Temporal Multiagent Planning with Concurrent Action Constraints

Matthew Crosby and **Ronald P. A. Petrick**
School of Informatics
University of Edinburgh
Edinburgh EH8 9AB, Scotland, UK
m.crosby@ed.ac.uk, rpetrick@inf.ed.ac.uk

Abstract

This paper investigates how centralised, cooperative, multiagent planning problems with concurrent action constraints and heterogeneous agents can be encoded with some minor additions to PDDL, and how such encoded domains can be solved via a translation to temporal planning. Concurrency constraints are encoded on affordances (object-action tuples) and determine the conditions under which a particular object can (or must) be utilised concurrently. The effectiveness of the approach is evaluated on the Vehicles testing domain and on a new Warehouse domain, which is inspired by a real-world warehouse problem in which a centralised mission planner must find a concurrent plan for a fleet of robots in a manufacturing plant. The approach is shown to be promising, with the potential to support future work in the area.

Introduction

Planning in multiagent domains with concurrent actions—especially with complex concurrent action constraints—is not an easy task. With any meaningful number of agents and actions, the number of possible joint actions makes any direct planning approach infeasible. Furthermore, an efficient method for representing concurrency constraints is required as they are defined over the whole joint action space.

In previous approaches, concurrent action constraints are often defined on actions or unground action schema (Boutilier and Brafman 2001). While this technique provides a natural encoding method that specifies which *actions* can be performed simultaneously, it also leads to difficulties when dealing with problems that contain many objects that, while similar, have different concurrency properties. (For example, vehicles might carry different numbers of passengers, and doors might permit different numbers of agents to simultaneously pass through.) In domains such as these, it is useful to define concurrency constraints on the *objects* of the domain instead, leaving the features of PDDL that efficiently encode unground action schema untouched.

Given an encoding of concurrency constraints, the next task is to find plans that satisfy them. In this work, we consider an object-action representation of concurrency constraints and show that our encoding gives rise to an efficient translation into a temporal planning problem for which a solution is guaranteed to respect the concurrency constraints. The translation adds numeric fluents to the domain that keep track of which objects are being used, and also ensures that agents only perform one action at a time. We also show that, perhaps surprisingly, temporal planners can then be used to solve such problems in a reasonable time, even though there is no *a priori* reason to believe that they will be effective on the type of domains created by the translation process.

The motivating application for this work is a mission planning scenario for a fleet of heterogeneous robots in an assembly factory.[1] In this domain, the fleet of robots must navigate a factory floor and complete various picking and depalletising tasks in order to obtain a selection of parts as required for a kit. While one robot may be able to pick small, delicate objects, another may only be able to handle large, heavy objects. A centralised mission planner is in charge of finding and distributing plans for the robot fleet. As such, we treat this problem as an instance of centralised, completely cooperative, concurrent multiagent planning.

The capabilities of the robots are encoded in terms of *robot skills* (Bøgh et al. 2012), which are modelled at a level of abstraction above the robot control layer. Skill composition is traditionally done by hand, to create skill sequences that allow the robot to perform composite tasks. While this approach is satisfactory for controlled environments, by automating this process the robot will be better equipped to operate in continually-changing environments where the exact goals to be achieved are not necessarily known beforehand.

We model this domain as a classical planning problem, with additional concurrency constraints and agent action sets. The process of action execution is discretised into time steps with each action given an integer expected execution time. Currently, we only try to find a plan that assumes that the execution times are correct; we do not focus on methods for agent communication or the partial-order planning that would be required to deal with unknown execution times (Brenner 2003). In this work, plan synthesis and coordination are performed simultaneously so that a plan is generated both to achieve the goals and obey any concurrency constraints defined on the problem. Uncertainty about the envi-

[1] This work forms part of the EU STAMINA project. See http://stamina-robot.eu/ for more information.

ronment, and also about action execution success and duration, which are important features of the real-world problem, are left to future work and are not addressed in this paper.

The rest of this paper is structured as follows. We begin by discussing related work, broken down into four key topics: multiagent planning, temporal planning, robot skills, and affordances. The Warehouse domain is then introduced in order to motivate the approach taken in this paper. The details of the encoding are then presented, followed by the translation to temporal planning. Finally, we present the results of running a temporal planner on the translated domains, and conclude with a discussion of future work.

Related Work

We begin by discussing four strands of related work that are relevant to this paper: multiagent planning with concurrent actions, temporal planning, robot skills, and affordances.

Multiagent Planning with Concurrent Actions

As mentioned above, previous work on concurrent action constraints used the STRIPS (Fikes and Nilsson 1971) representation of actions, modified to include a concurrent action list describing the restrictions on the actions that could (or could not) be executed concurrently (Boutilier and Brafman 2001). This approach improved on previous work involving action interleaving which could not deal with truly concurrent interference effects. The work also showed how partial-order planning techniques could be used to solve such problems without the need for the introduction of an explicit notion of time. The proposed algorithm was not empirically evaluated but the work suggested that an interesting extension would be to utilise newer planning algorithms and techniques in solving such problems. Our work begins to do this, using a different encoding of constraints.

