Centralized Planning for Distributed Plans

Partial Order Planning with Concurrent Interacting Actions

by C. Boutilier and R. Brafman

Pavlos Moraitis
Example

To illustrate some of these issues, consider the following example which will be discussed in detail later in the paper: two agents must move a large set of blocks from one room to another. While they could pick up each block separately, a better solution would be to use an existing table in the following manner. First, the agents put all blocks on the table, then they each lift one side of the table. However, they must lift the table simultaneously; otherwise, if only one side of the table is lifted, all the blocks will fall off. Having lifted the table, they must move it to the other room. There they put the table down. In fact, depending on the precise goal and effects of actions, it may be better for one agent to drop its side of the table, causing all of the blocks to slide off at once. Notice how generating this plan requires the agents to coordinate in two different ways: first, they must lift the table together so that the blocks do not fall; later, one of them (and only one) must drop its side of the table to let the blocks fall.
Introduction

• Since the actions of distinct agents interact, we cannot, in general specify the effects of an individual’s actions without taking into account what other actions might be performed by other agents at the same time

• One way to handle action interaction is to specify the effects of all joint actions directly

• Let $A_i$ be the set of actions available to agent $i$ (assuming $n$ agents labeled $1...n$) and let the joint space be $A_1 \times A_2 \times ... \times A_n$

• We can treat each element of this space as a separate action and specify its effects using a STRIPS action representation

• The main advantage of this reduction scheme is that the resulting planning problem can be tackled using any standard planning algorithm
Introduction (cont.)

• However this solution has some drawbacks with respect to ease representation:
  – The number of joint actions increases exponentially with the number of agents; this has several implications for the specification and planning process
  – This reduction fails to exploit the fact that a substantial fraction of the individual actions may not interact at all, or at least not interact under certain conditions
  – We need a representation of actions in multi-agent setting that exploits the independence of individual action effects to whatever extent possible
  – E.g. while the lift actions of the two agents may interact, many other actions will not—one agent lifting the table and another picking up a block
  – Hence we don’t need to explicitly consider all combinations of these actions and can specify certain individual effects separately, combining the effects “as needed”
Problems to be taken into account…

• The representation problem: how do we naturally and concisely represent interactions among concurrently executed actions

• The planning problem: how do we plan in the context of such a representation
Representing Concurrent Actions and Plans

- The STRIPS Action Representation

```
(define (action pickup-block-B-from-floor)
  :precondition (and (on floor B) (handempty) (clear B))
  :effect (and (not (handempty)) (not (on floor B)) (holding B)))
```

Figure 1: The *Pickup-block-B-from-floor* action

```
(define (operator pickup)
  :parameters (?, ?)
  :precondition (and (on ? ?) (handempty) (clear ?) (not (= ? ?)))
  :effect (and (not (handempty)) (not (on ? ?)) (holding ?)))
```

Figure 2: The *Pickup* action schema
Representing Concurrent Actions and Plans (cont.)

• The STRIPS representation can be enhanced obtaining a more expressive language that allows for a form of universal quantification in the action description (e.g. as in UCPOP)

• Conditional effects can be captured using a when clause consisting of an antecedent and a consequent

• In states $s$ that satisfy the preconditions of the actions and the antecedent of the when clause, the actual effect is the union of the “standard” effects in the effect list and the consequent of the when clause
Representing Concurrent Actions and Plans (cont.)

Representing Concurrent Actions in STRIPS

- The introduction of concurrent interacting actions requires to address two issues specific to the multi-agent setting:
  - Who is performing the action
  - What other actions are being performed at the same time

- The identity of the performing agent is declared by introducing an agent variable to each agent schema

- In order to define the actions that may or may not be performed concurrently a set of constraints is necessary
  - Such constraints are captured by adding a concurrent action list
Representing Concurrent Actions and Plans (cont.)

- The concurrent action list is a list of action schemata and negated action schemata some of which can be partially instantiated.

- If an action schema A’ appears in the concurrent action list of an action A then an instance of a schema A’ must be performed concurrently with action A in order to have an intended effect.

- If an action schema A’ appears negated in the concurrent action list of an action A then no instance of a schema A’ can be performed concurrently with action A if A is to have the prescribed effect.

- The concurrent action list is similar to the precondition list in the sense that when the constraints it specifies on the environment in which the action is performed are satisfied, the action will have the effects specified in the effect list.
Representing Concurrent Actions and Plans (cont.)

