Solving Concurrent Multiagent Planning using Classical Planning

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

Abstract

In this work we present a novel approach to solving concurrent multiagent planning problems in which several agents act in parallel. Our approach relies on a compilation from concurrent multiagent planning to classical planning, allowing us to use an off-the-shelf classical planner to solve the original multiagent problem. The solution can be directly interpreted as a concurrent plan that satisfies a given set of concurrency constraints, while avoiding the exponential blowup associated with concurrent actions. Theoretically, we show that the compilation is sound and complete. Empirically, we show that our compilation can solve challenging multiagent planning problems that require concurrent actions.

Introduction

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-15 (Komenda, Stolba, and Kovacs 2016). Many recent multiagent planners are based on the MA-STRIPS formalism (Brafman and Domshlak 2008), and can be loosely classified into one of two categories: centralized, in which agents have full information and share the goal, and distributed, in which agents have partial information and individual goals. In CoDMAP-15, the most successful centralized planners were ADP (Crosby, Rovatsos, and Petrick 2013), MAP-LAPKT (Muise, Lipovetzky, and Ramirez 2015) and CMAP (Borrajo 2013), while prominent distributed planners included PSM (Tozicka, Jakubuv, and Komenda 2014), MAPlan (Stolba, Fiser, and Komenda 2016) and MH-FMAP (Torreño, Onaindia, and Sapena 2014).

Although establishing a common set of multiagent benchmark domains was a major step forward, the domains from the centralized track of CoDMAP-15 can all be solved using *sequential* plans in which one agent acts at a time. In contrast, there are many applications that require agents to act in *parallel*; examples include shared journey planning, robot coordination tasks such as RoboCupSoccer (Nardi et al. 2014) and RoboCupRescue (Sheh, Schwertfeger, and Visser 2016), and real-time strategy games, to name a few.

In this paper we consider the problem of *concurrent* centralized multi-agent planning in which agents can act in parallel 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 restrictions are needed to ensure that concurrent actions are well-formed. Usually, these restrictions take the form of *concurrency constraints* (Boutilier and Brafman 2001; Crosby 2013), which can model both the case when two actions *must* occur in parallel, and the case when two actions *cannot* occur in parallel.

Although some planners from CoDMAP-15 (CMAP, MAPlan and MH-FMAP) can produce concurrent plans, there are few that can reliably handle more complex concurrency constraints. A notable exception is the work of Crosby, Jonsson, and Rovatsos (2014), 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 a classical planner.

Brafman and Zoran (2014) extended the distributed forward-search planner MAFS (Nissim and Brafman 2014) to support concurrency constraints while preserving privacy. In MAFS, each agent maintains its own search space, and has a queue for expanded states (closed list) and another for states to be expanded (open list). When a state s is expanded, the agent uses its own operators only; thus, two agents expanding the same state will generate different successors. Messages are exchanged between agents in order to inform each other about the expansion of relevant states. Consequently, agents explore the search space together while preserving privacy. Maliah, Brafman, and Shani (2017) proposed MAFBS, which extended MAFS to use forward and backward messages. This new approach reduced the number of messages required and also resulted in an increase in the privacy of agents.

In this paper we describe a planner that can handle arbitrary concurrency constraints, as originally proposed by Boutilier and Brafman (2001). Our approach is similar to that of Crosby, Jonsson, and Rovatsos (2014) in that we transform a multiagent planning problem into a single-agent problem that is significantly easier to solve, while avoiding the exponential blowup associated with concurrent actions. However, the concurrency constraints of Boutilier and Brafman are significantly more expressive than those of Crosby, Jonsson, and Rovatsos, enabling us to solve multiagent problems with more complex interactions (e.g. effects that depend on the concurrent actions of other agents). We show

that our planner is sound and complete, and perform experiments in several concurrent multiagent planning domains to evaluate its performance.

The remainder of this paper is structured as follows. We first introduce the different planning formalisms that are needed to describe our planner. Next, we describe the compilation from multiagent planning to single-agent planning that our planner employs. We then present the results of experiments with our planner in several domains that require agents to act in parallel. Finally, we relate our planner to existing work in the literature, and conclude with a discussion.

Background

In this section we describe the planning formalisms that we use in the paper: classical planning, concurrent planning, and concurrent multiagent planning.

Classical Planning

We consider the fragment of classical planning with conditional effects and negative conditions and goals. Given a set of fluents F, a *literal* l is a valuation of a fluent in F, where $l = f$ denotes that l assigns true to $f \in F$, and $l = \neg f$ that l assigns false to f. A set of literals L is *well-defined* if it does not assign conflicting values to any fluent, i.e. if L does not contain both f and $\neg f$ for some fluent $f \in F$. Let $\mathcal{L}(F)$ be the set of well-defined literal sets on F, i.e. the set of all partial assignments of values to fluents. Given a literal set $L \in \mathcal{L}(F)$, let $\neg L = \{\neg l : l \in L\}$ be the *complement* of L. We also define the *projection* $L_{|X}$ of a literal set L onto a subset of fluents $X \subseteq F$.

A *state* $s \in \mathcal{L}(F)$ is a well-defined literal set such that $|s| = |F|$, i.e. a total assignment of values to fluents. Explicitly including negative literals $\neg f$ in states simplifies subsequent definitions, but we often abuse notation by defining a state s only in terms of the fluents that are true in s, as is common in classical planning.

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, $I \in \mathcal{L}(F)$ is an initial state, and $G \in \mathcal{L}(F)$ is a goal condition (usually satisfied by multiple states). Each action $a \in A$ has a precondition $\mathsf{pre}(a) \in \mathcal{L}(F)$ and a set of conditional effects $\mathsf{cond}(a)$. Each conditional effect $C \rhd E \in \mathsf{cond}(a)$ is composed of two literal sets $C \in \mathcal{L}(F)$ (the condition) and $E \in \mathcal{L}(F)$ (the effect).

