Planning for Loosely Coupled Agents using Partial Order Forward-Chaining

Jonas Kvarnström
Department of Computer and Information Science
Linköping University, SE-58183 Linköping, Sweden (jonkv@ida.liu.se)

Abstract

We investigate a hybrid between temporal partial-order and forward-chaining planning where each action in a partially ordered plan is associated with a partially defined state. The focus is on centralized planning for multi-agent domains and on loose commitment to the precedence between actions belonging to distinct agents, leading to execution schedules that are flexible where it matters the most. Each agent, on the other hand, has a sequential thread of execution reminiscent of forward-chaining. This results in strong and informative agent-specific partial states that can be used for partial evaluation of preconditions as well as precondition control formulas used as guidance. Empirical evaluation shows the resulting planner to be competitive with TLPLAN and TALplanner, two other planners based on control formulas, while using a considerably more expressive and flexible plan structure.

1 Introduction

A major earthquake has struck in the middle of the night with devastating effects. Injured people are requesting medical assistance, many of them spread out in sparsely located homes in the countryside. Clearing all roadblocks will take days. Fortunately we have access to a fleet of unmanned aerial vehicles (UAVs) that can rapidly be deployed to send crates of medical and food supplies. In preparation, ground robots can move crates out of warehouses and onto carriers that allow a UAV to carry multiple crates.

This dramatic scenario is just one example of a wide class of domains involving multiple agents in the sense of "execution entitites" that may or may not be autonomous. Other domains abound, with planning competitions providing a variety of examples such as the ZenoTravel, Satellite, Rovers, Elevators and Storage domains.

Though any planner can plan for multiple agents by modeling them as action arguments, some plan structures are more suitable than others. For example, a sequential planner may generate a plan where a single UAV delivers all crates, as this would have exactly the same apparent quality as a plan making good use of all available agents in parallel. This cannot be remedied by relaxing precedence constraints after planning (Bäckström 1998). We therefore prefer to work directly with partial-order (PO) plans, giving the planner access to a suitable model for concurrent execution.

Most planners generating partially ordered plans use partial-order causal link (POCL[1]) techniques, which were initially conceived to increase the efficiency of planning: *Least commitment*, avoiding "premature commitments to a particular [action] order" (Sacerdoti 1975), requires less backtracking. Sequential execution was usually assumed, as reflected in the common definition of a PO plan being correct iff every compatible total ordering is correct. Thus, partial ordering was originally a means to an end.

Despite lacking the benefits of least commitment, current forward-chaining (FC) planners tend to be more efficient than PO planners, as they can generate complete state information and take advantage of powerful state-based heuristics or domain-specific control (Bacchus and Kabanza 2000; Kvarnström and Doherty 2000). However, such planners do not generate partial-order plans, which for us is an end in itself. This leads to the question of whether some aspects of forward-chaining could be used in the generation of partial-order plans, combining the flexibility of one with the performance of the other – for example through the generation of stronger state information for partial-order plans.

Related work. The idea of combining these types of planning is not new. For example, FLECS (Veloso and Stone 1995) uses means-ends analysis to add relevant actions to a partial-order plan. A FLExible Commitment Strategy then determines when an action should be moved to the end of a totally ordered plan prefix, in essence updating the initial state. As sequential execution is assumed, committing to arbitrary total orderings is not considered harmful if it improves planning performance.

A planner by Brenner (2003) uses a PO plan structure whose *frontier* contains those state variable assignments that are "achieved", meaning that they occur in the plan and cannot be interfered with. As only actions whose preconditions hold in the frontier are considered applicable, it can efficiently be determined which actions can be added to the current plan. Each new action is placed after the "achievers" supporting its precondition, and additional constraints are added to ensure potential threats are resolved, resulting

[1] We assume basic familiarity with POCL planning and refer to e.g. Weld (1994) for an overview of concepts and terminology.

in a new valid plan. The algorithm is sound but incomplete.

Coles et al. (2010) independently developed a similar approach in POPF, also keeping track of a completely defined frontier state and placing new actions after the achievers supporting their preconditions. However, POPF is complete, integrates interesting features such as continuous linear change, and provides effective domain-independent heuristics.

The TFPOP approach. Some time ago, we began exploring another approach to making use of forward-chaining when generating partially ordered plans (Kvarnström 2010). This approach has a stronger focus on execution flexibility for multiple agents and is therefore in several ways closer to standard partial-order planning than either of the planners above. For example, it neither uses a total-order prefix nor keeps track of a completely defined frontier state, but retains a partially ordered plan structure at all times and allows any action in a plan to support new causal links.

Creating highly informative states from a partial-order plan does require *some* form of additional structural constraint, though. The key is that for flexible execution, partial ordering is considerably more important between different agents than within the subplan assigned to any given agent: Each UAV can only perform its flight and delivery actions in sequence, as it cannot be in several places at once. Generating the actions for each agent in sequential temporal order would be a comparatively small sacrifice, allowing actions belonging to distinct agents to remain independent to exactly the same extent as in a standard partial order plan. Each agent-specific *thread* of actions can then be used to generate informative agent-specific states in a forward-chaining manner, resulting in a Threaded Forward-chaining Partial Order Planner called TFPOP.