• A schema $A'$ appearing in the concurrent list action list $A$ can be partially instantiated or constrained:
  – If $A'$ contains free variables appearing in the parameter list of $A$, then these variables must be instantiated as they are instantiated in $A$

• Constraints that restrict the possible instantiations of the schema $A$ can appear within the concurrent action list

```
(define (operator pickup)
  :parameters (?a1 ?x ?y)
  :precondition (and (on ?x ?y) (handempty ?a1) (clear ?x) (not (= ?x ?y)))
  :concurrent (not (and (pickup ?a2 ?x ?y) (not (= ?a1 ?a2))))
  :effect (and (not (handempty ?a1)) (not (on ?x ?y)) (holding ?a1 ?x)))
```

Figure 3: The multiagent *Pickup* schema
Representing Concurrent Actions and Plans (cont.)

- This representation it is possible to represent actions whose effects are modified by the concurrent execution of other actions

```
(define (operator lower)
  :parameters (?a1 ?s1)
  :precondition (and (holding ?a1 ?s1) (raised ?s1))
  :effect (and (not (raised ?s1))
    (forall ?x
      (when ((ontable ?x)
        (not (and (lower ?a2 ?s2)(not (= ?s1 ?s2)))))
        (and (onfloor ?x) (not (ontable ?x)))))))
```

Figure 4: The Lower action schema
Representing Concurrent Actions and Plans (cont.)

• This operator contains a universally quantified effect that is an effect of the form (forall ?x (effect ?x))
  – This allows to state that the conditional effect described by the when clause applies to any object ?x
  – When we lower one side of the table, will cause that object the fall as long as the other side of the table is not being lowered at the same time
  – We use universal quantification to describe the fact that this will happen to any object that is on table

• In the concurrent part of the antecedent appears a constrained schema

• An action description can have no when clause, one when clause or multiple when clause; in the later case, the preconditions of all the when clauses must be disjoint
The Semantics of Concurrent Action Specification

- It is not individual actions that transform one state of the world into another state of the world.

- **Joint** actions define these transitions; they describe the set of individual actions (some of which could be no-ops) performed by each of the agents.

- Joint actions are n-tuples of individual actions.

- Given a joint action $\alpha = <\alpha_1, \ldots, \alpha_n>$ we refer to the individual actions $\alpha_i$ as the **elements** of $\alpha$.

- We say that the concurrent action list of an element $\alpha_i$ of $\alpha$ is **satisfied** with respect to $\alpha$ just when:
  - for every positive schema $A$ in this list, $\alpha$ contains some element $\alpha_j$ ($j \neq i$) which is an instance of $A$.
  - for every negative schema $A'$ in the list, none of the elements $\alpha_j$ ($1 \leq j \leq n$) is an instance of $A'$. 
The Semantics of Concurrent Action Specification (cont.)

**Definition** Let $a = \langle a_1, \ldots, a_n \rangle$ be a joint action where no individual action $a_i$ contains a *when* clause. We say $a$ is consistent if

- The precondition lists $p_i$ of each $a_i$ are jointly (logically) consistent (i.e., they do not contain a proposition and its negation).
- The effect lists $e_i$ of each $a_i$ are jointly consistent.
- The concurrent action list of each element of $a$ is satisfied w.r.t. $a$.

**Definition** Given a joint action $a = \langle a_1, \ldots, a_n \rangle$ and state $s$, the active *when* clause $w_i$ of $a_i$ relative to $s$ and $a$ is the (unique) *when* clause that is satisfied by $s$ and $a$ (i.e., whose preconditions are satisfied by $s$ and whose concurrency constraints are satisfied by $a$).
Definition Let $a = \langle a_1, \ldots, a_n \rangle$ be a joint action (where individual actions $a_i$ may contain \textit{when} clauses). Let $s$ be some state, let $w_i$ be the active \textit{when} clause for $a_i$ (w.r.t. $s$, $a$), and let $w_i$ have preconditions $wp_i$, concurrency constraints $wc_i$, and effects $we_i$. We say $a$ is \textit{consistent} at state $s$ if:

- The precondition lists $p_i$ and active \textit{when}-preconditions $wp_i$ of each $a_i$ are mutually consistent.
- The effect lists $e_i$ and active \textit{when}-effects $we_i$ of each $a_i$ are mutually consistent.
- The concurrent action list of each element of $a$ is satisfied w.r.t. $a$. 