An action $a \in A$ is applicable in state s if and only if $\mathsf{pre}(a) \subseteq s$, and the resulting (triggered) *effect* is given by

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

i.e. effects whose conditions hold in s. We assume that $\mathsf{eff}(s, a)$ is a well-defined literal set in $\mathcal{L}(F)$ for each state-action pair (s, a). The result of applying a in s is a new state $\theta(s, a) = (s \setminus \neg\mathsf{eff}(s, a)) \cup \mathsf{eff}(s, a)$. It is straightforward to show that if s and $\mathsf{eff}(s, a)$ are in $\mathcal{L}(F)$, then so is $\theta(s, a)$.

Given a planning problem Π, a *plan* is an action sequence $\pi = \langle a_1, \ldots, a_n\rangle$ that induces a state sequence $\langle s_0, s_1, \ldots, s_n\rangle$ such that $s_0 = I$ and, for each i such that $1 \leq i \leq n$, action a_i is applicable in s_{i-1} and generates the successor state $s_i = \theta(s_{i-1}, a_i)$. The plan π *solves* Π if and only if $G \subseteq s_n$, i.e. if the goal condition is satisfied following the application of π in I.

Concurrent Planning

Concurrent planning is the extension of classical planning that allows actions to be applied in parallel, forming *concurrent* or *joint* actions. Given a classical planning problem $\Pi = \langle F, A, I, G\rangle$, an unconstrained concurrent planning problem is given by $\Pi_{conc} = \langle F, 2^A, I, G\rangle$, where the action set 2^A is given by the power set of the original action set A. The aim is to find a *concurrent plan* π_{conc} that solves Π_{conc} by applying a sequence of concurrent actions in 2^A.

To ensure that joint actions have well-defined effects, researchers often impose *concurrency constraints* that model whether two atomic actions *must* or *cannot* be done concurrently. For example, in PDDL 2.1 (Fox and Long 2003), two actions a^1 and a^2 cannot be applied concurrently if a^1 has an effect on a fluent f and a^2 has a precondition or effect on f. Crosby (2013) defines concurrency constraints on actions that have the same *object* in their preconditions or effects.

We adopt a formulation of concurrency constraints due to Boutilier and Brafman (2001), later extended by Kovacs (2012). The idea is to extend preconditions and conditional effects with *actions* in addition to fluents. We overload notation and let each $a \in A$ denote a propositional variable. If a^1 has a precondition a^2, then a^1 is only applicable if a^2 is concurrent with a^1. If a^1 has a precondition $\neg a^2$, then a^1 is only applicable if a^2 is *not* concurrent with a^1.

We extend the notation for classical planning as follows. By viewing A as a set of propositional variables, we can model a subset of atomic actions in 2^A as a well-defined literal set $b \in \mathcal{L}(A)$ such that $|b| = |A|$, analogous to how states are formed from fluents. The subset includes those actions that appear as positive literals in b, while negative literals denote those actions that are *not* part of the subset.

For a given atomic action $a \in A$, the precondition $\mathsf{pre}(a) \in \mathcal{L}(F \cup A)$ and any condition $C \in \mathcal{L}(F \cup A)$ of a conditional effect $C \rhd E \in \mathsf{cond}(a)$ are extended to include actions in addition to fluents. Each effect $E \in \mathcal{L}(F)$ of a conditional effect $C \rhd E \in \mathsf{cond}(a)$ is exclusively on fluents as before. Given a state s and a set of actions that are concurrent with a, represented by a literal set $b \in \mathcal{L}(A)$, a is applicable if and only if $\mathsf{pre}(a) \subseteq s \cup b$, and the resulting effect is given by

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

i.e. effects whose conditions hold in $s \cup b$.

We can now define the semantics of a joint action $b \in 2^A$. Concretely, b satisfies the concurrency constraints if and only if $\mathsf{pre}(a)_{|A} \subseteq b \setminus \{a\}$ for each $a \in b$. If b is applicable, its precondition and effect are defined as the union of the preconditions and effects of the constituent atomic actions:

$$\mathsf{pre}(b) = \bigcup_{a \in b} \mathsf{pre}(a)_{|F},$$

$$\mathsf{eff}(s, b) = \bigcup_{a \in b} \mathsf{eff}(s \cup b \setminus \{a\}, a), \quad \forall s.$$

As before, we assume that $\mathsf{eff}(s, b)$ is a well-defined literal set in $\mathcal{L}(F)$ for each pair of state s and applicable joint action b.

Concurrent Multiagent Planning

In concurrent multiagent planning, each atomic action belongs to an *agent*. We consider the problem of *centralized* multiagent planning in which agents share the goal.

A *concurrent multiagent planning problem* (CMAP) is a tuple $\Pi = \langle N, F, \{A^i\}_{i=1}^n, I, G\rangle$, where $N = \{1, \ldots, n\}$ is a set of agents and A^i is the set of atomic actions of agent $i \in N$. This is identical to the standard definition of multiagent planning problems (Brafman and Domshlak 2008), with the only difference being that the actions include concurrency constraints as defined in the previous section. The fluent set F, initial state I and goal condition G are defined as before.

A CMAP implicitly defines a negative concurrency constraint on each pair of atomic actions $(a^1, a^2) \subseteq A^i$ that belong to the same agent i. Consequently, each agent can contribute at most one atomic action to each joint action. These concurrency constraints are not included in action definitions.