Previous work has also suggested encoding constraints on resources to specify which objects in the domain prohibit concurrent access (Knoblock 1994). We extend this idea to include resources which require concurrent access, and to include the possibility of resources that prohibit or require simultaneous access. While prior work has been interested in using the resource definitions to schedule concurrent actions to reduce overall plan execution time, we are more interested in encoding domains with inherent joint-action restrictions and planning over these restrictions.

Other relevant work considers joint actions in *strategic* multiagent planning (Jonsson and Rovatsos 2011). This work assumes the existence of an admissibility function that indicates which joint actions are possible, but does not discuss the details of such an encoding, instead simply assuming that an efficient encoding can be found depending on the features of a domain. A best-response planning (BRP) approach is introduced, that can find stable solutions to a certain class of planning problems called congestion games. Best response planning is well suited to plan refinement once a suitable starting plan has been found, but in our domains it is hard to find a suitable starting plan as this requires concurrent coordination from all the agents.

Temporal Planning

Temporal planning capabilities were introduced with PDDL 2.1 (Fox and Long 2003) and allow for the modelling of durative actions and the formulation of concurrent plans. In temporal planning problems, time is modelled explicitly, making temporal planning a natural fit for dealing with multiagent planning problems with concurrent actions. However, since (Brenner 2003) the approach has not been very prevalent in the recent multiagent planning literature.

In this paper we make use of POPF2 (Coles et al. 2010), a forward chaining partial-order planner which can find concurrent plans with low makespans. POPF2 was chosen because of its good performance—it was the runner-up in the temporal track of the 2011 International Planning Competition (Coles et al. 2012)—and because it has the capability to cope with all the constructs used in our translation.

Robot Skills

The encoding developed in this paper is motivated by the notion of *robot skills* (Bøgh et al. 2012), which have recently been proposed as an effective abstraction of the complex tasks that a robot can perform, and a tool for bridging the gap between low-level robot control and high-level planning.

In the taxonomy of robot skills, robot capabilities are separated into a three-level hierarchy consisting of motion primitives, skills and tasks. Motion primitives are the basic motion commands of the robot, at the lowest level of the hierarchy. Above this are robot skills, which represent the higher-level capabilities of the robot, which may require the use of many motion primitives. At the highest level are tasks, which can be scheduled without the need for specific robot control knowledge. It is at this level that planning occurs.

While the long-term goal of this work is to use automated methods for encoding planning actions from a description of robot skills, for this paper we hand-code each robot skill as a planning action, and then add separate skill distributions and concurrency constraints to the domain.

Affordances

Finally, we briefly consider the use of *affordances* in the related literature. The idea of an affordance can be traced back to Gibson (1977) and is a well-known concept in the robotics community and associated fields (see, e.g., (Duchon, Warren, and Kaelbling 1998; Lewis and Simó 2001; Steedman 2002; Sahin et al. 2007; Krüger et al. 2011), among others). For the purpose of this work, an affordance can be thought of as *the capacity of an object to be utilised in a certain manner*. We believe that affordances are an important component for encoding more complex concurrency constraints, and is a useful representational tool for building multiagent planning encodings. Affordances are discussed in more detail below in the context of encoding concurrency constraints.

The Warehouse Domain

We now describe the Warehouse domain, which models a problem in which a centralised mission planner is tasked with finding a concurrent plan for a heterogeneous robot fleet, working towards the shared goal of collecting sets of

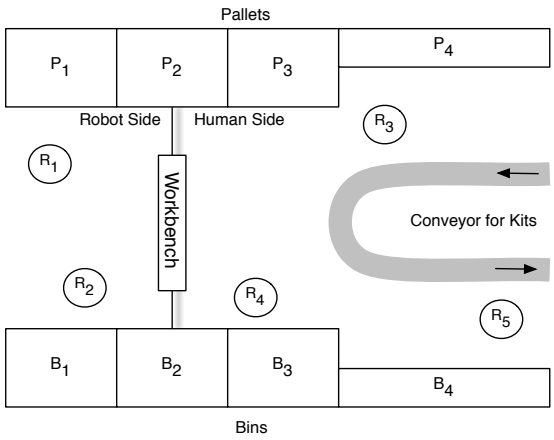

Figure 1: A depiction of the Warehouse domain. Some of the robots are only allowed on the robot side of the factory while others can navigate safely amongst humans. The aim is to pick given objects and place them in the correct kits which are located on the workbench and accessible to all robots. Once completed, the kits are carried to the conveyor and sent off for use in the manufacturing process.

parts (called kits) for use in a manufacturing plant. As mentioned above, the robots have a built-in representation of 'skills' which abstracts away from their low-level sensorimotor commands, and which denotes the capabilities that can be used to derive actions for planning purposes.

There are many different levels of abstraction at which the Warehouse domain can be modelled. For this initial investigation, we chose to focus on the features of concurrency constraints and heterogeneous agents, to test the feasibility of our encoding and of using temporal planning. This means that there are many factors that we do not cover that will nevertheless be important in a completely accurate representation of the real-world problem, including incomplete domain knowledge, action failure, and dynamic goals.