Concurrent Plan Representation

• The representation of multi-agent plans is a rather straightforward extension of standard single-agent, partially ordered plan representations

• A (single-agent) linear plan consists of:
  – A set of action instances
  – Various strict ordering constraints using the relations < and > on the ordering of these actions
  – Codesignation or non-codesignation constraints on the values of variables appearing in these actions forcing them to have the same or different values respectively
  – A plan of this sort represents its sets of possible linearizations, the set of totally ordered plans formed from its actions instances that do not violate any of the ordering, codesignation, and non-codesignation constraints
  – A plan is consistent if it has some linearization
  – The set of linearizations can be seen as the “semantics” of a nonlinear plan in some sense
  – A (consistent) nonlinear plan satisfies a goal G, given starting state s, if any linearization is guaranteed to satisfy G
Concurrent Plan Representation

• A concurrent nonlinear planning for n agents (1,…n) is similar:
  – It consists of a set of action instances (with agents arguments, though not necessarily instantiated)
  – A set of arbitrary ordering constraints over the actions (i.e. <, >, = and ≠)
  – The usual codesignation or non-codesignation constraints
  – Unlike single-agent nonlinear plans it is allowed equality and inequality ordering constraints so that concurrent or nonconcurrent execution of a pair of actions can be imposed
  – The semantics must allow for the concurrent execution of actions by n agents
  – To this end the notion of linearization is extended
**Definition** Let \( P \) be a concurrent nonlinear plan for agents 1, \ldots, \( n \). An \( n \)-linearization of \( P \) is a sequence of joint actions \( A_1, \ldots, A_k \) for agents 1, \ldots, \( n \) such that

1. each individual action instance in \( P \) is a member of exactly one joint action \( A_i \);
2. no individual action occurs in \( A_1, \ldots, A_k \) other than those in \( P \), or individual No-op actions;
3. the codesignation and non-codesignation constraints in \( P \) are respected; and
4. the ordering constraints in \( P \) are respected. More precisely, for any individual action instances \( a \) and \( b \) in \( P \), and joint actions \( A_j \) and \( A_k \) in which \( a \) and \( b \) occur, any ordering constraints between \( a \) and \( b \) are true of \( A_j \) and \( A_k \); that is, if \( a \{<, >, =, \neq \} b \), then \( j \{<, >, =, \neq \} k \).
Concurrent Plan Representation

- The actions in P are arranged in a set of joint actions such that the ordering of individual actions satisfies the constraints and “synchronization” is ensured by no-ops.

- If we have a set of \( k \) actions (which are allowed to be executed by distinct agents) with no ordering constraints, the set of linearizations includes:
  - The “short” plan with a single joint action where all \( k \) actions are executed concurrently by different agents (assuming \( k \leq n \))
  - A “strung out” plan where the \( k \) actions are executed one at a time by a single agent with all others doing nothing.
Example

Example Suppose our planner outputs the following plan for a group of three agents: the set of actions is

\{a(1), b(2), c(2), d(3), e(1), f(2)\}

with the ordering constraints

\{e(1) = b(2), c(2) \neq d(3), a(1) < e(1), d(3) < f(2)\}

- The numerical arguments denote the agents performing the action
- Joint actions involve one action for each of the three agents
Example (cont.)

A simple 3-linearization of this plan using N to denote no-ops for the corresponding agent is:

\[ \langle a(1), N(2), N(3) \rangle, \langle e(1), b(2), N(3) \rangle, \langle N(1), c(2), N(3) \rangle, \langle N(1), N(2), d(3) \rangle, \langle N(1), f(2), N(3) \rangle \]

Another possible linearization is:

\[ \langle a(1), c(2), N(3) \rangle, \langle e(1), b(2), d(3) \rangle, \langle N(1), f(2), N(3) \rangle \]

-the shortest 3-linearization of the plan
The POMP algorithm

- The planning algorithm call function is $POMP(<A, O, L, NC, B>, agenda)$
  - $A$ contains all action instances inserted into plan so far
  - $O$ contains ordering constraints on elements on $A$
  - $L$ contains causal links
  - $NC$ contains nonconcurrency constraints
  - $B$ contains the current codesignation constraints

- $L$, $NC$, $B$ are initially empty

- $A$ contains $A_0$ and $A_\infty$
The POMP algorithm (cont.)

• One key distinction between POP and POMP is the notion of a *threat* used in POMP which is more general than in POP
  – Threats are handled using *demotion* (like in POP) or *weak promotion*
  – The later differs from the standard promotion technique used in POP: it allows $A_t$ (when $A_p \leq A_t < A_c$) to be ordered *concurrently* with $A_c$ not just after $A_c$

• In concurrent plans there exists another form of threats, namely *NC-threats*
  – The action instance $A_t$ *threatens* the concurrency constraint $A \neq A_c$ if $O \cup \{A_t = A_c\}$ is consistent and $A_t$ is an instantiation of $A$ that does not violate any of the codesignation constraints
  – Demotion and promotion can be used to handle NC-threats
The POMP algorithm (cont.)