To illustrate CMAPs, we use the TABLEMOVER domain (Boutilier and Brafman 2001), in which two agents move blocks between rooms. There are two possible strategies:

1. Pick up blocks and carry them using their arms.
2. Put blocks on a table, carry the table together to another room, and tip the table to make the blocks fall down.

Figure 1 shows the definition of the lift-side action in the notation of Kovacs (2012), which is used by agent ?a to lift side ?s of the table. The precondition is that the side must be down (i.e. on the floor) and the agent cannot be holding anything. Moreover, the precondition also states that no other agent ?a2 can lower side ?s2 at the same time. When the action is applied, ?s is no longer down but up, and ?a is busy lifting ?s. The action also has a conditional effect (represented by the when clause): if some side ?s2 is not lifted by any agent ?a2, then all blocks on the table fall to the floor.

Note that the action lift-side is defined using forall quantifiers. In practice, such quantifiers are compiled away, such that the resulting actions have quantifier-free preconditions and effects, as in our definition of actions.

```
(: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)) (lifting ?a ?s)
        (up ?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 action lift-side using the notation of Kovacs (2012) (concurrency constraints in bold).

Compilations for CMAPs

Let $\Pi = \langle N, F, \{A^i\}_{i=1}^n, I, G\rangle$ be a CMAP, and let $A = A^1 \cup \cdots \cup A^n$ be the set of atomic actions of Π. A straightforward approach to solving Π is to define a concurrent planning problem $\Pi_{conc} = \langle F, B, I, G\rangle$ and apply a classical planner to solve Π_{conc}. If $B \subseteq 2^A$ is exactly the subset of joint actions that satisfy the concurrency constraints of the actions in Π, this approach is both sound and complete.

However, even though the joint action set B might be much smaller than 2^A, it is still worst-case exponential in the number of agents. Most classical planners ground the actions, and if B is too large, they often do not make it past preprocessing. Moreover, to generate the set B we would typically have to iterate over *all* joint actions, and test whether each action satisfies the concurrency constraints of atomic actions.

Here we describe an alternative approach to solving a CMAP Π. The idea is to model each joint action $b = \{a^1, \ldots, a^k\}$ using multiple atomic actions: one set of actions for *selecting* $a^1, \ldots, a^k$, one set of actions for *applying* $a^1, \ldots, a^k$, and one set of actions for *resetting* $a^1, \ldots, a^k$. The result is a classical planning problem $\Pi' = \langle F', A', I', G'\rangle$ such that the size of the action set A' is *linear* in $|A|$, the number of atomic actions of agents.

Simulating a joint action b using a sequence of atomic actions $\langle a^1, \ldots, a^k\rangle$ is problematic for the following reason: when applying an atomic action a^i, we may not yet know which atomic actions will be applied by other agents. Since those other actions may be part of the precondition and conditional effects of a^i, it becomes difficult to ensure that the concurrency constraints of a^i are correctly enforced.

Our approach is to divide the simulation of a joint action b into three phases: selection, application, and reset. In the selection phase, we use an auxiliary fluent active-a^i to model that the atomic action a^i has been selected. In the application phase, since the selection of atomic actions is known, we can substitute each action a^i in preconditions and conditional effects with the auxiliary fluent active-a^i. In the reset phase, various auxiliary fluents are reset.

Note that this compilation takes into account the multiagent nature of the problem. Each agent can apply at most one atomic action per time step, and agents collaborate to form joint actions whose constituent atomic actions are compatible and/or inapplicable on their own.

We proceed to define the components of the compilation.

Fluents

We describe the fluents in PDDL format, i.e. each fluent is instantiated by assigning objects to predicates.

The set of fluents $F' \supseteq F$ includes all original fluents in F, plus the following auxiliary fluents:

- Fluents free, select, apply and reset modeling the phase.
- For each agent i, fluents free-agent(i), busy-agent(i) and done-agent(i) that model the agent state: free to select an action, selected an action, and applied the action.
- For each action $a^i \in A^i$ in the action set of agent i, a fluent active-a^i which models that a^i has been selected. We use F_{act} to denote the subset of fluents of this type.

By simple inspection, the total number of fluents in F' is given by $|F'| = |F| + 4 + 3n + \sum_{i \in N} |A^i| = O(|F| + |A|)$.

The initial state I' of the compilation Π' is given by

$$I' = I \cup \{\text{free}\} \cup \{\text{free-agent}(i) : i \in N\},$$

i.e. the initial state on fluents in F is I, we are not simulating any joint action, and all agents are free to select actions. The goal condition is given by $G' = G \cup \{\text{free}\}$, i.e. the goal condition G has to hold at the end of a joint action simulation.

Actions

For a literal set $L \in \mathcal{L}(F \cup A)$, let $L_{|A}/F_{act}$ denote the projection of L onto A, followed by a substitution of the actions in A with the corresponding fluents in F_{act}. Note that both $L_{|F}$ and $L_{|A}/F_{act}$ are literal sets on fluents in F', i.e. the dependence on actions in A is removed.

The first four actions in the set A' allow us to switch between simulation phases, and are defined as follows:

select-phase: $\text{pre} = \{\text{free}\}$, $\text{cond} = \{\emptyset \rhd \{\neg\text{free}, \text{select}\}\}$.

apply-phase: $\text{pre} = \{\text{select}\}$, $\text{cond} = \{\emptyset \rhd \{\neg\text{select}, \text{apply}\}\}$.

reset-phase: $\text{pre} = \{\text{apply}\}$, $\text{cond} = \{\emptyset \rhd \{\neg\text{apply}, \text{reset}\}\}$.

finish: $\text{pre} = \{\text{reset}, \text{free-agent}(i) : i \in N\}$, $\text{cond} = \{\emptyset \rhd \{\neg\text{reset}, \text{free}\}\}$.

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

The action select-a^i causes i to select action a^i during the selection phase, and is defined as follows:

$$\text{pre} = \{\text{select}, \text{free-agent}(i)\} \cup \text{pre}(a^i)_{|F},$$
$$\text{cond} = \{\emptyset \rhd \{\text{busy-agent}(i), \neg\text{free-agent}(i), \text{active-}a^i\}\}.$$

The precondition ensures that we are in the selection phase, that i is free to select an action, and that the precondition of a^i holds on fluents in F. The effect prevents i from selecting another action, and marks a^i as selected.

The action do-a^i applies the effect of a^i in the application phase, and is defined as follows:

$$\text{pre} = \{\text{apply}, \text{busy-agent}(i), \text{active-}a^i\} \cup \text{pre}(a^i)_{|A}/F_{act},$$
$$\text{cond} = \{\emptyset \rhd \{\text{done-agent}(i), \neg\text{busy-agent}(i)\}\}$$
$$\cup \{C_{|F} \cup C_{|A}/F_{act} \rhd E : C \rhd E \in \text{cond}(a^i)\}.$$

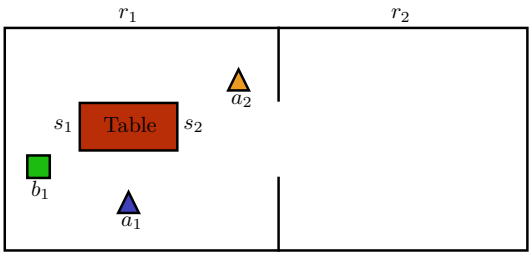

Figure 2: Initial state of a simple TABLEMOVER instance.

The precondition ensures that we are in the application phase, that a^i was previously selected, and that all concurrency constraints in the precondition of a^i hold. The effect is to apply all conditional effects of a^i, where each condition $C_{|F} \cup C_{|A}/F_{act}$ is generated from C by substituting each action $b^j \in A$ with active-b^j. Agent i is also marked as done to prevent a^i from being applied a second time.

The action end-a^i resets auxiliary fluents to their original value, and is defined as follows:

$$\text{pre} = \{\text{reset}, \text{done-agent}(i), \text{active-}a^i\},$$
$$\text{cond} = \{\emptyset \rhd \{\text{free-agent}(i), \neg\text{done-agent}(i), \neg\text{active-}a^i\}\}.$$

The precondition ensures that we are in the reset phase and that a^i was previously selected and applied (due to done-agent(i)). The effect is to make agent i free to select actions again, and to mark a^i as no longer selected.

Again, by inspection we can see that the total number of actions in A' is given by $|A'| = 4 + 3\sum_i |A^i| = O(|A|)$.

Properties

Figure 2 shows an example instance of TABLEMOVER in which the goal is for agents a_1 and a_2 to move block b_1 from room r_1 to room r_2. An example concurrent plan that solves this instance is defined as follows:

```
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)
```

In this plan, agent a_2 first puts the block on the table, and then a_1 and a_2 concurrently lift each side of the table and move the table to room r_2. Finally, a_1 lowers its side of the table, causing the table to tip and the block to fall to the floor.

The following sequence of classical actions in A' can be used to simulate the first joint action of the concurrent plan:

```
1 (select-phase )
2 (select-to-table a1 r1 s2)
3 (select-pickup-floor a2 b1 r1)
4 (apply-phase )
5 (do-pickup-floor a2 b1 r1)
6 (do-to-table a1 r1 s2)
7 (reset-phase )
8 (end-to-table a1 r1 s2)
9 (end-pickup-floor a2 b1 r1)
10 (finish )
```

We show that the compilation is both sound and complete.

Theorem 1 (Soundness)**.** *A classical plan π' that solves Π' can be transformed into a concurrent plan π that solves Π.*

Proof. When fluent free is true, the only applicable action is select-phase. The only way to make free true again is to cycle through the three phases and end with the finish action.

During the selection phase, a subset of actions $a^1, \ldots, a^k$ are selected, causing the corresponding agents to be busy. Because of the precondition $\mathsf{free\text{-}agent}(i)$ of the finish action, each selected action a^i has to be applied in the application phase, and reset in the reset phase. The resulting simulated joint action is given by $b = \{a^1, \ldots, a^k\}$.

The precondition of b holds since the precondition of each a^i on fluents in F is checked in the selection phase, during which no fluents in F change values. The concurrency constraints of a^i are checked in the application phase when all actions have already been selected. This also ensures that the conditional effects of a^i are correctly applied. Finally, auxiliary fluents are cleaned in the reset phase. Hence the joint action b satisfies all concurrency constraints and is correctly simulated by the corresponding action subsequence of π'.

Let π be the concurrent plan composed of the sequence of joint actions simulated by the plan π'. Since π' solves Π', the goal condition G holds at the end of π', implying that G also holds at the end of π. This implies that π solves Π. □

Theorem 2 (Completeness)**.** *A concurrent plan π that solves Π corresponds to a classical plan π' that solves Π'.*

Proof. Let $b = \{a^1, \ldots, a^k\}$ be a joint action of the concurrent plan π. We can use a sequence of actions in A' to simulate b by selecting, applying and resetting each action among $a^1, \ldots, a^k$. Since b is part of π, its precondition and concurrency constraints have to hold, implying that the precondition and concurrency constraints of each atomic action hold. Hence the action sequence is applicable and results in the same effect as b. By concatenating such action sequences for each joint action of π, we obtain a plan π'. Since π solves Π, the goal condition G holds at the end of π, implying that G holds at the end of π'. This implies that π' solves Π'. □

Extensions

The basic compilation checks concurrency constraints in the application phase. Here we describe an extension that checks negative concurrency constraints in the selection phase, allowing a classical planner to identify inadmissible joint actions as early as possible, reducing the branching factor.

Assume that action a^i has a negative concurrency constraint $\neg a^j$. As before, we can simulate this constraint using the fluent $\neg\mathsf{active}\text{-}a^j$. However, a^j may be selected *after* a^i in the selection phase, in which case $\neg\mathsf{active}\text{-}a^j$ holds when selecting a^i. To prevent inadmissible joint actions from being selected, we introduce additional fluents in the set F':

- For each action $a^i \in A^i$ in the action set of agent i, a fluent $\mathsf{req\text{-}neg}\text{-}a^i$ which indicates that a^i cannot be selected.

We now redefine the action $\mathsf{select}\text{-}a^i$ as follows:

$$\begin{aligned}\mathrm{pre} &= \{\mathsf{select}, \mathsf{free\text{-}agent}(i), \neg\mathsf{req\text{-}neg}\text{-}a^i\} \cup \mathsf{pre}(a^i)_{|F} \\ &\quad \cup \{\neg\mathsf{active}\text{-}b^j : \neg b^j \in \mathsf{pre}(a^i)\}, \\ \mathrm{cond} &= \{\emptyset \rhd \{\mathsf{busy\text{-}agent}(i), \neg\mathsf{free\text{-}agent}(i), \mathsf{active}\text{-}a^i\}\} \\ &\quad \cup \{\emptyset \rhd \{\mathsf{req\text{-}neg}\text{-}b^j : \neg b^j \in \mathsf{pre}(a^i)\}\}.\end{aligned}$$

To select a^i, $\mathsf{req\text{-}neg}\text{-}a^i$ has to be false. For each negative concurrency constraint $\neg b^j$ of a^i, action $\mathsf{select}\text{-}a^i$ adds fluent $\mathsf{req\text{-}neg}\text{-}b^j$, preventing b^j from being selected after a^i.

With this extension, we only need to check *positive* concurrency constraints (i.e. required concurrency) in the application phase. We also redefine $\mathsf{end}\text{-}a^i$ such that fluents of type $\mathsf{req\text{-}neg}\text{-}a^i$ are reset in the cleanup phase, using the opposite effect of $\mathsf{select}\text{-}a^i$. The initial state and goal condition do not change since the new fluents are always false while no joint action is simulated.

The second extension is to impose a bound C on the number of atomic actions selected in the selection phase, resulting in a classical planning problem $\Pi'_C = \langle F'_C, A'_C, I'_C, G'_C\rangle$. The fluent set $F'_C \supseteq F'$ extends F' with fluents $\mathsf{count}(j)$, $0 \leq j \leq C$. Counter parameters are added to the select and reset actions so that they can respectively increment and decrement the value of the counter. Crucially, no select action is applicable when $j = C$, preventing us from selecting more than C actions. The benefit is to reduce the branching factor by disallowing joint actions with more than C atomic actions.

We leave the following proposition without proof:

Proposition 3. *The compilation Π'_C that includes both proposed extensions is sound.*

Note that the compilation Π'_C is not complete. For instance, consider a concurrent multiagent plan that contains a joint action involving 4 atomic actions. If $C < 4$, then the concurrent multiagent plan cannot be converted into an equivalent classical plan without exceeding the bound C.

Experimental Results

We tested our compilations in two different sets of domains: centralized multiagent domains from the CoDMAP-15 competition, and four domains (MAZE, TABLEMOVER, WORKSHOP and BOXPUSHING) that require concurrency[1].

In each domain, we used three variants of our compilations: unbounded joint action size, and joint action size bounded by $C = 2$ and $C = 4$. In all variants, we used the extension that identifies negative concurrency constraints in the selection phase. The resulting classical planning problems were then solved using Fast Downward (Helmert 2006) in the LAMA setting (Richter and Westphal 2010).

