jonsson and Rovatsos (2011) [10] present a best-response approach for MAPs with concurrent actions, where each agent attempts to improve its own part of a concurrent plan while the actions of all other agents are fixed. However, their approach only serves to improve an existing concurrent plan, and is unable to compute an initial concurrent plan. FMAP [9] also allows agents to execute actions in parallel, but the authors do not present experimental results for MAP domains that require concurrency.

Crosby *et al.* (2014) [11], as explained in Section 2.3.2, describe a method which is quite similar to ours since it also converts MAPs with concurrent actions into

single-agent planning problems. The authors only present results from the MAZE domain, and the actions associated to a specific object subset are always of the same type. Consequently, all agents always perform the same action in each concurrent action (either rowing a boat, or crossing a bridge, etc.).

In contrast, the concurrency constraints in our work can be arbitrary and are not tied to specific objects subsets. For example, in WORKSHOP, there are several instances for which concurrent actions have to include different individual actions (push switch and turn key in order to open a security door, and lift pallet and examine pallet in order to perform inventory). Moreover, ours is the only method that can handle concurrency in conditional effects, as demonstrated in the TABLEMOVER domain, in which the effect of the blocks falling on the floor is conditional on how many agents are lifting or putting down a table simultaneously.

6.2 STP: Simultaneous Temporal Planner

Several authors have previously provided theoretical justification for splitting durative actions into classical actions [35] and proposed a compilation for doing so. An early approach, LPGP [36], turned out to be unsound and incomplete since it failed to (1) ensure that temporal actions end before reaching the goal, (2) ensure that the contexts of temporal actions are not violated, and (3) ensure that temporal constraints are preserved. Rintanen [37] 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 [15]. Both planners are based on compiling temporal planning problems to classical planning problems. TPSHE only handles instances for which required concurrency is in the form of single hard

envelopes. On the other hand, as explained in Chapter 4, partially compiles temporal actions into classical planning and introduces an STN into the Fast Downward classical planner to enforce temporal constraints. There are more algorithms that use STNs, like POPF [27] and OPTIC [38]. More precisely, POPF encodes the STN using linear programming which allows it to compute plans with actions that cause continuous linear numeric changes, while 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 *landmarks*. This group includes the TEMPLM planner [39] 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.* [40] 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 [29] 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 for temporal planning instances with the aim of producing a SAT Modulo Theories encoding where action sequencing and scheduling are tightly integrated [41]. Rintanen [42] showed that while PDDL forces temporal gaps in action scheduling (which have a performance penalty), NDL avoids such gaps using the notion of resources and resulting in better performances (the number of solved instances in IPC domains is higher).

6.3 Smart Mobility using Temporal Planning

There have been other attempts to compute joint plans for multiple agents in navigation scenarios [43, 44]. However, the focus is usually on satisfying certain constraints (e.g. that the agents should not collide), rather than collaboration between agents. *Numeric planning* [45] makes it possible to reason with costs and resources during navigation, but numeric planners are typically more complex and only plan for a single agent at a time.

Chapter 7

Conclusions and Future Work

This work makes several contributions to concurrent planning. A common framework (notation and definitions) is introduced for different planning forms. We mainly focused on the relationship that concurrent planning maintains with multiagent and temporal planning. We proposed methods for compiling concurrent multiagent and temporal planning problems into classical planning problems.

In the case of the concurrent multiagent compilation, we avoid choosing between an exponential number of joint actions, which is a problem that current planners face. Instead, the number of resulting actions is linear in the description of the multiagent planning problem while respecting explicit concurrency constraints. The transformation has been proven to be sound and complete. In future work, it would be interesting to:

1. Explore strategies for encouraging concurrent actions that involve several agents.

2. Compare the performance of the resulting compilation using a classical planner against those planners participating in the CoDMAP 2015 competition.

3. For each domain, compare the number of actions in our compilation and the exponential number of actions that current multiagent planners deal with.

Regarding temporal planning, we propose STP, an extension of the TP planner to support simultaneous events (i.e. events that occur exactly at the same time) while keeping temporal constraints safe. Previous temporal planners did not support problems of this kind. To justify its existence, we introduce a domain called AIA based on Allen's Interval Algebra, where different problems requiring simultaneous events can be created.

The main problem of methods that use a compilation from temporal to classical planning is the loss of information about duration. Therefore, the quality of planners like TPSHE, TP or STP is much worse in some domains than those of other temporal planners. In the future, it would be convenient to investigate how to partially incorporate information about duration in the compilation.