In particular, many state variables are associated with a specific agent and are only affected by the agent itself. This holds for the location of an agent (unless some agents can move others) and for the fact that an agent is carrying a particular object (unless one agent can "make" another carry something). Complete information about such state variables can easily be generated at any point along an agent's sequential thread and is in fact particularly useful for the agent itself. For example, whether a given UAV can fly to a particular location depends on its own current location and fuel level, not those of other agents. Additionally, we have complete information about static facts such as the location of a depot and the carrying capacity of a carrier.

This usually results in sufficient information to quickly determine whether a given precondition holds, thus allowing TFPOP to very efficiently rule out most inapplicable actions. Agents are also often comparatively loosely coupled (Brafman and Domshlak 2008): Not every action requires an interaction with other agents. Therefore, agents will for extended periods of time mostly depend on state variables that are not *currently* affected by other agents. This is not crucial for the approach to be applicable but does further strengthen the available information. A fallback procedure reminiscent of the operation of a standard POCL planner takes care of the comparatively rare cases where states are too weak.

Partial states could be used to support new state-based heuristics for PO planning. In this paper, we instead use them to evaluate precondition control formulas (Bacchus and Ady 1999) which have previously proven highly effective in pruning the search space for sequential planning. The result is a new knowledge-rich partial-order planner competitive with the sequential versions of the award-winning planners TLPLAN (Bacchus and Kabanza 2000) and TALplanner (Kvarnström and Doherty 2000) in terms of performance.

We will now present TFPOP. We first describe planning domains, problem instances and plan structures. We then give a brief high-level description of the planner and its search space before discussing the algorithms involved in detail. Finally, we discuss soundness and completeness and provide an empirical evaluation before concluding the paper.

2 Domains, Problems, Plans, Solutions

We assume a typed finite-domain state-variable or *fluent* representation. For example, loc(*crate*) might be a location-valued fluent taking a crate as its only parameter.

An *operator* has a list of typed parameters, where the first parameter always specifies the executing *agent*. For example, flying between two locations may be modeled using the operator fly(*uav*, *from*, *to*), where the *uav* is the executing agent. An *action* is a fully instantiated (grounded) operator.

To simplify the presentation, an agent has a single *thread* of execution. The extension to multiple threads is trivial.

Each operator o has a *precondition* formula $pre(o)$ that may be disjunctive and quantified. We currently assume all *effects* are conjunctive and unconditional and take place in a single effect state at the end of an action.

Action *durations* often depend on circumstances outside the control of a planner or execution mechanism: The time required for flying between two locations may depend on wind, temporary congestion at a depot, and other factors. We therefore model *expected* action durations $expdur(a) > 0$ but never use such durations to infer that two actions will end in a specific order. Each duration is specified as a temporal expression that may depend on operator arguments.

Many domains involve mutual exclusion, as when only one UAV can refuel at a time. To explicitly model this, each operator is associated with a set of binary parameterized *semaphores* that are acquired throughout the execution of an action. For example, use-of(*location*) may be a semaphore ensuring exclusive access to a particular location. The set of semaphores acquired by an action a is denoted by $sem(a)$.

For any problem instance, the *initial state* must completely define the values of all fluents. *Goal formulas*, like preconditions, may be disjunctive and quantified.

2.1 Plan Structures

A *plan* is a tuple $\pi = \langle A, L, O, M \rangle$, where:

- A is the set of all *action instances* occurring in the plan.
- L is a set of ground *causal links* $a_i \xrightarrow{\mathsf{f}=v} a_j$ representing the commitment that a_i will achieve the condition $\mathsf{f} = v$ for a_j.
- O is a set of *ordering constraints* on A whose transitive closure is a partial order denoted by $\preceq_O$. The index of $\preceq$ will be omitted when obvious from context. As usual, $a_i \prec a_j$ iff $a_i \preceq a_j$ and $a_i \neq a_j$. The fact that $a \prec b$ means that a ends strictly before b begins.

- M is a set of *mutex sets*, where no two actions belonging to the same mutex set can be executed concurrently (maybe due to explicit semaphores or because their effects interfere) but whose order is not necessarily defined before execution. This enhances execution flexibility compared to a pure partial order where such actions must be ordered at plan time. $M(a)$ denotes the possibly empty set of actions with which a is mutex according to M.

As each action instance specifies the thread it is associated with, explicit separation into thread-specific action sets is unnecessary. Instead, $act(\pi, t)$ denotes the set of action instances executed by a given thread t and we add the *structural constraint* that such sets must be totally ordered by O.

Though we may use expected durations to infer approximately when an action is *expected* to start, actions are not actually scheduled to execute at a specific timepoint. Instead, they can only be invoked when explicitly notified that all predecessors have finished, which may take longer than initially expected. Therefore the PDDL2.1 issue of non-zero action separation (Fox and Long 2003) does not arise. Note also that plans with this structure are always dynamically controllable (Morris, Muscettola, and Vidal 2001).

Executability. To determine whether a plan $\langle A, L, O, M\rangle$ is executable, we must consider all possible orders in which it permits (potentially concurrent) actions to start and end. Each action $a \in A$ is therefore associated with an *invocation node* $inv(a)$ where preconditions must hold and semaphores are acquired, and an *effect node* $eff(a)$ where effects are guaranteed to have taken place and semaphores are released. Action precedence carries over directly to such nodes: If $a \in A$ then $inv(a) \prec eff(a)$, and if $a \prec b$ then $eff(a) \prec inv(b)$.