An example Warehouse problem is shown in Figure 1, which depicts the robots' operating environment. The goal of the domain is to assemble certain kits: collections of objects required for later use in the manufacturing process. The output should be a joint plan in which kits are put together efficiently and for which all the concurrency constraints are met so that, for example, no two agents attempt to access the same bin, pallet, or kit simultaneously.

The warehouse has been artificially split into two sections, one in which robots must navigate alongside humans, and the other in which only robots are allowed. Depending on the robot type, a robot may either be confined to the robot side or also allowed to work on the human side. A list of the skills that the each robot possesses, along with the time taken to perform each skill, is shown in Table 1. We assume that time can be discretised and that the actions always take the modelled time. In future extensions of this work, we will likely need to find partial-order plans that are robust to actions failing or taking longer than expected.

The pallets, found at the top of Figure 1, contain items that a robot can obtain if it has the depalletise skill. The bins at the bottom contain items that a robot can obtain if it can perform the requisite picking skill, either `pick_delicate` or `pick_heavy`, depending on the item in the bin. The complexities of robot navigation, and the picking, depalletising, and kitting processes are completely abstracted away by the robot skills formulation of our model, and assumed to be dealt with by low-level robot control processes.

Skill	Time	Robots with Skill
navigate-robot-side	2	R_1, R_2
navigate-human-side	3	R_3, R_4, R_5
pick-heavy	1	R_1, R_2, R_3
pick-delicate	1	R_1, R_3, R_4, R_5
depalletise	1	R_1, R_3, R_4
add-to-kit	1	R_1, R_2, R_3, R_4, R_5
deliver-kit	1	R_3, R_4, R_5

Table 1: Table showing the skills in the domain, their estimated duration, and the robots that can perform each skill.

While under completion, kits are placed on a workbench, accessible to all robots, but only one at a time. A finished kit must be carried to the conveyor, which requires two robots to carry the box simultaneously. While this last constraint is not part of the real-world problem, it is used to explore the ability of our approach to deal with concurrent coordination. The Vehicles domain presented later includes actions that require up to ten agents to coordinate concurrently.

Modelling and Encoding

This section discusses our method for encoding domains with properties such as those exhibited by the Warehouse domain. Our encoding is based on adding concurrency constraints to domains written in ordinary (non-temporal) PDDL (McDermott 2000). The features of the domains we would like to model include:

Multiagent Concurrent Actions: The domain contains multiple agents that execute their plans concurrently.

Cooperative Centralised Planning: The agents share the goal and a centralised mission planner plans for all agents.

Classical Planning Assumptions: All actions are deterministic, and there is complete domain knowledge with a fixed initial state and a known goal state (or set of states).

Agent Action Sets: Agents have different capabilities, the actions in the domain are split amongst the agents.

Concurrent Action Constraints: There are constraints over which actions can (or must) be performed simultaneously.

The problem that we are modelling is MA-STRIPS (Brafman and Domshlak 2008) (a multiagent extension of STRIPS) with the addition of concurrency constraints. More formally, we define a multiagent planning problem as a tuple $\Pi = \langle N, P, \{A_i\}_{i=1}^n, I, G, c\rangle$, where:

- $N = \{1, \ldots, n\}$ is the set of agents,
- P is the finite set of propositions of the domain,

- $I \subseteq P$ encodes the initial state,
- $G \subseteq P$ encodes the goal conditions,
- each A_i is agent i's action set, and
- $c : A_1 \times \ldots \times A_n \rightarrow \{0, 1\}$ specifies whether a particular joint action is valid under the concurrency constraints.

An action $a_i \in A_i$ has standard STRIPS syntax and semantics with $a_i = \langle pre(a), add(a), del(a) \rangle$ containing preconditions, add effects, and delete effects. A joint action $\bar{a} = (a_1, \ldots, a_n)$ is a member of the set $A = A_1 \times \ldots \times A_n$. For joint action $\bar{a}$, $pre(\bar{a}) = \bigcup_i pre(a_i), del(\bar{a}) = \bigcup_i del(a_i)$, and $add(\bar{a}) = \bigcup_i add(a_i)$. Joint action $\bar{a}$ is applicable in state s if and only if $del(\bar{a}) \cap add(\bar{a}) = \emptyset$, $c(\bar{a}) = 1$ (i.e., $\bar{a}$ satisfies the concurrency constraints), and $pre(\bar{a})$ holds in s. The application of a joint action updates the combined add and delete effects as in standard STRIPS.

A *plan* $\pi = [\bar{a}^1, \ldots, \bar{a}^k]$ is a sequence of joint actions such that $\bar{a}^1$ is applicable in the initial state I, and each subsequent joint action is applicable in the state resulting from the application of the previous action. It is assumed that each action set contains a no-op action with empty precondition and effects that can be used to align agents' plans.

As the domain of c is exponential in the number of agents, explicitly defining it (or even the set of joint actions) is not a practical approach. We therefore introduce an efficient method for encoding concurrency constraints by adding a construct to the PDDL problem file. A similar method is used to encode agent action sets for heterogeneous agents.