Finally, we propose a real (although simplified) problem on which we can apply concurrent planning (specifically, temporal planning). This problem is concerned with smart mobility, where different vehicles and passengers plan together a path that allows them to reach their target positions. There are several possible ways to extend the work on concurrent planning to compute collaborative plans. In our current version, the temporal planner attempts to minimize the total time of travel, but does not take into account the cost of fuel or, alternatively, the cost of contamination. Integrating such costs into the planning process requires careful modeling, as well as access to a temporal planner that is sensitive to such costs. Another aspect is that our current system only computes approximately optimal plans, since optimal planning is typically harder than satisficing planning (finding any solution). Hence, there is a tradeoff between the response time and the quality of the proposed solution.

List of Figures

List of Tables

Bibliography

[1] Kitano, H., Asada, M., Kuniyoshi, Y., Noda, I. & Osawa, E. RoboCup: The Robot World Cup Initiative. In *Agents*, 340–347 (1997).

[2] Komenda, A., Stolba, M. & Kovacs, D. L. The International Competition of Distributed and Multiagent Planners (CoDMAP). *AI Magazine* **37**, 109–115 (2016). URL `http://www.aaai.org/ojs/index.php/aimagazine/article/view/2658`.

[3] Brafman, R. I. & Domshlak, C. From One to Many: Planning for Loosely Coupled Multi-Agent Systems. In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling, ICAPS 2008, Sydney, Australia, September 14-18, 2008*, 28–35 (2008). URL `http://www.aaai.org/Library/ICAPS/2008/icaps08-004.php`.

[4] Crosby, M., Rovatsos, M. & Petrick, R. P. A. Automated Agent Decomposition for Classical Planning. In *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling, ICAPS 2013, Rome, Italy, June 10-14, 2013* (2013). URL `http://www.aaai.org/ocs/index.php/ICAPS/ICAPS13/paper/view/6051`.

[5] Muise, C., Lipovetzky, N. & Ramirez, M. MAP-LAPKT: Omnipotent Multi-Agent Planning via Compilation to Classical Planning. In *Competition of Distributed and Multiagent Planners* (2015). URL `http://www.haz.ca/papers/muise_CoDMAP15.pdf`.

[6] Borrajo, D. Plan Sharing for Multi-Agent Planning. In *DMAP 2013 - Proceedings of the Distributed and Multi-Agent Planning Workshop at ICAPS*, 57–65 (2013).

[7] Tozicka, J., Jakubuv, J. & Komenda, A. Generating Multi-Agent Plans by Distributed Intersection of Finite State Machines. In *ECAI 2014 - 21st European Conference on Artificial Intelligence, 18-22 August 2014, Prague, Czech Republic - Including Prestigious Applications of Intelligent Systems (PAIS 2014)*, 1111–1112 (2014). URL `http://dx.doi.org/10.3233/978-1-61499-419-0-1111`.

[8] Stolba, M., Fiser, D. & Komenda, A. Potential Heuristics for Multi-Agent Planning. In *Proceedings of the Twenty-Sixth International Conference on Automated Planning and Scheduling, ICAPS 2016, London, UK, June 12-17, 2016.*, 308–316 (2016). URL `http://www.aaai.org/ocs/index.php/ICAPS/ICAPS16/paper/view/13117`.

[9] Torreño, A., Onaindia, E. & Sapena, O. FMAP: Distributed cooperative multi-agent planning. *Appl. Intell.* **41**, 606–626 (2014). URL `http://dx.doi.org/10.1007/s10489-014-0540-2`.

[10] Jonsson, A. & Rovatsos, M. Scaling Up Multiagent Planning: A Best-Response Approach. In *Proceedings of the 21st International Conference on Automated Planning and Scheduling, ICAPS 2011, Freiburg, Germany June 11-16, 2011* (2011). URL `http://aaai.org/ocs/index.php/ICAPS/ICAPS11/paper/view/2696`.

[11] Crosby, M., Jonsson, A. & Rovatsos, M. A Single-Agent Approach to Multiagent Planning. In *ECAI 2014 - 21st European Conference on Artificial Intelligence, 18-22 August 2014, Prague, Czech Republic - Including Prestigious Applications of Intelligent Systems (PAIS 2014)*, 237–242 (2014). URL `https://doi.org/10.3233/978-1-61499-419-0-237`.

[12] Boutilier, C. & Brafman, R. I. Partial-Order Planning with Concurrent Interacting Actions. *J. Artif. Intell. Res. (JAIR)* **14**, 105–136 (2001). URL http://dx.doi.org/10.1613/jair.740.