All experiments ran on Intel Xeon E5-2673 v4 @ 2.3GHz processors. They had a time limit of 30 minutes and a memory limit of 8 GB.

[1]The code of the compilation and the domains are available at https://github.com/aig-upf/universal-pddl-parser-multiagent.

CoDMAP Domains

Although the centralized multiagent benchmark domains of CoDMAP involve multiple agents, none of these domains require concurrency between agents, i.e. their instances can be solved by sequences of atomic actions. Instead, the purpose of CoDMAP was to solve problems involving privacy over predicates, constants and objects.

As our approach and CoDMAP planners differ in their purpose, we decided to compare the former against the classical planner it uses to solve the compilation (Fast Downward). Since CoDMAP domains can be sequentially solved, Fast Downward can be directly applied on them. Furthermore, we will be able to see how the results of solving a problem sequentially compare to the results if the same problem is solved when concurrency is allowed. In addition, the coverage of Fast Downward is almost as high as the winner of the centralized track at CoDMAP-15, ADP (Crosby, Rovatsos, and Petrick 2013), which solved 222 instances.

We compare the results of our compilation with those obtained by running Fast Downward (FD) directly on the given instances. We wanted to test the following hypotheses:

1. Using our compilations it is possible to solve approximately the same number of instances as a classical planner that ignores the multiagent nature of the problem.
2. The plans resulting from solving the compiled problems are implicitly more compressed since atomic actions are grouped into joint actions.

The domains forming this set of experiments were manually modified to add the appropriate negative concurrency constraints. Otherwise, the instances would violate our assumption that the triggered effect $\mathsf{eff}(s, b)$ of a joint action is well-defined, producing invalid plans. For instance, an atomic action adding a fluent f and another action deleting f could have been part of the same joint action. Therefore, note that the complexity of the tasks increases with respect to the original tasks that had no concurrency constraints.

Table 1 shows the results for this set of experiments. There are four different metrics:

- Coverage: number of instances solved.
- Time: average number of seconds taken to find a solution.
- Plan length: number of actions in the plan, corresponding to the number of joint actions for our compilations.
- #Actions: total number of actions instantiated by FD.

From the table, we observe that the planner with the largest coverage is FD (219, 91.25%). The compilation with the largest coverage is the variant whose joint action size is bounded by 2 (158, 65.83%), followed by the unbounded variant (143, 59.58%) and the variant whose bound is set to 4 (140, 58.3%). Thus, although all variants solve less instances, they are not far from the results obtained by FD for many domains. As expected, when concurrency is not required to solve a MAP, the auxiliary fluents and additional copies of actions (corresponding to the three phases of a joint action simulation) introduced by our compilations do not pay off in solution efficiency, resulting in a smaller coverage and a larger time to solve the compiled instances.

Regarding plan length, the compilations always result in shorter solutions than FD. Since FD outputs a sequential plan with no joint actions, it has no way of compressing the plans, unlike the compilations in which atomic actions are grouped into joint actions.

As stated in the previous section, the number of actions of the compilation with unbounded joint action size is approximately $3|A|$, and the number of actions of the compilation with joint action size bounded by C is approximately $(C + 1)|A|$, which is reflected quite well in the number of instantiated actions (results vary somewhat because of the reachability tests performed by FD during grounding).

Domains with Required Concurrency

In this section, we first give a brief description of the domains that were included in the experiments, and the other algorithms that were used for comparison.

The MAZE domain (Crosby 2014) consists of a grid of interconnected locations. Each agent in the maze 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, and are unlocked by pushing a specific switch, placed anywhere in the maze.
- Bridge: can be crossed by multiple agents at once, but is destroyed after the first crossing.
- Boat: can only be used by two or more agents in the same direction.

The WORKSHOP domain is a new domain in which the objective is to perform inventory in a high-security storage facility. It has the following characteristics:

- 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 BOXPUSHING domain (Brafman and Zoran 2014) consists in a grid of interconnected locations. Agents must push the boxes from one location to another. A different number of agents is needed depending on the box size: one for small boxes, two for mediums, and three for large boxes.

The algorithm that we use for comparison is that of Crosby, Jonsson, and Rovatsos (2014) (which we refer to as CJR), who define concurrency constraints in the form of affordances on subsets of objects. For example, the affordance on the subset of objects $\{\text{location}, \text{boat}\}$ 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. Even though their algorithm cannot handle the concurrency constraints of CMAPs, the MAZE and WORKSHOP domains can be reformulated using their concurrency constraints.

The concurrency constraints of CJR are not as expressive as those of Kovacs because:

Domain	N	Coverage				Time (s.)				Plan length				# Actions			
		2	4	∞	FD	2	4	∞	FD	2	4	∞	FD	2	4	∞	FD
BLOCKSWORLD	20	7	2	4	**20**	759.5	-	-	**0.2**	**32.1**	-	-	32.8	6848	12323	4110	**1270**
DEPOT	20	13	10	9	**17**	202.9	246.4	223.9	**58.3**	30.6	15.7	**14.9**	44.0	10100	18176	6061	**2007**
DRIVERLOG	20	18	17	18	**20**	67.3	58.8	73.7	**26.1**	21.1	**20.5**	25.2	35.6	38416	69145	23051	**7386**
ELEVATORS08	20	9	8	10	**20**	13.8	12.5	9.5	**0.2**	31.0	**30.3**	36.4	65.1	10779	19399	6469	**2155**
LOGISTICS00	20	**20**	**20**	**20**	**20**	1.9	2.9	212.1	**0.0**	30.3	**28.0**	30.1	50.2	1781	3202	1070	**318**
ROVERS	20	**20**	**20**	19	**20**	45.2	75.2	20.9	**0.1**	46.5	47.4	**42.9**	56.8	18314	32962	10990	**2609**
SATELLITES	20	19	17	19	**20**	82.8	128.0	32.2	**1.0**	**32.6**	35.5	34.2	55.9	45106	81188	27065	**8122**