The *node sequences* that may arise from execution are exactly those that (a) are consistent with $\prec$ and (b) ensure that no two actions that are mutually exclusive according to M are executed concurrently. Durations cannot be used to infer additional necessary orderings, only *probable* orderings.

A plan is executable iff all associated node sequences $[n_0, n_1, \ldots, n_k]$ satisfy the following constraints. First, for every pair of distinct actions $\{a_i, a_j\} \subseteq A$ whose execution overlaps in the given node sequence:

- $sem(a_i) \cap sem(a_j) = \emptyset$: Two concurrently executing actions cannot require the same semaphore.
- $affected(a_i) \cap affected(a_j) = \emptyset$, where $affected(a)$ is the set of fluents occurring in the effects of a: Two concurrently executing actions cannot affect the same fluent.
- $affected(a_i) \cap required(a_j) = \emptyset$, where $required(a)$ is the set of fluents represented in incoming causal links to a: No action can affect a fluent used to support the precondition of another concurrently executing action.

Second, the effects of any action $a \in A$ must be internally consistent. Third, all preconditions must be satisfied: Let s_0 be the initial state and for all $0 < i \leq k$, let s_{i+1} be

$$\begin{cases} s_i \text{ modified with the effects of } n_i & \text{if } n_i \text{ is an effect node,} \\ s_i & \text{otherwise.} \end{cases}$$

Then for every invocation node n_i corresponding to an action a, we must have $s_i \models pre(a)$.

Solutions. An executable plan is a *solution* iff every associated node sequence results in a state s satisfying the goal.

3 The TFPOP Planner

We now turn our attention to how to find a solution, starting with an overview of the TFPOP search space, its use of partial states and the planner itself. In the subsequent sections, the algorithms involved will be discussed in detail.

Search Space. The *initial node* in the search space is the executable plan $\langle\{a_0\}, \emptyset, \emptyset, \emptyset\rangle$, where a_0 is a special initial action not associated with a specific thread. As in POCL planning, this action has no preconditions or semaphores and its effects define the initial state. For all other $a_i \in A$ in any plan, $a_0 \prec a_i$. As means-ends analysis is not used, the initial plan does not contain an action representing the goal.

A *successor* is a new executable plan π where exactly one new action has been added to the end of a specific thread t (occurring last in $act(\pi, t)$), possibly also constrained to occur before or after certain actions in other threads. Thus, unlike POCL planners, TFPOP does not allow intermediate plans to have "flaws" such as unsatisfied preconditions. Allowing flaws would weaken state information, and as all flaws must be corrected in the final solution, delaying their correction would not increase execution flexibility. As shown in Section 5, this does not affect completeness.

Partial States. Our intention is to improve performance by generating informative states to be used in precondition evaluation. Each action in a plan is therefore associated with a state specifying facts known to be true from the instant the action ends until the start of the next action in the same thread, or "forever" if there is no next action yet. The initial action is a special case, belonging to no thread and providing an initial state for *all* threads. Section 3.4 shows how states are generated and updated when new actions are added.

In general, one cannot infer complete information about any point during the execution of a partially ordered plan. Therefore, a formula can evaluate to *true*, *false*, or *unknown*.

Also, ensuring states contain all information that *can* theoretically be inferred from a partial order requires comparatively complex state structures and state update procedures. This tends to require more time than is saved by rapid evaluation. Formula evaluation is therefore permitted to return *unknown* even when it would be theoretically possible to determine whether a formula holds at a certain point in a plan, and a complete but more complex fallback procedure is called for any *unknown* formula (*make-true*, see below).

As maximal state information is not required, a *partial state* can be a simple structure specifying a finite set of possible values for each fluent ($\mathsf{f} \in \{v_1, \ldots, v_n\}$). The evaluation procedure $eval(\phi, s)$ is the natural extension of a standard recursive formula evaluator to three truth values. For example, suppose ϕ is $\alpha \wedge \beta$. If $eval(\alpha, s) = false$ or $eval(\beta, s) = false$, then $eval(\phi, s) = false$. If both subformulas are *true*, the conjunction is *true*. Otherwise, the conjunction is *unknown*. This combination has proven to yield a good balance between information and speed, representing most of the useful information that can be inferred but also supporting very efficient state updates and formula evaluation.

The Planner. First, the planner (pseudocode outline below) tests whether the goal is already satisfied (Section 3.5). If so, a solution is returned. If not, a successor must be found.

By definition, any successor of a node adds a single new action a to the end of an existing thread t. The planner therefore begins by determining which thread t to extend (Section 3.1). The last state associated with this thread specifies facts that must hold when a is invoked and can therefore be used to evaluate the preconditions of potential candidates.

The result may be *false*, in which case a candidate has efficiently been filtered out. Otherwise we check the action's effects for consistency. We then call *make-true* to search for precedence constraints and causal links ensuring the precondition will hold (Section 3.2), which might fail if the precondition was *unknown*, and that there is no interference between a and existing actions in the plan (Section 3.3).