[13] Kovacs, D. L. A Multi-Agent Extension of PDDL3.1. In *Proceedings of the 3rd Workshop on the International Planning Competition (IPC)*, 19–27 (2012).

[14] Crosby, M. A Temporal Approach to Multiagent Planning with Concurrent Actions. In *PlanSig 2013 - Workshop of the UK Planning and Scheduling Special Interest Group* (2013).

[15] Celorrio, S. J., Jonsson, A. & Palacios, H. 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 (2015). URL http://www.aaai.org/ocs/index.php/ICAPS/ICAPS15/paper/view/10623.

[16] Geffner, H. & Bonet, B. *A Concise Introduction to Models and Methods for Automated Planning*. Synthesis Lectures on Artificial Intelligence and Machine Learning (Morgan & Claypool Publishers, 2013). URL https://doi.org/10.2200/S00513ED1V01Y201306AIM022.

[17] Nebel, B. On the Compilability and Expressive Power of Propositional Planning Formalisms. *J. Artif. Intell. Res. (JAIR)* **12**, 271–315 (2000). URL https://doi.org/10.1613/jair.735.

[18] Crosby, M. *Multiagent Classical Planning*. Ph.D. thesis, University of Edinburgh (2014). URL https://books.google.es/books?id=3U-yoQEACAAJ.

[19] Fox, M. & Long, D. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *J. Artif. Intell. Res. (JAIR)* **20**, 61–124 (2003). URL https://doi.org/10.1613/jair.1129.

[20] Howey, R., Long, D. & Fox, M. 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 (2004). URL https://doi.org/10.1109/ICTAI.2004.120.

[21] Helmert, M. The Fast Downward Planning System. *J. Artif. Intell. Res. (JAIR)* **26**, 191–246 (2006). URL http://dx.doi.org/10.1613/jair.1705.

[22] Richter, S. & Westphal, M. The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks. *Journal of Artificial Intelligence Research* **39**, 127–177 (2010).

[23] Allen, J. F. Maintaining Knowledge About Temporal Intervals. *Commun. ACM* **26**, 832–843 (1983).

[24] Dechter, R., Meiri, I. & Pearl, J. Temporal Constraint Networks. *Artif. Intell.* **49**, 61–95 (1991). URL https://doi.org/10.1016/0004-3702(91)90006-6.

[25] Coles, A., Fox, M., Halsey, K., Long, D. & Smith, A. Managing concurrency in temporal planning using planner-scheduler interaction. *Artif. Intell.* **173**, 1–44 (2009). URL https://doi.org/10.1016/j.artint.2008.08.003.

[26] Cushing, W., Kambhampati, S., Mausam & Weld, D. S. 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 (2007). URL http://ijcai.org/Proceedings/07/Papers/299.pdf.

[27] Coles, A. J., Coles, A., Fox, M. & Long, D. 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 (2010). URL http://www.aaai.org/ocs/index.php/ICAPS/ICAPS10/paper/view/1421.

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

[29] Rankooh, M. F. & Ghassem-Sani, G. ITSAT: An Efficient SAT-Based Temporal Planner. *J. Artif. Intell. Res.* **53**, 541–632 (2015). URL `https://doi.org/10.1613/jair.4697`.

[30] FoCAS Manifesto – A roadmap to the future of Collective Adaptive Systems (2016). `http://www.focas.eu/focas-manifesto.pdf`.

[31] Zambonelli, F., Bicocchi, N., Cabri, G., Leonardi, L. & Puviani, M. On Self-Adaptation, Self-Expression, and Self-Awareness in Autonomic Service Component Ensembles. In *Fifth IEEE Conference on Self-Adaptive and Self-Organizing Systems, SASOW 2011, Ann Arbor, MI, USA, October 3-7, 2011, Workshops Proceedings*, 108–113 (2011). URL `https://doi.org/10.1109/SASOW.2011.24`.

[32] Bucchiarone, A. *et al.* An Approach for Collective Adaptation in Socio-Technical Systems. In *IEEE SASO Workshops*, 43–48 (2015).

[33] Bozhinoski, D., Bucchiarone, A., Malavolta, I., Marconi, A. & Pelliccione, P. Leveraging Collective Run-Time Adaptation for UAV-Based Systems. In *SEAA'16*, 214–221 (2016).

[34] Fox, M. & Long, D. The 3rd International Planning Competition: Results and Analysis. *J. Artif. Intell. Res. (JAIR)* **20**, 1–59 (2003).