Encoding Agent Action Sets

We assume that the domain file contains a type `agent` which contains every member of N and no other objects. We also assume that each action has at least one parameter of type `agent`. A new construct, `capabilities`, is defined in the PDDL problem file which has the following (EBNF) syntax:

```
(:capabilities <cap-def>+)
<cap-def> ::= (<agent-name> <action-name>+)
```

where `agent-name` is a name of one of the agents, and `action-name` is a name of an action defined in the domain file. The intended interpretation is that the agent denoted by `agent-name` is capable of performing only the actions that appear in the list. In other words, the capabilities definition contains an agent and a list of action names that the agent can perform.

For example, for the Warehouse domain we would have:

```
(:capabilities
  (R1 navigate-robot-side pick-heavy
      pick-delicate depalletise add-to-kit)
  (R2 ...)
  ...
)
```

Using this method, it is possible to encode actions at the domain level without having to worry about which agents can perform each action in a particular problem instance.[2]

[2]As there are many problems for which some (or all) agents can perform all actions we assume that a missing capabilities definition means that all agents are capable of performing every action in the domain. We also assume that any agents not included in the capabilities definition have the ability to perform all actions.

Encoding Concurrency Constraints

At first glance, it may seem that concurrency constraints should be defined along with action specifications, since they constrain which actions can be performed concurrently. However, considering the type of domains we would like to encode, it makes more sense to associate constraints with objects. In particular, a standard action schema may have different concurrency constraints depending on the size, shape, or capacity of the object that it is ground to. (For example, consider a domain with multiple types of vehicles, or network connections of varying bandwidth.)

While the next obvious step might be to define concurrency constraints directly on the objects themselves, in the general case, this is not expressive enough. For instance, consider the case of an object that can be used in multiple different ways: a door might only be *passed through* by a single agent at a time, yet can be simultaneously *painted* by multiple agents. There are countless possible examples of *different ways* an object can be used, all of which potentially affect concurrency constraints. We call these different ways an object can be used its *affordances*.

Affordances are clearly related to combinations of objects and actions, but the exact details of this relationship are not obvious. A further complication comes from the fact that an affordance of an object may be associated with multiple actions. For example, a claw hammer may have the associated actions of `bash_nail` and `extract_nail`. However, in terms of a planning problem we may only want to constrain the higher level affordance of the hammer to be *used as tool*.

In order to deal with the preceding cases we define constraints over object-action tuples. Each tuple consists of an object and a list of actions that can be thought of as utilising the object for a particular affordance. This way an object can have multiple affordances (when it appears in different constraints with different lists of actions) and an affordance can be utilised by multiple actions (an action list is used).

We encode concurrency constraints as follows:

```
(:concurrencies <conc>+)
<conc> ::=
     (<obj-name> <act-name>+ <min> <max>)
```

where `obj-name` is an object from the problem definition, `act-name` is an action name from the domain file in which `obj-name` appears in a possible grounding, and `min` and `max` are positive integers with `max` $\geq$ `min`. The intended interpretation of the concurrency constraints is that 'not more than `max` and at least `min` agents can simultaneously utilise object o via the actions in the action list'. A concurrency constraint of `(o act 1 n)` therefore effectively provides no constraint on the object o.

We expect that all objects of a particular type share in their relevant affordances stipulating that if a concurrency constraint exists for object o of type t with action list $\bar{a}$, then a concurrency constraint exists with action list $\bar{a}$ for each object of type t. This can be easily achieved by adding 'dummy' affordances with $min = 1$ and $max = n$. We then

define the affordances of the domain as each pair (type, action list) for which a ground object of that type appears with that action list in the concurrencies specification.

The following example shows a selection of concurrency constraints for the Warehouse domain:

```
(:concurrencies
  (bin1 pick-delicate pick-heavy 1 1)
  (pallet1 depalletise 1 1)
  (kit1 deliver-kit 2 2)
  ...
)
```

In particular, it specifies that if a `pick-delicate` or `pick-heavy` action is applied to `bin1` then no other agent can concurrently perform a `pick` action on `bin1`. It also says that the action `deliver-kit`, when ground to `kit1`, must be performed by exactly two agents concurrently. More complicated concurrency constraints will be discussed in the evaluation section when we look at the Vehicles domain.

It should be noted that there are certain nuances to defining concurrency constraints in this way. Concurrency constraints specify constraints on the simultaneous execution of actions, not facts about what can be true over any state in the world. So, for example, modelling that two robots cannot be in the same location should be part of the domain definition as it is a fact about which possible states are allowed.

Translation to Temporal Planning

At this point, we have a domain written in classical single-agent PDDL, with concurrency constraints and capability definitions added to the problem file. The next step is to try and find a plan that obeys the intended semantics of the capabilities and concurrency constraints. To do this, we translate our encoded domains into PDDL 2.1 with durative actions and numeric fluents, for use with a temporal planner.

Pseudocode for the translation is shown in Algorithm 1, which is split into three steps: adding capabilities, translating to durative actions and adding concurrency constraints. Along with the introduced encoding, the translation assumes that the domain definition includes an *agent* type and there are at least two objects of this type, each action has an associated duration and all objects in the domain are typed.