SOKOBAN	20	0	0	0	**18**	-	-	-	**32.3**	-	-	-	**54.1**	3319	5970	1993	**663**
TAXI	20	**20**	**20**	**20**	**20**	1.3	2.6	0.7	**0.0**	14.8	**14.7**	**14.7**	18.7	544	975	328	**108**
WIRELESS	20	2	2	2	**4**	-	-	-	-	-	-	-	-	15644	28156	9388	**3128**
WOODWORKING08	20	14	8	4	**20**	290.0	256.0	-	**0.9**	22.4	**11.4**	-	46.1	17406	31327	10445	**3447**
ZENOTRAVEL	20	16	16	18	**20**	87.2	125.6	164.8	**1.5**	**23.6**	24.1	34.2	46.9	67586	121652	40553	**13502**
Total	240	158	140	143	**219**												

Table 1: Summary of results for CoDMAP domains. Planners "2" and "4" are compilations having joint action size bounded by 2 and 4 respectively, while "∞" is the variant with unbounded joint action size. "FD" is Fast Downward directly applied to the given instances (no compilation involved). N is number of instances; time and length are averages over all instances solved, for all planners that solved at least 5 instances. The total number of actions is an average over all instances.

- Actions cannot be used in conditional effects, so their algorithm cannot solve instances of TABLEMOVER.
- To represent concurrency constraints on multiple action templates in PDDL, they have to be defined on the same subset of objects. In contrast, the constraints of Kovacs can be defined on arbitrary pairs of actions.

Furthermore, CJR does not separate the atomic action selection from the atomic action application. This is a big problem since one of the atomic actions can delete the precondition of other atomic actions, thus canceling the formation of the joint action. For example, in the MAZE domain, the action for crossing a bridge requires that the bridge exists, and destroys the bridge as an effect. Therefore, as this approach does not separate the selection from the application, this action can be done just by one agent at a time (and not by infinite agents as the problem states). The same occurs in the BOXPUSHING domain. Instances where a medium or a large box must be moved cannot be solved with this approach because the first agent to "push" the box will move it. Thus, the box location precondition for the other agent(s) does not hold, so the box is not moved in the end.

As for our compilations, we used Fast Downward in the LAMA 2011 setting to solve the instances produced by CJR.

Another algorithm we could have used for comparison is MAFS (Brafman and Zoran 2014). Unlike our approach and Crosby, Jonsson, and Rovatsos's, it is a distributed approach that preserves privacy and that has been shown to work in the BOXPUSHING domain. However, we have not tested it as we have not had access to the code to perform experiments.

Table 2 shows the results for the four domains. To provide an idea of how each planner behaves as a function of the number of agents, the table shows for each domain the same metrics for different numbers of agents. In the case of BOXPUSHING with three agents, as the version with the bound set to 2 cannot solve instances with large boxes, there are separate results for the cases where the number such boxes is 0, and the cases where it is greater than 0.

In terms of coverage, the unbounded compilation (∞) performs the best (93, 69.92%). The variant bounded to 2 and the variant bounded to 4 have similar coverage: 82 (61.65%) and 89 (66.92%) respectively. The former is clearly affected by the fact it cannot solve the instances requiring the concurrent action of 3 agents in the BOXPUSHING domain. Finally, CJR is the approach with the worst coverage (17, 12.78%). It performs reasonably well in MAZE in spite of its limitation regarding the action for crossing a bridge. On the other hand, its results are not very good in WORKSHOP, and it cannot solve instances from the TABLEMOVER and BOXPUSHING domains. Furthermore, the higher the number of agents, the worse the coverage becomes because problems are harder to solve (the number of available actions grows).

Regarding execution time, the unbounded compilation and the compilation bounded to 2 are the fastest. The higher the number of agents, the longer it takes to compute a plan.

In the case of plan length (i.e. number of joint actions), we observe that the plans obtained with our approach are shorter than the ones obtained with CJR. CJR obtains worse results because it explicitly builds joint actions only if their constituent joint actions are associated with a concurrency constraint. Thus, any action that appears out of a joint action, can be considered as a joint action of size 1. In contrast, our approach allows to combine atomic actions arbitrarily, so it already compresses the solution while planning.

In summary, it is not clear which variant is the best since they highly depend on the problem. In general, the version bounded to 2 works well, but it is useless if concurrency is required between 3 or more agents. The version bounded to 4 has a similar coverage to the unbounded one, but mainly because there are not instances requiring concurrency between more than 4 agents. Thus, we believe that the unbounded option is the most convenient since its performance is not far from the best in all domains, and it is more general.

Finally, we have performed preliminary scalability results in the BOXPUSHING domain. In these experiments we checked the results when $n \in \{1, \ldots, 10\}$ agents are required to move a box from a room r_1 to a neighbor room

Domain	N	Coverage				Time (s.)				Plan length				# Actions			
		2	4	∞	CJR	2	4	∞	CJR	2	4	∞	CJR	2	4	∞	CJR
MAZE	20	**13**	8	6	11	351.9	435.2	**144.4**	192.8	47.2	22.0	**11.7**	77.3	41723	69368	**27900**	156886
$a = 10$	10	**8**	6	5	7	243.6	564.8	**169.1**	225.5	48.3	25.0	**12.2**	79.6	39909	67417	**26155**	119374
$a = 15$	10	**5**	2	1	4	**525.2**	-	-	-	**45.4**	-	-	-	43989	71807	**30080**	194397
TABLEMOVER	24	**15**	12	**15**	-	**263.3**	336.5	341.0	-	**58.7**	59.0	61.5	-	7487	13127	**4667**	-
$a = 2$	12	10	10	**11**	-	**103.8**	226.4	214.6	-	63.5	**62.0**	64.5	-	3450	6154	**2098**	-
$a = 4$	12	**5**	2	4	-	**558.2**	-	-	-	**49.0**	-	-	-	11524	20100	**7236**	-
WORKSHOP	20	**15**	13	13	6	132.3	298.6	**51.8**	629.0	35.7	37.0	**32.5**	63.5	18002	31000	11502	**5425**
$a = 4$	10	**8**	**8**	**8**	5	42.1	261.6	**36.6**	587.3	**37.3**	43.9	**37.3**	65.8	7772	13621	4847	**2351**
$a = 8$	10	**7**	5	5	1	235.5	357.8	**76.0**	-	33.9	26.0	**24.8**	-	28231	48378	18157	**8499**
BOXPUSHING	69	39	56	**59**	-	**26.8**	79.9	63.9	-	**9.4**	11.0	10.2	-	3075	5360	**1932**	-
$a = 2$	21	**21**	**21**	**21**	-	**14.1**	15.6	16.2	-	10.5	10.6	**10.3**	-	2099	3775	**1261**	-
$a = 3$	48	18	35	**38**	-	**41.5**	118.5	90.2	-	**8.2**	11.2	10.2	-	3502	6054	**2226**	-
$l = 0$	21	**18**	17	**18**	-	**41.5**	64.7	53.3	-	**8.2**	8.3	8.3	-	3373	5887	**2116**	-