[35] Cooper, M. C., Maris, F. & Régnier, P. Managing Temporal Cycles in Planning Problems Requiring Concurrency. *Computational Intelligence* **29**, 111–128 (2013). URL `https://doi.org/10.1111/j.1467-8640.2012.00430.x`.

[36] Long, D. & Fox, M. 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 (2003). URL `http://www.aaai.org/Library/ICAPS/2003/icaps03-006.php`.

[37] Rintanen, J. 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 (2007). URL http://www.aaai.org/Library/ICAPS/2007/icaps07-036.php.

[38] Benton, J., Coles, A. J. & Coles, A. 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* (2012). URL http://www.aaai.org/ocs/index.php/ICAPS/ICAPS12/paper/view/4699.

[39] Marzal, E., Sebastia, L. & Onaindia, E. 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* (2014). URL http://www.aaai.org/ocs/index.php/ICAPS/ICAPS14/paper/view/7887.

[40] Karpas, E., Wang, D., Williams, B. C. & Haslum, P. 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 (2015). URL http://www.aaai.org/ocs/index.php/ICAPS/ICAPS15/paper/view/10605.

[41] Rintanen, J. 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 (2015). URL http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9428.

[42] Rintanen, J. 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 (2015). URL http://ijcai.org/Abstract/15/237.

[43] Hönig, W. *et al.* Multi-Agent Path Finding with Kinematic Constraints. In *Proceedings of the Twenty-Sixth International Conference on Automated Plan-*

ning and Scheduling, ICAPS 2016, London, UK, June 12-17, 2016., 477–485 (2016). URL http://www.aaai.org/ocs/index.php/ICAPS/ICAPS16/paper/view/13183.

[44] Ma, H., Tovey, C. A., Sharon, G., Kumar, T. K. S. & Koenig, S. Multi-Agent Path Finding with Payload Transfers and the Package-Exchange Robot-Routing Problem. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA.*, 3166–3173 (2016). URL http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/12437.

[45] Scala, E., Ramírez, M., Haslum, P. & Thiébaux, S. Numeric Planning with Disjunctive Global Constraints via SMT. In *Proceedings of the Twenty-Sixth International Conference on Automated Planning and Scheduling, ICAPS 2016, London, UK, June 12-17, 2016.*, 276–284 (2016). URL http://www.aaai.org/ocs/index.php/ICAPS/ICAPS16/paper/view/13177.