Adding Capabilities

Adding capabilities is straightforward and does not require any temporal constructs and as such is performed before the translation to durative actions. For each action `act` that appears in the capabilities definition, a new `(can-act ?a - agent)` predicate is added to the list of predicates. Then, each action that appears gains an additional precondition `(can-act ?x)` for each parameter `?x` of type agent. This precondition ensures that any agent the action is ground to has capability of performing the action. Finally, for each agent `a` defined as capable of performing `act`, the static proposition `(can-act a)` is added to the initial state.

If an action does not appear in the capabilities, then it does not get modified, which means that it can be ground to any agent and therefore any agent may perform the action. If no capabilities are defined, then nothing is changed in this step and it is possible for all agents to perform each action.

Algorithm 1: Translate to temporal planning problem.

Input : PDDL domain and problem files (dfile, pfile)
Output: PDDL temporal problem and domain files

```
// Add Capabilities
foreach act in :capabilities do
    dfile.predicates.add(can-act ?a - agent)
    foreach predicate ?a of act of type agent do
        dfile.act.addprecondition(can-act ?a)
    foreach agent in :capabilities containing act do
        pfile.init.add(can-act agent)
// Translate to durative actions
dfile.requirements.add(:fluents)
dfile.predicates.add(free ?a - agent)
foreach act in dfile with duration d do
    action ← durative-action
    act.add(= ?duration d)
    precondition ← condition
    foreach condition in act do
        condition ← (at start (condition))
    foreach effect in act do
        if effect is positive then
            effect ← (at end (effect))
        else
            effect ← (at start (effect))
    foreach agent parameter ?a in act do
        act.conditions.add(at start(free ?a))
        act.effects.add(at start(not (free?a)))
        act.effects.add(at end(free ?a))
// Add concurrency constraints
foreach affordance with type t and action list ā do
    dfile.functions.add(using-t-ā ?o - t)
    dfile.funcitons.add(min-t-ā ?o - t)
    dfile.functions.add(max-t-ā ?o - t)
    foreach act in ā do
        if max > 1 then
            copy act to new act-join
            act-join.con.add(at start (> (using-t-ā ?o) 0))
            act-join.eff.add(at start (increase (using-t–ā ?o)
            1))
        act.con.add(at start (= (using-t–ā ?o) 0))
        act.con.add(at end (>= (using-t-ā ?o) (min-t-ā ?o)))
        act.con.add(at end (<= (using-t-ā ?o) (min-t-ā ?o)))
        act.eff.add(at start(increase (using-t-ā ?o) 1))
        act.eff.add(at end(assign (using-t-ā ?o) 0))
        act ← act-start
foreach concurrency constraint (o, ā, min, max) do
    pfile.init.add(= (using-t-ā o) 0)
    pfile.init.add(= (min-t-ā o) min)
    pfile.init.add(= (max-t-ā o) max)
```

Translating to Durative Actions

The next step of the algorithm is to translate all actions into durative actions for use in temporal planning. This step also adds a predicate `(free ?a)` that is used to make sure that each agent only ever performs a single action at a time. This predicate is removed whenever an agent starts an action and is only replaced on completion of the action (lines 20–23 in Algorithm 1). Action durations equal to those specified for each action are also added at this stage.

By convention, we assume that all preconditions are

required to be met at the beginning of the durative action, and they are therefore all changed to `(at start (condition))`. We set negative effects to be updated at the beginning of an action while positive effects occur on completion of the action. While it is certainly possible to design a domain for which this is a non-natural interpretation, it works well for the domains we have used and we assume that the domain creator is aware of this fact.

Adding Concurrency Constraints

The final part of the algorithm (lines 24–42) is the most involved and requires the addition of new actions and fluents. For each action, the translation creates at most two new temporal actions, 'action–start' and 'action–join'. For each different affordance, three new functions must be added to the domain: `using`, `min`, and `max`. The first of these is updated by the corresponding actions to show how many agents are currently utilising a constrained affordance. The latter two are used to ensure that the number of agents that simultaneously use a constrained affordance is between the minimum and maximum specified for that resource.

The first part of the translation shown in Algorithm 1 (lines 30–32) involves the creation of the new 'join' action for each action that appears in a concurrency constraint with $\texttt{max} > 1$. The additional elements for the join action will be explained after we have discussed the 'start' action. It appears first in the algorithm only because it is built from a copy of the unmodified start action.

To each start action, the following is added to the condition for each constrained parameter `?p`:

```
(at start (=  (using-p-ā ?p)  0))
(at end   (>= (using-p-ā ?p) (min-p-ā ?p)))
(at end   (<= (using-p-ā ?p) (max-p-ā ?p)))
```

This means a joint action can only be started if there are currently no agents already engaged in it, and by the end of the action the number of agents engaged with the related object is between the maximum and minimum values. The following is added to the effects for each constrained parameter:

```
(at start (increase (using-p-ā ?p) 1))
(at end   (assign   (using-p-ā ?p) 0))
```

This updates the number of agents utilising the relevant affordance of `?p` and then resets it once the action is complete.