Resource usage is updated and resource constraints are checked (Section 4). If no constraints are violated, we determine which existing actions $a' \in A$ share a semaphore or an affected fluent with the new action a and update the set M of mutex sets to ensure such actions cannot be executed in parallel with a. As this cannot fail, a new successor has been found. A partial state is generated for the new action and existing states may have to be updated due to potential interference (Section 3.4). This results in the following "skeleton" planner, where **choose** refers to standard non-deterministic choice, in practice implemented by backtracking.

procedure TFPOP
$\pi \leftarrow \langle\{a_0\}, \emptyset, \emptyset, \emptyset\rangle$ *// Initial plan*
repeat
  if goal satisfied **return** π
  choose a thread t to which an action should be added
  // Use partial state to filter out most potential actions
  $s \leftarrow$ partial state at the end of thread t
  choose an action a for t such that $pre(a)$ is not *false* in s
  if effects of a are inconsistent **then fail** (backtrack)
  // Complete check: Can the action really be added, and how?
  choose precedence constraints C and causal links L ensuring
    $pre(a)$ is satisfied, a does not interfere with existing actions
    and no existing action interferes with a
  update resource usage
  if resource constraints violated **then fail** (backtrack)
  add a, C, L and necessary mutex sets to π
  update existing partial states and create new partial state for a

Search Guidance. TFPOP is currently not guided by heuristics but by precondition control formulas (Bacchus and Ady 1999). Such formulas represent conditions that are required not for executability but for an action to be meaningful in a given context, and have been used to great effect in TLPLAN (Bacchus and Kabanza 2000) and TALplanner (Kvarnström and Doherty 2000).

Though any planner supports preconditions, most PO planners use means-ends techniques where stronger preconditions may generate subgoals but provide no guidance. TFPOP requires immediate support for all preconditions, yielding efficient and effective pruning of the search space.

Control formulas often need to refer to the goal of the current problem instance. For example, this is needed to determine where the crates on a carrier should be delivered, so UAVs can be prevented from flying the carrier to other locations – which would be possible but pointless. TFPOP therefore supports the construct $\mathsf{goal}(\phi)$ (Bacchus and Kabanza 2000), which is used in preconditions to test whether ϕ is entailed by the goal.

3.1 Selecting a Thread to Extend

By default, the planner aims to minimize makespan by distributing actions as evenly as possible across available threads. It therefore prefers to extend threads whose current actions are expected to finish earlier given the assumption that each action will start as soon as its explicit predecessors have finished and it can acquire the required semaphores.

This prioritization can be done by calculating the expected start time $expstart(a)$ and finish time $expfin(a)$ for every action $a \in A$ using the procedure below. The implementation uses a variation where times are updated incrementally as new actions are added. Again, timing cannot be used to infer that one action *must* occur before another.

procedure *calculate-expected-times*($\pi = \langle A, L, O, M\rangle$)
$expstart(a_0) = expfin(a_0) = 0$ *// Initial action a_0 at time 0*
while some action in A remains unhandled
  $E \leftarrow \{a \in A \mid$ all parents of a are handled$\}$ *// Executable now*
  // We know $E \neq \emptyset$, since the partial order is non-cyclic.
  // Calculate for each $a \in E$ when its parents should be finished.
  // If several actions could start first, break ties arbitrarily.
  forall $a \in E$: $t(a) = \max_{p \in parents(e)} expfin(p)$
  $a \leftarrow$ an arbitrary action in E minimizing $t(a)$ *// Could start first*
  $expstart(a) \leftarrow t(a)$; $expfin(a) \leftarrow expstart(a) + expdur(a)$
  forall unhandled $a' \in M(a)$: $O \leftarrow O \cup \{a \prec a'\}$

The final step temporarily modifies O to ensure that when one action acquires a particular mutually exclusive resource, it will strictly precede all other actions that require it but have not yet acquired it.

3.2 Satisfying Preconditions

When a thread t has been selected, each potential action a for the thread is considered in turn. For a to be applicable, it must first be possible to satisfy its preconditions.

Let a_L be the last action currently in the thread t (with $a_L = a_0$ if t is empty), and let s be the last state in t. If $eval(pre(a), s) = false$, then a cannot be applicable and we can immediately continue to consider the next action.

If *unknown* is returned, the reason may be that $pre(a)$ requires support from an effect that (given the current precedence constraints) may or may not occur before a, that there is potential interference from other actions, or simply that the state was too weak to determine whether $pre(a)$ holds. As we do not know, we must test whether we can introduce new precedence constraints that ensure $pre(a)$ holds. Even if the precondition was *true*, guaranteeing that support can be found, we must still generate new causal links to protect the precondition from interference by actions added later.

A provisional plan $\pi' = \langle A \cup \{a\}, L, O \cup \{a_L \prec a\}, M\rangle$ is created, where a is placed at the end of its thread. Then, $make\text{-}true(\phi, a, s, \pi')$ determines whether it is possible to guarantee that $\phi = pre(a)$ holds when a is invoked in π' in

the partial state s. A set of *extended plans* is returned, each extending π' with links and possibly constraints corresponding to one particular way of ensuring that ϕ will hold. If this is impossible, the empty set of extensions is returned.