Appendix A

The TableMover Domain

The domain definition using Kovacs (2012) [13] notation:

```
(define (domain tablemover)
    (:requirements :equality :typing :conditional-effects :multi-agent)
    (:types agent block table - locatable
            locatable room side)
    (:constants Table - table)
    (:predicates
       (on-table ?b - block)
       (on-floor ?b - block)
       (down ?s - side)
       (up ?s - side)
       (clear ?s - side)
       (at-side ?a - agent ?s - side)
       (lifting ?a - agent ?s - side)
       (inroom ?l - locatable ?r - room)
       (available ?a - agent)
       (handempty ?a - agent)
       (holding ?a - agent ?b - block)
       (connected ?r1 ?r2 - room)
    )
    (:action pickup-floor
        :agent ?a - agent
        :parameters (?b - block ?r - room)
        :precondition (and
                        (on-floor ?b)
                        (inroom ?a ?r)
                        (inroom ?b ?r)
                        (available ?a)
                        (handempty ?a)
                        (forall (?a2 - agent)
                               (not (pickup-floor ?a2 ?b ?r))
                        )
                    )
```

```
    :effect (and
                (not (on-floor ?b))
                (not (inroom ?b ?r))
                (not (handempty ?a))
                (holding ?a ?b)
            )
)
(:action putdown-floor
    :agent ?a - agent
    :parameters (?b - block ?r - room)
    :precondition (and
                    (available ?a)
                    (inroom ?a ?r)
                    (holding ?a ?b)
                 )
    :effect (and
                (on-floor ?b)
                (inroom ?b ?r)
                (handempty ?a)
                (not (holding ?a ?b))
            )
)
(:action pickup-table
    :agent ?a - agent
    :parameters (?b - block ?r - room)
    :precondition (and
                    (on-table ?b)
                    (inroom ?a ?r)
                    (inroom Table ?r)
                    (available ?a)
                    (handempty ?a)
                    (forall (?a2 - agent)
                           (not (pickup-table ?a2 ?b ?r))
                    )
                 )
    :effect (and
                (not (on-table ?b))
                (not (handempty ?a))
                (holding ?a ?b)
            )
)
(:action putdown-table
    :agent ?a - agent
    :parameters (?b - block ?r - room)
    :precondition (and
                    (inroom ?a ?r)
                    (inroom Table ?r)
                    (available ?a)
                    (holding ?a ?b)
                    ; check table not lifted
                    (forall (?s - side) (down ?s))
                    ; check table not intended to be lifted!
                    (forall (?a2 - agent ?s - side)
```

```
                                (not (lift-side ?a2 ?s))
                        )
                )
    :effect (and
                (on-table ?b)
                (handempty ?a)
                (not (holding ?a ?b))
            )
)
(:action to-table
    :agent ?a - agent
    :parameters (?r - room ?s - side)
    :precondition (and
                        (clear ?s)
                        (inroom ?a ?r)
                        (inroom Table ?r)
                        (available ?a)
                        (forall (?a2 - agent)
                                (not (to-table ?a2 ?r ?s))
                        )
                )
    :effect (and
                (not (clear ?s))
                (at-side ?a ?s)
                (not (available ?a))
            )
)
(:action leave-table
    :agent ?a - agent
    :parameters (?s - side)
    :precondition (and
                        (at-side ?a ?s)
                        (not (lifting ?a ?s))
                )
    :effect (and
                (clear ?s)
                (not (at-side ?a ?s))
                (available ?a)
            )
)
(:action move-agent
    :agent ?a - agent
    :parameters (?r1 ?r2 - room)
    :precondition (and
                        (inroom ?a ?r1)
                        (connected ?r1 ?r2)
                )
    :effect (and
                (not (inroom ?a ?r1))
                (inroom ?a ?r2)
            )
)
(:action move-table
```

```
    :agent ?a - agent
    :parameters (?r1 ?r2 - room ?s - side)
    :precondition (and
                    (lifting ?a ?s)
                    (inroom ?a ?r1)
                    (connected ?r1 ?r2)
                    (exists (?a2 - agent ?s2 - side)
                           (and
                                (not (= ?s ?s2))
                                (move-table ?a2 ?r1 ?r2 ?s2)
                           )
                    )
                 )
    :effect (and
                (not (inroom ?a ?r1))
                (not (inroom Table ?r1))
                (inroom ?a ?r2)
                (inroom Table ?r2)
            )
)
(:action lift-side
    :agent ?a - agent
    :parameters (?s - side)
    :precondition (and
                    (down ?s)
                    (at-side ?a ?s)
                    (handempty ?a)
                    (forall (?a2 - agent ?s2 - side)
                           (not (lower-side ?a2 ?s2))
                    )
                 )
    :effect (and
                (not (down ?s))
                (up ?s)
                (lifting ?a ?s)
                (not (handempty ?a))
                (forall (?b - block ?r - room ?s2 - side)
                       (when
                            (and
                                (inroom Table ?r)
                                (on-table ?b)
                                (down ?s2)
                                (forall (?a2 - agent)
                                       (not (lift-side ?a2 ?s2))
                                )
                            )
                            (and
                                (on-floor ?b)
                                (inroom ?b ?r)
                                (not (on-table ?b))
                            )
                       )
                )
```

```
            )
    )
    (:action lower-side
        :agent ?a - agent
        :parameters (?s - side)
        :precondition (and
                        (lifting ?a ?s)
                        (forall (?a2 - agent ?s2 - side)
                               (not (lift-side ?a2 ?s2))
                        )
                    )
        :effect (and
                    (down ?s)
                    (not (up ?s))
                    (not (lifting ?a ?s))
                    (handempty ?a)
                    (forall (?b - block ?r - room ?s2 - side)
                           (when
                                (and
                                    (inroom Table ?r)
                                    (on-table ?b)
                                    (up ?s2)
                                    (forall (?a2 - agent)
                                           (not (lower-side ?a2 ?s2))
                                    )
                                )
                                (and
                                    (on-floor ?b)
                                    (inroom ?b ?r)
                                    (not (on-table ?b))
                                )
                           )
                    )
            )
    )
)
```

An example problem instance:

```
(define (problem table1_1) (:domain tablemover)
(:objects
    a1 a2 - agent
    b1 - block
    r1x1 r1x2 - room
    left1 right1 - side1
)
(:init
    (on-floor b1)
    (down left1)
    (down right1)
    (clear left1)
    (clear right1)
    (inroom a1 r1x1)
```

```
    (inroom a2 r1x1)
    (inroom b1 r1x1)
    (inroom Table1 r1x1)
    (available a1)
    (available a2)
    (handempty a1)
    (handempty a2)
    (connected r1x1 r1x2)
)
(:goal (and
           (on-floor b1)
           (down left1)
           (down right1)
           (inroom b1 r1x2)
       )
))
```