$l > 0$	27	-	18	**20**	-	-	169.2	**123.5**	-	-	13.9	**12.0**	-	3602	6184	**2312**	-

Table 2: Summary of results for domains requiring concurrency. Planners "2" and "4" are compilations having joint action size bounded by 2 and 4 respectively, while "∞" is the variant with unbounded joint action size. CJR is the compilation proposed by Crosby, Jonsson, and Rovatsos (2014). a is the number of agents, N is number of instances; time and length are averages over all instances solved, for all planners that solved at least 5 instances. For BOXPUSHING, l is the number of large boxes. The total number of actions is an average over all instances (solutions are not required to get this metric).

r_2. As n grows, the time required by FD for grounding increases due to the memory requirements. For $n = 6$, FD needs 6 seconds to find a solution; for $n = 7$, it needs 110 seconds; and for $n > 7$ it surpasses the memory limit.

Related Work

Several other authors consider the problem of concurrent multiagent planning. Boutilier and Brafman (2001) 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. CMAP (Borrajo 2013) produces an initial sequential plan for solving a MAP, but performs a post-processing step to compress the sequential plan into a concurrent plan.

Jonsson and Rovatsos (2011) 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 (Torreño, Onaindia, and Sapena 2014) is a partial-order planner that also allows agents to execute actions in parallel, but the authors do not present experimental results for MAP domains that require concurrency.

The planner of Crosby, Jonsson, and Rovatsos (2014) is similar to ours in that it also converts CMAPs into classical planning problems. The authors only present results from the MAZE domain, and concurrency constraints are defined as affordances on object sets that appear as arguments of actions. These concurrency constraints are not as flexible as those of Boutilier and Brafman (2001), since the latter can model any arbitrary concurrency constraint between pairs of actions, as well as unidirectional constraints that only affect one of the two actions. Moreover, affordances are not used to define concurrency constraints in conditional effects.

Brafman and Zoran (2014) extended the MA-STRIPS modeling language to support concurrency constraints, and the MAFS multiagent distributed planner to solve this kind of problems while preserving privacy. They examined the scalability of their approach in the BOXPUSHING domain.

Compilations from multiagent to classical planning have also been considered by other authors. Muise, Lipovetzky, and Ramirez (2015) proposed a transformation to respect privacy among agents. The resulting classical planning problem was then solved using a centralized classical planner as in our approach. Besides, compilations to classical planning have also been used in temporal planning, obtaining state-of-the-art results in many of the International Planning Competition domains (Jiménez, Jonsson, and Palacios 2015).

Conclusion

This work makes several contributions to concurrent planning. A common framework is introduced for different planning forms. We focused on the relation between concurrent and multiagent planning. We proposed a sound and complete method for compiling CMAPs into classical planning problems. The method does not need an exponential number of actions to represent the problem; instead, the number of resulting actions is linear in the description of the CMAP while respecting explicit concurrency constraints.

In future work, it would be interesting to explore strategies for encouraging concurrent actions that involve several agents (i.e. strategies for finding shorter concurrent plans). Furthermore, privacy preserving is a central topic on multiagent planning; thus, this approach could be combined with suitable privacy-preserving mechanisms in the future.

Acknowledgments

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

References

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

Boutilier, C., and Brafman, R. I. 2001. Partial-Order Planning with Concurrent Interacting Actions. *J. Artif. Intell. Res. (JAIR)* 14:105–136.

Brafman, R. I., and Domshlak, C. 2008. 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.

Brafman, R. I., and Zoran, U. 2014. Distributed Heuristic Forward Search with Interacting Actions. In *Proceedings of the 2nd ICAPS Distributed and Multi-Agent Planning workshop (ICAPS DMAP-2014)*.

Crosby, M.; Jonsson, A.; and Rovatsos, M. 2014. 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.

Crosby, M.; Rovatsos, M.; and Petrick, R. P. A. 2013. 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*.

Crosby, M. 2013. A temporal approach to multiagent planning with concurrent actions. *PlanSIG*.

Crosby, M. 2014. *Multiagent Classical Planning*. Ph.D. Dissertation, University of Edinburgh.

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

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

Jiménez, S.; Jonsson, A.; and Palacios, H. 2015. Temporal Planning With Required Concurrency Using Classical Planning. In *Proceedings of the 25th International Conference on Automated Planning and Scheduling (ICAPS'15)*.

Jonsson, A., and Rovatsos, M. 2011. 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*.

Komenda, A.; Stolba, M.; and Kovacs, D. L. 2016. The International Competition of Distributed and Multiagent Planners (CoDMAP). *AI Magazine* 37(3):109–115.

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

Maliah, S.; Brafman, R. I.; and Shani, G. 2017. Increased Privacy with Reduced Communication in Multi-Agent Planning. In *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling, ICAPS 2017, Pittsburgh, Pennsylvania, USA, June 18-23, 2017.*, 209–217.

Muise, C.; Lipovetzky, N.; and Ramirez, M. 2015. MAP-LAPKT: Omnipotent Multi-Agent Planning via Compilation to Classical Planning. In *Competition of Distributed and Multiagent Planners*.

Nardi, D.; Noda, I.; Ribeiro, F.; Stone, P.; von Stryk, O.; and Veloso, M. 2014. RoboCup soccer leagues. *AI Magazine* 35(3):77–85.

Nissim, R., and Brafman, R. I. 2014. Distributed Heuristic Forward Search for Multi-agent Planning. *J. Artif. Intell. Res.* 51:293–332.

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

Sheh, R.; Schwertfeger, S.; and Visser, A. 2016. 16 years of robocup rescue. *KI - Künstliche Intelligenz* 30(3):267–277.

Stolba, M.; Fiser, D.; and Komenda, A. 2016. 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.

Torreño, A.; Onaindia, E.; and Sapena, O. 2014. FMAP: Distributed cooperative multi-agent planning. *Appl. Intell.* 41(2):606–626.

Tozicka, J.; Jakubuv, J.; and Komenda, A. 2014. 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.