procedure *make-true*$(\phi, a, s, \pi = \langle A, L, O, M\rangle)$
if ϕ is $v_1 = v_2$
  if $v_1 = v_2$ **return** $\{\pi\}$ **else return** $\emptyset$
else if ϕ is goal(ϕ)
  if goal state $\models \phi$ **return** $\{\pi\}$ **else return** $\emptyset$
else if ϕ is $\mathsf{f} = v$
  if $s \models \mathsf{f} \neq v$ **return** $\emptyset$ *// Use state for efficiency*
  // Find potential supporters a_i that may be placed before a
  $X \leftarrow \emptyset$
  forall $a_i \in A$ **such that** a_i assigns $\mathsf{f} = v$ and not $a \prec a_i$
    // Add supporter, find ways of fixing interference (see below)
    $\pi' \leftarrow \langle A, L \cup \{a_i \xrightarrow{\mathsf{f}=v} a\}, O \cup \{a_i \prec a\}, M\rangle$
    $X \leftarrow X \cup$ *fix-interference*$(\mathsf{f}, a_i, a, \pi')$ *// See next section*
  return X
else if ϕ is $\neg(\mathsf{f} = v)$ *// Handled similarly*
else if ϕ is $\neg\alpha$
  Push negation inwards using standard equivalences
  such as $\neg(\alpha \wedge \beta) = \neg\alpha \vee \neg\beta$ and recurse
else if ϕ is $\alpha \wedge \beta$
  // For each way of satisfying α, find all ways of also satisfying β
  $X \leftarrow \emptyset$
  forall $\pi' \in$ *make-true*(α, a, s, π)
    $X \leftarrow X \cup$ *make-true*(β, a, s, π')
  return X
else if ϕ is $\alpha \vee \beta$
  // Find all ways of satisfying either α or β
  return *make-true*$(\alpha, a, s, \pi) \cup$ *make-true*(β, a, s, π)
else if ϕ is quantified
  Instantiate with all values of the variable's finite domain
  and handle as a conjunction or disjunction
end if

3.3 Avoiding Interference

As shown above, when *make-true* finds a possible supporter a_i for an atomic formula $\mathsf{f} = v$, it is placed before a and a causal link is added. The *fix-interference* procedure then determines if and how we can guarantee that no other actions can interfere by also assigning a value to f between the end of a_i and the start of a. It iterates over all potential interferers a', adding (if permitted by π) constraints that place a' either before a_i or after a. After each iteration, X contains all minimally constraining ways of avoiding interference from all actions considered so far (and possibly some that are more constraining than necessary, which could be filtered out).

procedure *fix-interference*$(\mathsf{f}, a_i, a, \pi = \langle A, L, O, M\rangle)$
$X \leftarrow \{\pi\}$
forall $a' \in A \setminus \{a, a_i\}$ assigning a value to f
  if $a' \prec_O a_i$ or $a \prec_O a'$ **then** a_i cannot interfere **else**
    $X' \leftarrow \emptyset$ *// Then, for every solution to the earlier interferers*
    forall $\pi' = \langle A', L', O', M'\rangle \in X$
      // Can we place a' before the supporter a_i?
      if not $a_i \prec_{O'} a'$ **then** $X' \leftarrow X' \cup \{\langle A', L', O' \cup \{a' \prec a_i\}, M'\rangle\}$
      // Can we place a' after a?
      if not $a' \prec_{O'} a$ **then** $X' \leftarrow X' \cup \{\langle A', L', O' \cup \{a \prec a'\}, M'\rangle\}$
      // Note that both extensions may be impossible,
      // leading to a reduction of the size of X!
    $X \leftarrow X'$
return X

This ensures no actions in π can interfere with the preconditions of a, but a may also interfere with existing actions in π. That is handled by identifying all actions $a' \in A$ whose incoming causal links depend on fluents f affected by a and then preventing interference using *fix-interference*.

3.4 Generating and Updating Partial States

A partial state represents facts that hold during a certain interval of time. If the effects of a new action may occur within that interval, the state must be **updated** and "weakened" to remain sound. Suppose the state s associated with a_1 = fly(uav8,depot1,loc12) claims that loc(uav5)$\in\{$depot4$\}$: Given the current plan, this fact *must* hold from the end of a_1. Suppose further that a = fly(uav5,depot4,loc57) is added, and that its effects may occur within the interval of time where s should be valid. Then, s must be updated to claim that throughout its interval, loc(uav5) takes on one of two possible values: loc(uav5)$\in\{$depot4,loc57$\}$.

As only soundness is required, updates do not have to yield the *strongest* information possible. Thus, while one could test precedence constraints to determine exactly which states the new action may interfere with, one can also simply weaken *all* existing states: If a state claims that $\mathsf{f} \in V$ and the new action has the effects $\mathsf{f} := v$, the state would be modified to claim $\mathsf{f} \in V \cup \{v\}$. For increased efficiency, we note that only the last state in each thread is used for precondition evaluation. Consequently only those states need to be weakened to ensure soundness.

When a new action a is added, we must also **generate a new state** valid from the end of that action until infinity. It is clear that the state s of its predecessor in the same thread is valid when the action is invoked. We therefore begin by making a copy s' of s. Given the new precedence constraints, we also know that any condition $\mathsf{f} = v$ supported by a new causal link $a' \xrightarrow{\mathsf{f}=v} a$ must hold when a is invoked. We therefore strengthen s' with the knowledge that $\mathsf{f} = v$. New information thus flows between threads along causal links, counteracting the weakening discussed above and resulting in a stronger state valid exactly when a is invoked.

To create a state valid from the *end* of a until infinity, we also apply all effects of a to s', resulting in a state valid exactly when a ends. Finally, we weaken this state using all effects that may possibly occur in other threads from a until infinity, in a procedure essentially identical to the state update above. The result is a new sound state for a.