Appendix B

The Maze Domain

The domain definition using Crosby *et al.* (2014) notation:

```
(define (domain maze)
    (:requirements :typing :concurrency-network :multi-agent)
    (:types agent location door bridge boat switch)
    (:predicates
        (at ?a - agent ?x - location)
        (has-switch ?s - switch ?x - location ?y - location ?z - location)
        (blocked ?x - location ?y - location)
        (has-door ?d - door ?x - location ?y - location)
        (has-boat ?b - boat ?x - location ?y - location)
        (has-bridge ?b - bridge ?x - location ?y - location)
    )
    (:action move
        :agent ?a - agent
        :parameters (?d - door ?x - location ?y - location)
        :precondition (and
                        (at ?a ?x)
                        (not (blocked ?x ?y))
                        (has-door ?d ?x ?y)
                    )
        :effect (and
                    (at ?a ?y)
                    (not (at ?a ?x))
                )
    )
    (:action row
        :agent ?a - agent
        :parameters (?b - boat ?x - location ?y - location)
        :precondition (and
                        (at ?a ?x)
                        (has-boat ?b ?x ?y)
                    )
        :effect (and
```

```
                (at ?a ?y)
                (not (at ?a ?x))
            )
    )
    (:action cross
        :agent ?a - agent
        :parameters (?b - bridge ?x - location ?y - location)
        :precondition (and
                        (at ?a ?x)
                        (has-bridge ?b ?x ?y)
                 )
        :effect (and
                (at ?a ?y)
                (not (at ?a ?x))
                (not (has-bridge ?b ?x ?y))
                (not (has-bridge ?b ?y ?x))
            )
    )
    (:action pushswitch
        :agent ?a - agent
        :parameters (?s - switch ?x - location ?y - location ?z - location)
        :precondition (and
                        (at ?a ?x)
                        (has-switch ?s ?x ?y ?z)
                 )
        :effect (and
                (not (blocked ?y ?z))
                (not (blocked ?z ?y))
            )
    )
    (:concurrency-constraint v1
        :parameters (?d - door)
        :bounds (1 1)
        :actions ( (move 1) )
    )
    (:concurrency-constraint v2
        :parameters (?b - boat ?x - location)
        :bounds (2 inf)
        :actions ( (row 1 2) )
    )
    (:concurrency-constraint v3
        :parameters (?b - bridge)
        :bounds (1 inf)
        :actions ( (cross 1) )
    )
    (:concurrency-constraint v4
        :parameters (?s - switch)
        :bounds (1 1)
        :actions ( (pushswitch 1) )
    )
)
```