The join actions are simpler. To each join action the following is added to the condition:

```
(at start (> (using-p-ā ?p) 0))
```

This ensures that a join action can only be performed if the start action is currently initiated. The following is added to the actions effects:

```
(at start (increase (using-p-ā ?p) 1))
```

which is simply the same counter used in the start action.

Putting this all together, we end up with a start action for each constrained action and a join action only for actions that have concurrency constraints that allow multiple agents (i.e., $\texttt{max} \geq 2$). There is no 'end' action needed as the conditions of the start action ensure that the correct number of agents are performing the action by the time it has finished. When run with a temporal planner, a joint action is formed from all actions scheduled at a particular time step. It is possible, for actions with long durations, that the join action does not appear in the same timestep as the start action (which violates the intended interpretation). However, the planner used in this work, POPF2, always schedules join actions immediately after start actions as it attempts to minimise makespan, making the above translation sufficient.

Optimisations

While the previous algorithm creates a functionally correct domain, there are many optimisations that can be performed based on the type of concurrency constraints that exist in the problem. We introduce a few of them here and then examine their effects on planning in the evaluation section below.

Actions that require two agents to perform them simultaneously can be encoded directly with two agent parameters, and by including the condition `(not (= ?a1 ?a2))` to ensure that the parameters cannot be ground to the same agent. However, this is not good practice in the general case (where n agents are required to perform the action) as, firstly, it requires $(n(n+1))/2$ inequality clauses and, secondly, this method does not allow the grounding of actions to objects with different types of concurrency constraints. If a concurrency constraint of `(o a 2 2)` appears for all groundings of an affordance, instead of following the algorithm, we use the equality definition just mentioned.

It is also possible to optimise the translation for concurrency constraints of the form `(o a 1 1)`. These constraints are quite common, as there are many objects that can only be used by one agent at a time. Notice that the constraint `(o a 1 1)` is similar to the constraint that each agent may only perform one action at a time, which can be encoded using a `(free ?a)` predicate. The same method can be used for objects, with a `(free-o ?o)` predicate in place of the agent one. A final small optimisation can be applied when an action appears in each agent's capabilities list. In this case, the associated `(can-act ?a)` predicate does not need to be added to the domain (i.e., lines 2-5 in Algorithm 1 can be skipped).

Evaluation

This section presents the results of running a temporal planner on the translated domains. A python script was written to perform the translations, including the optimisations mentioned above.[3] The planner we chose for this work was POPF2 (Coles et al. 2010), since it has the capability to cope with all the constructs used in the translation, and also to produce plans that attempt to minimise makespan. POPF2 was also the best planner in IPC11 at dealing with temporal problems with inbuilt concurrencies. All experiments were run on the same machine with 48GB of memory and a 2.66GHz processor. Experiments were allowed to run until a plan was found or an out of memory error was reported.

[3] Available for download at `http://homepages.inf.ed.ac.uk/mcrosby1`.

	Small		Medium		Large	
Kits	time(s)	span	time(s)	span	time(s)	span
1	0.02	21	0.03	12	0.14	8
2	0.78	49	1.15	32	0.93	18
3	8.01	87	9.71	53	74.5	24
4	30.7	105	37.5	62	–	–
5	106	139	133	76	–	–

Table 2: Table showing the time in seconds and makespan of the first plan found by POPF2 for different domain sizes and number of kits/goals.

Warehouse

This section presents an evaluation of the Warehouse domain. The first aim was to ensure that the planner could indeed solve problems of the scale we are likely to deal with in real-world settings. Three problem instances were created—small, medium and large—with medium having the same setup as depicted in Figure 1 and the robot capabilities shown in Table 1. The small domain involved just the right-hand side of Figure 1, with three robots, two bins (one containing a delicate item and one containing a heavy item), and two pallets. The only robots left were R_3, R_4, and R_5, which had the same skills as in the previous domain. The large domain contains ten robots, eight pallets, and eight bins. The five new robots added were given the same skills as those for R_1 to R_5, respectively, and the new pallets and bins were spread evenly across the two sides.

Table 2 shows how the planner performed over the different sizes of domains as the number of kits that needed to be delivered was changed. The goals were to deliver from one to five kits, each containing five separate parts. The goals were kept the same across the problems so that they could be compared directly in terms of time and makespan.

In the medium and large problems, the extra robots mean that more actions can be scheduled simultaneously so that the makespan can potentially be lower to achieve the same goals. It was unknown as to whether the fact that the problem was easier (due to more available robots) or that the problem was harder (due to a massively increased state space) would dominate the results. While being almost twice the size, the medium domain was solved in times that were not much slower than the small domain, showing that, perhaps surprisingly, increasing the number number of agents capable of performing tasks can somewhat counteract the increase in domain size. As can be seen from the table, the number of kits had a large effect on the planning times while increasing the number of robots and bins had a lesser effect.

While the final two large problems were unsolvable, the planner performed much better than expected given the number of joint actions, and the way they were encoded (not designed for ease of planning). For example, in the initial state of the medium problem each robot can perform eight or nine valid actions, meaning that there are over forty-six thousand possible joint actions. Obviously, the temporal planner does not deal directly with the joint action space, which is why it is a feasible approach for solving these kind of problems in the first place.

