Automated Planning for Collaborative UAV Systems

Jonas Kvarnström and Patrick Doherty
Department of Computer and Information Science
Linköping University, SE-58183 Linköping, Sweden
{jonkv,patdo}@ida.liu.se

***Abstract*—Mission planning for collaborative Unmanned Aircraft Systems (UAS:s) is a complex topic which involves trade-offs between the degree of centralization or decentralization required, the degree of abstraction in which plans are generated, and the degree to which such plans are distributed among participating UAS:s. In realistic environments such as those found in natural and man-made catastrophes where emergency services personnel are involved, a certain degree of centralization and abstraction is necessary in order for those in charge to understand and eventually sign off on potential plans. It is also quite often the case that unconstrained distribution of actions is inconsistent with the loosely coupled interactions and dependencies which arise between collaborating systems. In this article, we present a new planning algorithm for collaborative UAS:s based on combining ideas from forward chaining planning with partial-order planning leading to a new hybrid partial order forward-chaining (POFC) framework which meets the requirements on centralization, abstraction and distribution we find in realistic emergency services settings.**

***Index Terms*—Partial-order planning, unmanned aerial vehicles, planning with control formulas**

I. Introduction

A devastating earthquake has struck in the middle of the night. Injured people are requesting medical assistance, but clearing all roadblocks will take days. There are too few helicopters to immediately transport medical personnel to all known wounded, and calling in pilots will take time. Fortunately, we also have access to a fleet of unmanned aerial vehicles (UAVs) that can rapidly be deployed to send prepared crates of medical supplies to those less seriously wounded. Some are quite small and carry single crates, while others move carriers containing many crates for subsequent distribution. In preparation, a set of ground robots can move crates out of warehouses and (if required) onto carriers.

This somewhat dramatic scenario involves a wide variety of agents, such as UAVs and ground robots, that need to collaborate to achieve common goals. For several reasons, the actions of these agents often need to be known to some degree before the start of a mission. For example, authorities may require pre-approval of unmanned autonomous missions occurring in specific areas. Even lacking such legal requirements, ground operators responsible for missions or parts of missions often prefer to know what will happen in advance. At the same time, pre-planning a mission in every detail may lead to brittle plans, and it is generally preferable to leave a certain degree of freedom to each agent in the execution of its assigned tasks.

We are therefore interested in solutions where high-level mission plans are generated at a centralized level, after which agent-specific subplans are extracted and delegated to individual agents. Each agent can then view its own subset of the original high-level plan as a set of goals and constraints, after which it can generate a more detailed plan with full knowledge of the fine-grained platform-specific actions at its disposal. This can be said to be a hierarchical hybrid between centralized and decentralized planning.

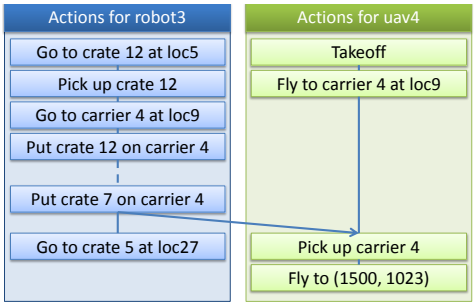

Fig. 1. Example plan structure

A suitable plan structure for this type of collaborative multi-agent mission should be sufficiently flexible to reflect the most essential aspects of the true execution capabilities of the agent or agents involved. In particular, the plan structure should allow us to make full use of the potential for concurrent execution and to provide a high degree of execution flexibility. Plans should therefore be minimally constrained in terms of precedence between actions performed by different agents, and should not force an agent to wait for other agents unless this is required due to causal dependencies, resource limitations, or similar constraints arising from the domain itself. This requires a plan structure capable of expressing both precedence between actions and the *lack* of precedence between actions.

Though partially ordered plans satisfy this requirement, the full expressivity afforded by such plans may be excessive for our motivating scenario: While partial ordering is required between actions executed by *different* agents, the actions for each *individual* agent could be restricted to occur sequentially[1]. Figure 1 shows a small example for two agents: A UAV flies to a carrier, but is only allowed to pick it up after a ground robot has loaded a number of crates. The actions of each agent are all performed in a predetermined sequence. Due to partial ordering between different agents, the ground robot can immediately continue to load another carrier without waiting for the UAV.

[1]Note that this can easily be extended to allow multiple sequential threads of execution within each agent.