An example problem instance:

```
(define (problem maze5_4_1) (:domain maze)
(:objects
       a1 a2 a3 a4 a5 - agent
       loc1x1 loc1x2 loc1x3 loc1x4 loc2x1 loc2x2 loc2x3 loc2x4 loc3x1
          loc3x2 loc3x3 loc3x4 loc4x1 loc4x2 loc4x3 loc4x4 - location
       d1 d2 d3 d4 d5 d6 d7 d8 d9 d10 d11 d12 d13 d14 d15 - door
       b1 b2 b3 - bridge
       bt1 bt2 bt3 bt4 bt5 bt6 - boat
       s1 s2 s3 s4 s5 s6 - switch
)
(:init
       (at a1 loc2x3)
       (at a2 loc3x3)
       (at a3 loc4x3)
       (at a4 loc1x4)
       (at a5 loc1x4)
       (has-door d1 loc1x1 loc1x2)
       (has-door d1 loc1x2 loc1x1)
       (blocked loc1x1 loc1x2)
       (blocked loc1x2 loc1x1)
       (has-switch s1 loc4x4 loc1x1 loc1x2)
       (has-door d2 loc1x1 loc2x1)
       (has-door d2 loc2x1 loc1x1)
       (blocked loc1x1 loc2x1)
       (blocked loc2x1 loc1x1)
       (has-switch s2 loc3x2 loc1x1 loc2x1)
       (has-door d3 loc1x2 loc1x3)
       (has-door d3 loc1x3 loc1x2)
       (has-boat bt1 loc1x2 loc2x2)
       (has-boat bt1 loc2x2 loc1x2)
       (has-door d4 loc1x3 loc1x4)
       (has-door d4 loc1x4 loc1x3)
       (has-door d5 loc1x3 loc2x3)
       (has-door d5 loc2x3 loc1x3)
       (blocked loc1x3 loc2x3)
       (blocked loc2x3 loc1x3)
       (has-switch s3 loc3x3 loc1x3 loc2x3)
       (has-door d6 loc1x4 loc2x4)
       (has-door d6 loc2x4 loc1x4)
       (blocked loc1x4 loc2x4)
       (blocked loc2x4 loc1x4)
       (has-switch s4 loc2x3 loc1x4 loc2x4)
       (has-bridge b1 loc2x1 loc2x2)
       (has-bridge b1 loc2x2 loc2x1)
       (has-door d7 loc2x1 loc3x1)
       (has-door d7 loc3x1 loc2x1)
       (blocked loc2x1 loc3x1)
       (blocked loc3x1 loc2x1)
       (has-switch s5 loc1x3 loc2x1 loc3x1)
       (has-boat bt2 loc2x2 loc2x3)
       (has-boat bt2 loc2x3 loc2x2)
       (has-door d8 loc2x2 loc3x2)
```

```
        (has-door d8 loc3x2 loc2x2)
        (has-door d9 loc2x3 loc2x4)
        (has-door d9 loc2x4 loc2x3)
        (blocked loc2x3 loc2x4)
        (blocked loc2x4 loc2x3)
        (has-switch s6 loc4x2 loc2x3 loc2x4)
        (has-door d10 loc2x3 loc3x3)
        (has-door d10 loc3x3 loc2x3)
        (has-bridge b2 loc2x4 loc3x4)
        (has-bridge b2 loc3x4 loc2x4)
        (has-door d11 loc3x1 loc3x2)
        (has-door d11 loc3x2 loc3x1)
        (has-boat bt3 loc3x1 loc4x1)
        (has-boat bt3 loc4x1 loc3x1)
        (has-door d12 loc3x2 loc3x3)
        (has-door d12 loc3x3 loc3x2)
        (has-bridge b3 loc3x2 loc4x2)
        (has-bridge b3 loc4x2 loc3x2)
        (has-door d13 loc3x3 loc3x4)
        (has-door d13 loc3x4 loc3x3)
        (has-boat bt4 loc3x3 loc4x3)
        (has-boat bt4 loc4x3 loc3x3)
        (has-door d14 loc3x4 loc4x4)
        (has-door d14 loc4x4 loc3x4)
        (has-boat bt5 loc4x1 loc4x2)
        (has-boat bt5 loc4x2 loc4x1)
        (has-boat bt6 loc4x2 loc4x3)
        (has-boat bt6 loc4x3 loc4x2)
        (has-door d15 loc4x3 loc4x4)
        (has-door d15 loc4x4 loc4x3)
)
(:goal (and
        (at a1 loc2x4)
        (at a2 loc1x2)
        (at a3 loc3x1)
        (at a4 loc1x4)
        (at a5 loc3x3)
))
)
```

Appendix C

The AIA domain

The domain definition:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Captures possible relations between 2 intervals regarding Allen's
    algebra

(define (domain allen-algebra)
 (:requirements :typing :durative-actions)
 (:types interval)
 (:predicates
    (started ?i - interval)
    (ended ?i - interval)
    (not-started ?i - interval) ;; duplicates used to preserve positive
       preconditions
    (not-ended ?i - interval) ;;

    (before ?i1 ?i2 - interval) ;; desired goal conditions
    (meets ?i1 ?i2 - interval)
    (overlaps ?i1 ?i2 - interval)
    (starts ?i1 ?i2 - interval)
    (during ?i1 ?i2 - interval)
    (finishes ?i1 ?i2 - interval)
    (equal ?i1 ?i2 - interval)
 )
 (:functions
    (length ?i - interval)
 )

 ;;; Apply an interval
 (:durative-action apply-interval
  :parameters (?i - interval)
  :duration (= ?duration (length ?i))
  :condition
```

```
    (and
       (at start (not-started ?i))
    )
  :effect
    (and
       (at start (started ?i))
       (at start (not (not-started ?i)))
       (at end (ended ?i))
       (at end (not (not-ended ?i)))
    )
 )
)
```

An example problem instance:

```
(define (problem aa-overlaps-4)
 (:domain allen-algebra)
 (:objects i1 i2 i3 i4 - interval)
 (:init
    (not-started i1)
    (not-ended i1)
    (not-started i2)
    (not-ended i2)
    (not-started i3)
    (not-ended i3)
    (not-started i4)
    (not-ended i4)
    (= (length i1) 5)
    (= (length i2) 5)
    (= (length i3) 5)
    (= (length i4) 5)
 )
 (:goal
    (and
       (overlaps i1 i2)
       (overlaps i2 i3)
       (overlaps i3 i4)
    )
 )
)
```

Appendix D

The SmartMobility domain

The domain definition:

```
(define (domain journey-planner)
 (:requirements :typing :durative-actions)
 (:types
    bus taxi car bike carpool flexibus - vehicle
    pedestrian vehicle - agent
    agent location
 )
 (:predicates
    (at ?a - agent ?x - location)
    (in ?p - pedestrian ?v - vehicle)
    (has-footpath ?p - pedestrian ?x - location ?y - location)
    (has-street ?x - location ?y - location)
 )
 (:functions
    (velocity ?a - agent ?x - location ?y - location) - number
    (distance ?x - location ?y - location) - number
 )
 (:durative-action embark
    :parameters (?p - pedestrian ?v - vehicle ?x - location)
    :duration (= ?duration 1)
    :condition (and
                  (at start (at ?p ?x))
                  (over all (at ?v ?x))
              )
    :effect (and
                (at end (in ?p ?v))
                (at start (not (at ?p ?x)))
            )
 )
 (:durative-action debark
    :parameters (?p - pedestrian ?v - vehicle ?x - location)
    :duration (= ?duration 1)
```

```
    :condition (and
                   (at start (in ?p ?v))
                   (over all (at ?v ?x))
               )
    :effect (and
                (at end (at ?p ?x))
                (at start (not (in ?p ?v)))
            )
)
(:durative-action walk
    :parameters (?p - pedestrian ?x - location ?y - location)
    :duration (= ?duration (/ (distance ?x ?y) (velocity ?p ?x ?y)))
    :condition (and
                   (at start (at ?p ?x))
                   (over all (has-footpath ?p ?x ?y))
               )
    :effect (and
                (at end (at ?p ?y))
                (at start (not (at ?p ?x)))
            )
)
(:durative-action travel
    :parameters (?v - vehicle ?x - location ?y - location)
    :duration (= ?duration (/ (distance ?x ?y) (velocity ?v ?x ?y)))
    :condition (and
                   (at start (at ?v ?x))
                   (over all (has-street ?x ?y))
               )
    :effect (and
                (at end (at ?v ?y))
                (at start (not (at ?v ?x)))
            )
)
)
```

An example problem instance:

```
(define (problem p01)
 (:domain journey-planner)
 (:objects
    c1 - carpool
    p1 p2 - pedestrian
    loc1 loc2 loc3 loc4 loc5 - location
 )
 (:init
    (at c1 loc1)
    (at p1 loc2)
    (at p2 loc3)

    (has-footpath p1 loc2 loc3)
    (has-footpath p1 loc3 loc2)

    (has-footpath p2 loc2 loc3)
```

```
    (has-footpath p2 loc3 loc2)

    (has-street loc1 loc2)
    (has-street loc2 loc1)

    (has-street loc2 loc5)
    (has-street loc5 loc2)

    (has-street loc4 loc5)
    (has-street loc5 loc4)

    (has-street loc1 loc4)
    (has-street loc4 loc1)

    (= (distance loc1 loc2) 4)
    (= (distance loc2 loc1) 4)

    (= (distance loc2 loc3) 1)
    (= (distance loc3 loc2) 1)

    (= (distance loc2 loc5) 4)
    (= (distance loc5 loc2) 4)

    (= (distance loc4 loc5) 4)
    (= (distance loc5 loc4) 4)

    (= (distance loc1 loc4) 4)
    (= (distance loc4 loc1) 4)

    (= (velocity p1 loc2 loc3) 1)
    (= (velocity p1 loc3 loc2) 1)

    (= (velocity p2 loc2 loc3) 1)
    (= (velocity p2 loc3 loc2) 1)

    (= (velocity c1 loc1 loc2) 2)
    (= (velocity c1 loc2 loc1) 2)

    (= (velocity c1 loc2 loc5) 2)
    (= (velocity c1 loc5 loc2) 2)

    (= (velocity c1 loc4 loc5) 2)
    (= (velocity c1 loc5 loc4) 2)

    (= (velocity c1 loc1 loc4) 2)
    (= (velocity c1 loc4 loc1) 2)
 )
 (:goal (and
           (at c1 loc4)
           (at p1 loc5)
           (at p2 loc5)
 ))
)
```