	No-Deliver		Equal		Robot-side	
Ag's	time(s)	span	time(s)	span	time(s)	span
3	0.97	65	0.97	70	0.97	70
4	4.25	41	4.82	42	4.82	42
5	8.18	37	10.83	36	4.71	29
6	6.64	26	15.82	24	8.02	28
7	14.06	23	26.16	28	–	–
8	35.25	23	56.77	26	14.4	41
9	139	24	210	22	51	39
10	43.25	21	70	20	26	39

Table 3: Table showing the time in seconds and makespan of the first plan found by POPF2 for different domain sizes and number of agents.

Given the previous results, we wanted to test how adding robots (but not otherwise increasing the problem size) affected planning. We used the middle problem from the previous table (medium size, 3 kits) as our starting point and only varied the number of agents. All agents were given every capability, except that half were confined to the robot side and half to the human side. The smallest problem instance contained 3 robots with two on the human side to ensure that the problem has a solution (i.e., the `deliver-kit` action requires two agents acting simultaneously).

The results of these experiments are shown in Table 3. The middle column shows the set-up described above and contains an anomalous looking result for the 9 agent problem. As the goal does not change over these problems, the makespan becomes closer to optimal as more agents are added (as there are more 'free' robots to assign actions to at each point). The time taken for the 9 robot problem appears to break the trend of the slow increase in planning times as the number of agents is increased. We hypothesised that this could be due to the `deliver-kit` action having exponentially more possible groundings now that another robot has been added to the human side. We therefore ran experiments without the `deliver-kit` action (column 1) and experiments where robots were only added to the robot side (column 3). From the results we can see that the `deliver-kit` action, while problematic, is not the sole cause of the anomaly. We can only conclude that due to the very large state space, there is noise in the results depending on the path of the heuristic search. This can also be seen by the fact that no result was found for the seven agent problem in column 3 even though one clearly exists.

Finally, we wanted to test the effectiveness of our translation optimisations. Table 4 shows the effects that the optimisations have on both planning times and plan cost. The reported values are of the form unoptimised/optimised (from Table 2) so that a value less than one represents an improved time or makespan. The cells containing 'x' show where the optimised version found a solution where the unoptimised version could not. The values of the table that contain '–' are cases where the planner did not manage to find a solution on the unoptimised translation. We can see that the coverage for the unoptimised case is much worse than for the optimised case. This means that the optimisations are nec-

	Small		Medium		Large	
Kits	time	span	time	span	time	span
1	333	**0.86**	3.33	1.08	1.86	1.13
2	14.7	**0.94**	22.3	1.03	120	1.06
3	1.20	**0.89**	7.97	**0.85**	x	x
4	2.71	**0.91**	x	x	–	–
5	x	x	x	x	–	–

Table 4: A comparison of the unoptimised translation with the results for the optimised translation. Each table entry is calculated as the value of unoptimised/optimised, meaning that the results show how much slower/more costly the unoptimised version was. A value of x shows where the unoptimised version did not find a plan but the optimised version did, whereas '–' represents that both translations failed.

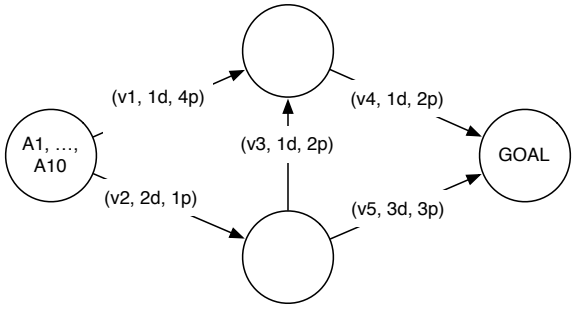

Figure 2: An example Vehicles problem. Ten agents (only six of which can drive), starting on the left, must travel to the rightmost location via a collection of vehicles requiring different numbers of drivers, and permitting different numbers of passengers. This problem is solved in 0.15s by POPF2.

essary, especially in the larger domains. It is also interesting to note that when the unoptimised domain returns a solution it is often of better quality than that of the unoptimised domain. However, coverage is a much more important factor as the planner can always be run for longer if a higher quality solution is required.

Vehicles

Finally, we test some further capabilities and properties of our approach on the Vehicles domain, first introduced in (Crosby 2014). In the domain, agents must use different vehicles to reach a particular goal location. The vehicles in the domain have associated concurrency constraints, such that one vehicle may require a driver and may hold up to four passengers, while another may require two drivers and not be able to hold any passengers. Some agents in the domain are designated as drivers and able to perform the `drive` action, while others are only able to perform the `passenger` action. An example problem is shown in Figure 2.