3.5 Testing Goal Satisfaction

The goal is satisfied iff it holds after all actions in the plan have finished. The last state of each thread t contains facts known to be true from the end of the last action in t until infinity, and consequently also after *all* actions have finished. Conjoining the final states of all threads results in even stronger information about what is true after all actions finish. If evaluating the goal formula in this state returns *false*, we have efficiently detected that it cannot be satisfied.

If evaluation returns *true* or *unknown*, a pseudo-action a_g is created with the goal as precondition and constrained to occur after the last action in all threads. Then, *make-true* is called to determine whether new precedence constraints can be introduced to arrange existing actions in such a way that the goal will definitely hold. If so, a solution is found. If not, the current plan cannot be a solution.

3.6 Some Efficiency Considerations

The *make-true* procedure may appear to investigate a combinatorial explosion of alternatives for conjunctive formulas: For every way of satisfying α, there may be many ways of satisfying β. In practice, though, there is often only one way of satisfying a subformula while retaining a consistent precedence relation, or even none at all. The same is true for *fix-interference*. Note also that POCL planners need to perform similar steps, though possibly spread out over time.

Furthermore, we have described the planning process as if the precondition of each action instance is evaluated separately. The implementation instead splits the precondition of any operator into the conjuncts p_0 in which no parameters occur, p_1 in which only the first parameter occurs, and so on. Preconditions are then evaluated using nested loops, a common approach in planners that do not pre-ground actions, permitting large subsets of inapplicable action instances to be detected with a single formula evaluation.

Finally, these procedures are implemented to yield a solution or applicable action when it is found rather than generating all solutions at once, saving their state for future calls.

4 Extension: Resource Constraints

Since TFPOP is based on state variables, it can easily support resources modeled as numeric fluents: Rather than modeling each crate as a separate object and keeping track of which crates are on a certain carrier, we model how *many* crates of a given type are loaded. Not only is this closer to how we think about the domain, since crates of a given type are interchangeable – it also reduces the branching factor.

Resources can often be produced or consumed in parallel, as when ground robots cooperate to load a carrier. Using ordinary preconditions crates(carrier,type) = n and effects crates(carrier,type) := $n + 1$ for resources leads to sequentialization of resource effects, since exact resource amounts must be known before and after the action. This is highly undesirable as it decreases execution flexibility and removes all pressure on the planner to use multiple agents to load a carrier. We therefore add specific support for numeric resource fluents and effects: *increase*(crates(carrier,type), 1).

Each resource instance r has an upper bound $u(r)$, such as the number of crates permitted on a carrier, and a lower bound $l(r)$. After adding an action, the planner must determine if the bounds can be satisfied at all times, and if so, which precedence constraints may need to be added.

For maximum expressivity, we can adapt existing general results such as (Policella et al. 2007). Initially, we instead exploit the common case where resource production and consumption occurs in *phases*. For example, crates(carrier,type) is only produced when crates are loaded at a depot. The carrier must be fully loaded (all actions in the production phase must be finished) before it is flown to a destination and the unloading phase begins. Thus, each phase can be unordered "internally" for flexibility but it is inherent in the domain that one cannot *both* produce and consume resources in parallel during the same time period.

We can thus require all actions in a production phase to be constrained to occur strictly before all actions in a subsequent consumption phase and vice versa. This semi-structured approach is simple and highly efficient, yet retains the flexibility to execute the actions within each phase concurrently or in arbitrary order.

Each resource instance r is therefore associated with its own sequence $[p_0, \ldots, p_n]$ of production and consumption phases. When a new action is added, the planner checks whether the initial amount available at the start of the current phase plus what is produced within the phase exceeds the permitted maximum. If so, no introduction of precedence orderings can make the plan executable, and backtracking is required. Conversely, if the bound is not exceeded, any execution ordering compatible with the plan satisfies resource constraints. Consumption is handled equivalently.

5 Soundness and Completeness

A detailed formal soundness and completeness proof for TFPOP requires several pages. Instead, we provide a proof sketch outlining the major aspects of the proof.

First, TFPOP is **sound**.

Proof sketch. The initial plan is clearly executable. An action a is only added to a plan π if *make-true* has first found explicit support for its precondition in π, ensured that the supporting actions precede a, and ensured that no existing actions in π can interfere with the supporter. Thus, $pre(a)$ is guaranteed to hold when a is added. The procedure also adds causal links ensuring support cannot be destroyed in the future, and a procedure similar to *fix-interference* ensures that a does not interfere with causal links for existing actions in the plan. Thus, adding a to an executable plan results in a new plan where all preconditions of a are satisfied and no preconditions of existing actions are disrupted.

The planner constructs mutex sets ensuring that no two actions acquiring the same semaphore or affecting the same fluent can occur concurrently. Resource constraints are also tested, and action effects are tested for internal consistency. These are all the requirements for a plan to be executable, and consequently TFPOP only creates executable plans.

Goal testing uses *make-true*, which again is sound and only succeeds if the goal is truly satisfied. Therefore, any plan actually returned is a solution. □

To show completeness, we first show that because there are no circular dependencies between actions, any valid plan can be generated through the incremental addition of actions to the empty plan with each intermediate step being executable. Conversely, any valid plan can be reduced to the initial plan with each intermediate step being executable.

Proof sketch. Let $\pi = \langle A, L, O, M \rangle$ be an executable plan where $A \neq \emptyset$. Let $a \in A$ be an action that has no descendants: There is no $b \in A$ such that $a \prec b$. Such an a must

exist, or precedence would be circular and not a partial order.