- The POMP algorithm must check for the consistency of ordering constraints in several places:
  - In *Action Selection* where an action chosen to achieve an effect must be consistently ordered before the consumer of that effect
  - In *Concurrent Action Selection* where each concurrency requirement added to the plan must be tested for consistency
  - In *Nonconcurrency Enforcement* where demotion or promotion is used to ensure that no nonconcurrency requirements are violated

- The consistency testing of a set of ordering constraints is very similar to that employed in POP with one key difference
  - The existence of equality (=) and inequality (≠) ordering constraints as opposed to simple strict inequalities (i.e. < and >)
  - Equality can be dealt with by simply “merging” actions that must occur concurrently
  - Inequalities are easily handled by assuming all actions occur at different points whenever possible
Planning with Concurrent Actions

POMP({A, O, L, NC, B}, agenda)

Termination: If agenda is empty, return {A, O, L, NC, B}.

Goal Selection: Let \( Q, A_{need} \) be a pair on the agenda. (\( A_{need} \) is an action and \( Q \) is a conjunct from its precondition list.)

Action Selection: Let \( A_{add} = \text{Choose} \) an action one of whose effects unifies with \( Q \) subject to the constraints in \( B \). (This may be a newly instantiated action from \( \Lambda \) or an action that is already in \( A \) and can be ordered consistently prior to \( A_{need} \).) If no such action exists, then return failure. Let \( L' = L \cup \{ A_{add} \overset{Q}{\rightarrow} A_{need} \} \). Form \( B' \) by adding to \( B \) any codesignation constraints that are needed in order to force \( A_{add} \) to have the desired effect. Let \( O' = O \cup \{ A_{add} \overset{<}{\rightarrow} A_{need} \} \). If \( A_{add} \) is newly instantiated, then \( A' = A \cup \{ A_{add} \} \) and \( O' = O' \cup \{ A_0 < A_{add} < A_\infty \} \) (otherwise, let \( A' = A \)).

Concurrent Action Selection: If \( A_{add} \) is newly instantiated then apply the following steps to all positive actions \( a_{conc} \) in its concurrent list: Let \( A_{conc} = \text{Choose} \) a newly instantiated action from \( \Lambda \) or an action that is already in \( A \) and can be ordered consistently concurrently with \( A_{add} \). Make sure that there is a free agent that can perform this action concurrently with \( A_{add} \) and any other concurrently scheduled actions. If no such action exists then return failure. Let \( O' = O \cup \{ A_{conc} \overset{<}{\rightarrow} A_{need} \} \). If \( A_{conc} \) is newly instantiated, then \( A' = A \cup \{ A_{add} \} \) and \( O' = O' \cup \{ A_0 < A_{conc} < A_\infty \} \) (otherwise, let \( A' = A \)). If \( a_{add} \) is the agent variable in \( A_{add} \) and \( a_{conc} \) is the agent variable in \( A_{conc} \), then add \( a_{add} \neq a_{conc} \) to \( B' \), as well as all similar non-codesignation constraints for actions \( A \) such that \( A = A_{add} \in O \).
Re-apply this step to \( A_{conc} \) if needed.
For every negative action \( A_{\neg conc} \) in \( A_{add} \) concurrent list let \( NC' = NC \cup \{ A_{\neg conc} \neq A_{add} \} \). Add to \( B' \) any codesignation constraints associated with \( A_{\neg conc} \).
Planning with Concurrent Actions

Updating of Goal State: Let $agenda' = agenda - \{(Q, A_{need})\}$. If $A_{add}$ is newly instantiated, then add $\{(Q_j, A_{add})\}$ to $agenda'$ for every $Q_j$ that is a logical precondition of $A_{add}$. Add the other preconditions to $B'$. If additional concurrent actions were added, add their preconditions as well.

Causal Link Protection: For every action $A_t$ that might threaten a causal link $A_p \xrightarrow{R} A_c$ perform one of

(a) Demotion: Add $A_t < A_p$ to $O'$.
(b) Weak Promotion: Add $A_t \geq A_c$ to $O'$. If no agent can perform $A_t$ concurrently with $A_c$, add $A_t > A_c$, instead.

If neither constraint is consistent, then return failure.

Nonconcurrency Enforcement For every action $A_t$ that threatens a nonconcurrency constraint $A \neq A$ (i.e., $A_t$ is an instance of the schema $A$ that does not violate any constraint in $B'$) add a consistent constraint, either

(a) Demotion: Add $A_t < A$ to $O'$.
(b) Promotion: Add $A_t > A$ to $O'$.