The Vehicles domain shows that it is possible to model conjunctive concurrency constraints with our approach, e.g.:

```
(and
  (v1 drive 1 1)
  (v1 passenger 0 4)
)
```

	Fig 2		(v, d1, p1)		(v, d5, p5)	
Drivers	time	span	time	span	time	span
5	–	–	–	–	0.03	2
6	0.15	3	–	–	0.15	2
7	–	–	0.64	5	2.42	2
8	0.32	4	1.1	5	–	–
9	–	–	2.2	6	–	–
10	3.13	5	1.99	6	–	–

Table 5: Results for the Vehicles domain as the number of drivers increases. The first column is for the problem shown in Figure 2 while the latter two are for the same problem with all vehicle constraints replaced with that shown.

The intended interpretation of this constraint is that the vehicle $v1$ can have at most one concurrent driver and simultaneously up to four passengers. This is translated to the temporal planning encoding by adding the condition:

```
(at start (> (using-drivable ?v ) 0))
```

to the passenger action. This method can be used for any conjunctive constraint by designating one element (with `min` > 0) as the initial action, and adding the relevant condition to all other actions. However, it is not yet known if this works for problems where the concurrency constraints overlap in terms of the actions they include.

The results of running our algorithm on the Vehicles domain are shown in Table 5. Interestingly, the results were very dependent on the number of driver agents in the domain, presumably because certain numbers of drivers lead to dead ends early in the problem. Overall, the coverage was not very impressive while the planning times when a solution was found were surprisingly fast. We take the optimistic view that the results are promising in that future work can build on the areas where temporal planners are effective for these type of problems.

Conclusion

This paper presented a method for encoding and solving planning problems with concurrent interacting actions and heterogeneous agents. For simple concurrency constraints, where the translation can be optimised, the approach was shown to be effective. However, more work is needed to plan with more complex concurrency constraints.

In the future, we plan to analyse further the conditions under which the temporal approach is effective, and use this as a starting point for creating a planning algorithm specifically designed for domains with concurrency constraints. We also intend to explore the notion of affordances for multiagent planning further and see how different representations can be utilised in the planning process. Finally, we intend to introduce some further complexities of the real-world Warehouse domain into our work.

Acknowledgements

The research leading to these results has received funding from the European Union's Seventh Framework Programme under grant agreement no. 610917 (STAMINA).

References

Bøgh, S.; Nielsen, O. S.; Pedersen, M. R.; Krüger, V.; and Madsen, O. 2012. Does your Robot have Skills? In *Proceedings of the International Symposium on Robotics (ISR 2012)*.

Boutilier, C., and Brafman, R. 2001. Partial-order planning with concurrent interacting actions. *Journal of Artificial Intelligence Research* 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 International Conference on Automated Planning and Scheduling (ICAPS 2008)*, 28–35.

Brenner, M. 2003. A Multiagent Planning Language. In *Proceedings of the Workshop on PDDL at the International Conference on Automated Planning and Scheduling (ICAPS 2003)*.

Coles, A. J.; Coles, A. I.; Fox, M.; and Long, D. 2010. Forward-Chaining Partial-Order Planning. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS 2010)*, 42–49.

Coles, A. J.; Coles, A.; Olaya, A. G.; Celorrio, S. J.; López, C. L.; Sanner, S.; and Yoon, S. 2012. A Survey of the Seventh International Planning Competition. *AI Magazine* 33(1):83–88.

Crosby, M. 2014. A Temporal Approach to Multiagent Planning with Concurrent Actions. In *Workshop of the UK Planning and Scheduling Special Interest Group (PlanSIG 2013)*.

Duchon, A. P.; Warren, W. H.; and Kaelbling, L. P. 1998. Ecological robotics. *Adaptive Behavior, Special issue on biologically inspired models of navigation* 6(3-4):473–507.

Fikes, R. E., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2(3-4):189–208.

Fox, M., and Long, D. 2003. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research* 20:61–124.

Gibson, J. 1977. The theory of affordances. In Shaw, R., and Bransford, J., eds., *Perceiving, Acting, and Knowing: Toward an Ecological Psychology*, 67–82.

Jonsson, A., and Rovatsos, M. 2011. Scaling Up Multiagent Planning: A Best-Response Approach. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS 2011)*, 114–121.

Knoblock, C. A. 1994. Automatically Generating Abstractions for Planning. *Artificial Intelligence* 68:243–302.

Krüger, N.; Geib, C.; Piater, J.; Petrick, R.; Steedman, M.; Wörgötter, F.; Ude, A.; Asfour, T.; Kraft, D.; Omrčen, D.; Agostini, A.; and Dillmann, R. 2011. Object-Action Complexes: Grounded abstractions of sensory-motor processes. *Robotics and Autonomous Systems* 59(10):740–757. doi:10.1016/j.robot.2011.05.009.

Lewis, M. A., and Simó, L. S. 2001. Certain principles of biomorphic robots. *Autonomous Robots* 11(3):221–226.

McDermott, D. 2000. The 1998 AI Planning Systems Competition. *AI Magazine* 21(2):35–55.

Sahin, E.; Çakmak, M.; Doğar, M. R.; Uğur, E.; and Ücoluk, G. 2007. To afford or not to afford: A new formalization of affordances toward affordance-based robot control. *Adaptive Behavior* 15(4):447–472.

Steedman, M. 2002. Plans, affordances, and combinatory grammar. *Linguistics and Philosophy* 25(5-6):723–753.