An action can only support the preconditions of its explicit descendants. Since a has none, removing it cannot remove support from another action. Neither can a be needed to satisfy resource constraints, as there is no descendant relying on its effects. Finally, removing a can lead to fewer semaphores being acquired, which cannot make a previously executable plan unexecutable.

Removing a with associated links and constraints from π must lead to a new executable plan π'. Inductively, any executable plan can be reduced to the initial plan by a sequence of such steps, each resulting in an executable plan. □

There are permissible single-action extensions that TFPOP will never generate, since it is permissible to add unnecessary constraints, which we naturally try to avoid. However, we can show that for every permissible extension, TFPOP can find one that is *less or equally* constraining.

Proof sketch. Let $\pi = \langle A, L, O, M\rangle$ be an executable plan and let $\pi' = \langle A \cup \{a\}, L \cup L', O \cup O', M \cup M'\rangle$ be any permissible single-action extension. Since π' is executable, $pre(a)$ is satisfiable in π. Then a will not be pruned when preconditions are tested: The formula evaluator is correct, and all partial states are sound since the initial state is correct and both state update and state generation procedures are sound. The result of *make-true* includes all minimally constraining ways of supporting an action's preconditions, and similarly for interference avoidance. The planner generates the unique minimally constraining mutex sets $M'' \subseteq M'$. Resource constraint checking is sound. Therefore the planner can find suitable links $L'' \subseteq L'$ and constraints $O'' \subseteq O'$ so that $\pi'' = \langle A \cup \{a\}, L \cup L'', O \cup O'', M \cup M''\rangle$ is an executable plan less or equally constraining than π'. □

Using weaker constraints when an action is added can never reduce the options available at the next step. Therefore the argument above can be extended: If π is a complete plan, then TFPOP can extend the initial plan to an executable plan π' less or equally constraining than π. As the branching factor is finite, a complete search procedure such as iterative deepening will eventually find such a plan.

We can now finally show that TFPOP is **complete**.

Proof sketch. Suppose π is a solution. Given a complete search procedure, TFPOP will find an executable plan π' less or equally constraining than π. Since the goal would be satisfied after the execution of π, it must be possible to satisfy it by the introduction of new precedence constraints in π'. The state-based goal filter test is sound and will not return *false*. The goal test based on *make-true* is sound and complete. Thus, the required precedence constraints will be found and a solution will be returned. □

6 Evaluation

We now turn to the empirical evaluation, which is explicitly intended to answer two specific questions.

How effective are partial states? The states generated by TFPOP cannot be expected to provide perfect information about the applicability of actions, as in planners constraining their plan structures to generate complete states. However, it has been our hypothesis that the information provided would enable the planner to quickly filter out at least 95% of all inapplicable actions using only partial states.

To test this hypothesis, we implemented a set of temporal benchmark domains from planning competitions as well as the UAV delivery domain used previously in this paper. We then introduced some simple yet effective domain-specific control for each domain. Finally, we measured the percentage of inapplicable actions found by precondition evaluation in partial states for a variety of problem instances.

For several domains from the third International Planning Competition (IPC-3), including *Satellite* and *ZenoTravel*, all of the larger "handcoded" problems used in the competition resulted in at least 99% of all inapplicable actions being detected. For our *UAV delivery* domain, we generated 1440 random problems with 32 to 512 crates, 4/8/16 UAVs, 4/8/16 crates per destination, one carrier per UAV, and 8 ground robots. This domain is designed to be more difficult by providing greater interaction among agents than Satellite and ZenoTravel: For each carrier a UAV delivers, it is dependent on one or more ground robots to load it, and a ground robot cannot load a carrier until a UAV has returned it. Despite the resulting weakening of partial states, 96%-97% of all inapplicable action instances are detected using partial states for the smaller problem instances, increasing to over 99% for the larger instances.

In summary, we consider the hypothesis validated.

What is the penalty of using partial orders? Even if all inapplicable actions were detected early, processing the *applicable* actions requires searching for precedence constraints and causal links, updating partial states, and maintaining a full partial-order plan structure. This could in theory require so much time that the approach would be infeasible. Our hypothesis has been that this would not be the case, and that procedures such as *make-true* would be efficient in practice.

To test this, we must isolate the impact of the plan structure from unrelated issues such as using different heuristics. We therefore translated the Satellite, ZenoTravel and UAV Delivery domains used above into the languages used by two planners based on control formulas with *complete states*: The sequential versions of TLPLAN (version "April 3rd 2006 0.1") and TALplanner (revision 3721). Testing was performed on a 2.4 GHz Core 2 Duo computer with 4 GB of memory, using the JDK 6 update 23 runtime for TALplanner and TFPOP which are written in Java.

Figure 1 shows that TFPOP is competitive despite its more complex plan structure, the differences between the planners being sufficiently small that they may be due to low-level implementation issues. This shows that using partial-order plan structures and partial states in knowledge-rich planning can have a minimal performance penalty.