If neither constraint is consistent, then return failure.

Recursive Invocation: $POMP(\{A', O', L', NC', B'\}, agenda')$

Figure 6: The Partially Ordered Multiagent Planning algorithm
Example of POMP Algorithm

- $\text{Pickup}(a, b)$: agent $a$ picks up a block $b$
- $\text{PutDown}(a, b)$: agent $a$ puts block $b$ on the table
- $\text{ToTable}(a, s)$: agent $a$ moves to side $s$ (left, right) of the table
- $\text{MoveTable}(a, r)$: agent $a$ moves to room $r$ with the table
- $\text{Lift}(a, s)$: agent $a$ lifts side $s$ of the table
- $\text{Lower}(a, s)$: agent $a$ lowers side $s$ of the table

The possible actions
The domain is described using the following predicates:

- \textit{OnTable}(b): block \( b \) is on the table
- \textit{OnFloor}(b): block \( b \) is on the floor
- \textit{AtSide}(a, s): agent \( a \) is at side \( s \) (left, right) of the table
- \textit{Up}(s): side \( s \) of the table is raised
- \textit{Down}(s): side \( s \) of the table is on the floor
- \textit{InRoom}(x, r): object \( x \) (agent, block, table) is in room \( r \)
- \textit{HandEmpty}(a): the hand of agent \( a \) is empty
- \textit{Holding}(a, x): agent \( a \) is holding \( x \) (block, side of table)
The initial state of our planning problem is:

\{InRoom(B, Room1), OnFloor(B), InRoom(Agent1, Room1), InRoom(Agent2, Room1),
    InRoom(Table, Room1), Down(LeftSide), Down(RightSide)\}

The goal propositions are:

\{InRoom(B, Room2), OnFloor(B), Down(LeftSide), Down(RightSide)\}
(define (operator pickup)
  :parameters (?a1 ?x)
  :precondition (and (inroom ?a1 ?r1) (inroom ?x ?r1)
                  (handempty ?a1) (onfloor ?x))
  :concurrent  (and (not (pickup ?a2 ?x)) (not (= ?a1 ?a2)))
  :effect      (and (not (handempty ?a1)) (not (onfloor ?x)) (holding ?a1 ?x)))

(define (operator putdown)
  :parameters (?a1 ?x)
  :precondition (and (inroom ?a1 ?r1) (inroom ?x ?r1) (inroom Table ?r1)
                  (holding ?a1 ?x))
  :concurrent  (not (lift ?a2 ?s1))
  :effect      (and (not (holding ?a1 ?x)) (ontable ?x) (handempty ?a1)))

(define (operator totable)
  :parameters (?a1 ?s1)
  :precondition (and (inroom ?a1 ?r1) (inroom Table ?r1) (not (atside ?a2 ?s1)))
  :concurrent  (and (not (totable ?a2 ?s1)) (not (= ?a1 ?a2))
                  (atside ?a1 ?s1))
(define (operator movetable)
  :parameters   (?a1 ?r1)
  :precondition (holding ?a1 Table)
  :concurrent   (and (movetable ?a2 ?r1) (not (= ?a1 ?a2)))
  :effect       (and (inroom ?r1 Table) (inroom ?r1 ?a1)
                  (when ((ontable ?x) ()) (inroom ?r1 ?x))))

(define (operator lower)
  :parameters   (?a1 ?s1)
  :precondition (and (holding ?a1 ?s1) (up ?s1))
  :concurrent   (and (not (lift ?a2 ?s2)) (not (= ?a1 ?a2)) (not (= ?s1 ?s2)))
  :effect       (and (not (up ?s1)) (not (holding ?a1 ?s1))
                  (when ((and (ontable ?x) (up ?s2) (not (= ?s1 ?s2)))
                  (and (not (lower ?a2 ?s2)) (not (= ?a2 ?a1))))
                  (and (onfloor ?x) (not (ontable x))))))

(define (operator lift)
  :parameters   (?a1 ?s1)
  :precondition (and (atside ?s1 ?a1) (down ?s1) (down ?s2) (not (= ?s1 ?s2)))
  :concurrent   (and (not (lower ?a2 ?s2)) (not (= ?a1 ?a2)) (not (= ?s1 ?s2)))
  :effect       (and (not (down ?s1)) (up ?s1) (holding ?a1 ?s1)
                  (when ((and (ontable ?x) (down ?s2) (not (= ?s1 ?s2)))
                  (and (not (lift ?a2 ?s2))))
                  (and (onfloor ?x) (not (ontable x))))))