Comparison with POPF. Since POPF is also built on the idea of combining partial-order and forward-chaining planning, a comparison is in order. However, a direct *performance* comparison would not be very illuminating. Without control formulas, the current version of TFPOP is not goal-directed and cannot compete. With control formulas, POPF cannot compete, as it spends a significant amount of time calcu-

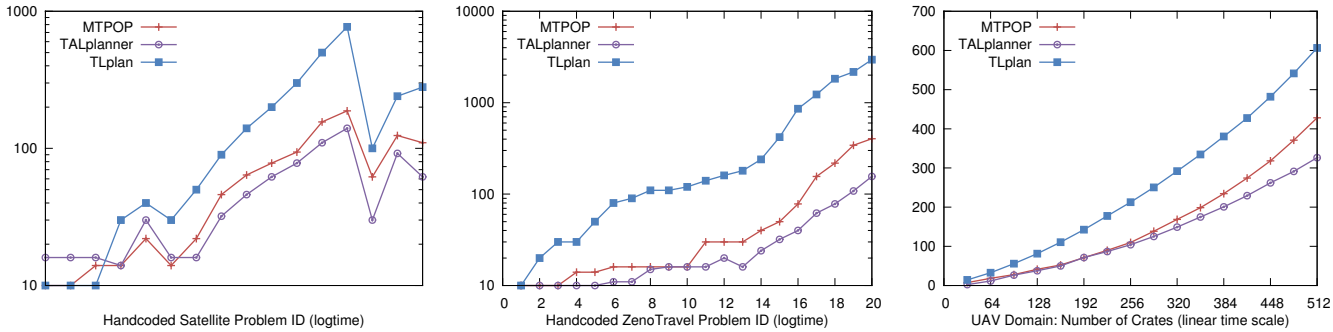

Figure 1: Comparison between TFPOP, TLPLAN and TALplanner. Times in milliseconds.

lating sophisticated heuristics that are no longer required in order to find a solution. If these calculations could be turned off, then who is faster given identical control formulas is likely to depend to a great extent on who happens to have the fastest low-level implementation of state structures and formula evaluation. Therefore, we will focus on the fundamental differences between the two search spaces and how this affects the planners.

In this respect, POPF is closer to standard forward-chaining in its assumption of complete states. This has several positive effects: The planner can take advantage of standard and extended heuristics assuming completeness, it can more easily integrate facilities for linear constraint solving, and it has no need to search plan structures to generate causal links.

The TFPOP approach is closer to standard partial-order planning, which improves execution flexibility compared to POPF: Actions that require the same semaphore or affect the same fluent need not be ordered before execution, and actions may produce or consume non-exclusive resources concurrently without serializing resource access during planning. Directly supporting this type of concurrency further increases the pressure on the planner to spread actions among multiple agents when possible, which can improve makespan. Using a plan structure that is not fundamentally based on complete states also provides a basis for extensions to planning with incomplete information.

7 Conclusions

We have presented an approach to taking advantage of certain aspects of forward-chaining in the generation of partially ordered multi-agent plans with a high degree of execution flexibility. By making use of the fact that agents often perform their own actions in strict sequence, a partially ordered plan can be annotated with partial but strong states used to filter out most inapplicable actions. The resulting planner uses expressive precondition control formulas for guidance, and despite using incomplete states and generating partially ordered plans, it is competitive with the sequential versions of the knowledge-rich planners TLPLAN and TALplanner in terms of performance.

In the future, we intend to investigate extensions for required concurrency as well as controllable and uncontrollable action durations (Morris, Muscettola, and Vidal 2001).

Acknowledgments

This work is supported by grants from the Swedish Research Council, CENIIT, the ELLIIT network organization for Information and Communication Technology, the Swedish National Aviation Engineering Research Program (NFFP5), and the Linnaeus Center for Control, Autonomy, Decision-making in Complex Systems (CADICS). We are grateful to Mikael Nilsson for assistance in empirical testing.

References

Bacchus, F., and Ady, M. 1999. Precondition control. http://www.cs.toronto.edu/~fbacchus/Papers/BApre.pdf.

Bacchus, F., and Kabanza, F. 2000. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence* 116(1-2):123–191.

Bäckström, C. 1998. Computational aspects of reordering plans. *Journal of Artificial Intelligence Research* 9(99):137.

Brafman, R., and Domshlak, C. 2008. From one to many: Planning for loosely coupled multi-agent systems. In *Proc. ICAPS*.

Brenner, M. 2003. Multiagent planning with partially ordered temporal plans. Technical report, Institut für Informatik, Universität Freiburg.

Coles, A.; Coles, A.; Fox, M.; and Long, D. 2010. Forward-Chaining Partial-Order Planning. In *Proc. ICAPS*.

Fox, M., and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research* 20(1):61–124.

Kvarnström, J., and Doherty, P. 2000. TALplanner: A temporal logic based forward chaining planner. *Annals of Mathematics and Artificial Intelligence* 30:119–169.

Kvarnström, J. 2010. Planning for loosely coupled agents using partial order forward-chaining. In *Proc. SAIS*.

Morris, P.; Muscettola, N.; and Vidal, T. 2001. Dynamic control of plans with temporal uncertainty. In *Proc. IJCAI*.

Policella, N.; Cesta, A.; Oddi, A.; and Smith, S. 2007. From precedence constraint posting to partial order schedules: A CSP approach to Robust Scheduling. *AI Communications* 20(3).

Sacerdoti, E. D. 1975. The nonlinear nature of plans. In *Proc. IJCAI*, 206–214.

Veloso, M., and Stone, P. 1995. FLECS: Planning with a flexible commitment strategy. *JAIR* 3:25–52.

Weld, D. S. 1994. An introduction to least commitment planning. *AI magazine* 15(4):27.