Since planners generating partial-order plans already exist, applying additional restrictions to the standard partially ordered plan structure is only reasonable if there is an accompanying gain in some other respect. The potential for such gains follows directly from the fact that stronger ordering requirements yield stronger information about the state of the world at any given point in the plan. Many planners exploit such information to great success in state-based heuristics [1], [2] or in the evaluation of domain-specific control formulas [3], [4]. Such control formulas have been shown to improve planning performance by orders of magnitude in many domains. They can also be used to efficiently forbid a variety of suboptimal action choices that can be made by a forward-chaining planner, often leading to higher-quality plans. It would therefore be interesting to investigate to what extent this potential can be realized in practice.

In this paper, we begin these investigations by introducing some aspects of forward-chaining into partial-order planning, leading to a new hybrid *partial order forward-chaining* (POFC) framework (Section II) and a prototype planner operating within this framework (Section III). This planner generates stronger state information than is usually available in partial-order planning, allowing the use of certain forms of domain-specific control formulas for pruning the search space. We then show how we apply this planner to collaborative UAV systems in the UASTech group (Section IV). Finally, we discuss related work (Section V) and present our conclusions (Section VI).

II. Partial Order Forward-Chaining

Partial order forward-chaining (POFC [5]) is a new framework intended for use in partly or fully centralized multi-agent planning, where each agent can be assigned a sequence of actions but where there should be a partial ordering between actions belonging to different agents.

A variety of planners could be implemented in the POFC framework. Common to these planners would be the assumption that a problem instance explicitly specifies a set of agents $\mathcal{A}$ and that every action (operator instance) specifies the agent to which it belongs. Plan structures will differ depending on the expressivity of a planner operating within this framework, but would typically include a set of actions A and a partial ordering relation $\preceq$ on actions. To reflect the constraint that all actions belonging to any specific agent should be sequentially ordered, POFC planning requires any pair of actions $a_1, a_2 \in A$ belonging to the same agent to satisfy $a_1 \preceq a_2$ or $a_2 \preceq a_1$. This is satisfied in the plan in Figure 1, for example. A POFC plan may also contain additional components, such as a temporal constraint network.

An identifying characteristic of partial order forward-chaining is that the subset of actions belonging to a specific agent are not only executed in sequential order but also *added* in sequential order during plan generation. Suppose a planner is considering possible extensions to the plan shown in Figure 1. Any potential new action for robot3 must then be added strictly after the action of going to loc27. On the other hand, the new action could remain unordered relative to the actions of uav4, unless an ordering constraint is required due to preconditions or other executability conditions. Actions belonging to distinct agents can therefore be independent of each other to the same extent as in a standard partial order plan.

In many domains, state variables such as the location or fuel level of an agent are only affected by actions performed by the agent itself (unless, for example, some agents actively move others). As a direct consequence of the fact that actions for each agent are added in sequential order, complete information about such "agent-specific" state variables can easily be generated at any point along an agent's action sequence in essentially the same way as in standard forward-chaining.

Furthermore, agents are in many cases comparatively loosely coupled [6]: Direct interactions with other agents are relatively few and occur comparatively rarely. For example, a ground robot would require a long sequence of actions to load a set of crates onto a carrier. Only after this sequence is completed will there be an interaction with the UAV that picks up the carrier. This means that for extended periods of time, agents will mostly act upon and depend upon state variables that are not *currently* affected or required by other agents. Again, POFC planning allows the values of such state variables to be propagated within the subplan associated with a specific agent in a way similar to forward-chaining.

Thus, the use of agent-specific action sequences is key to allowing POFC planners to generate agent-specific states that are partial but considerably richer than the information available to a standard partial-order planner. Such states can then be used in heuristics or control formulas, as well as in the evaluation of preconditions. This information is particularly useful for the agent itself, since its own actions are likely to depend to a large extent on its own agent-specific variables.

In general, though, state information cannot be complete. In Figure 1, we cannot know exactly where uav4 will be immediately after robot3 moves to loc27, since the precedence constraints do not tell us whether this occurs before or after uav4 flies to a new location. However, state information can be regained when interactions occur: If the next action for robot3 is constrained to occur *after* uav4 flies to a new location, complete information about the location of uav4 can once again be inferred. In this sense, state information can "flow" between agent-specific partial states along precedence constraints. See Section III-D for further details and examples.

These general ideas will now be exemplified and clarified through the definition of a prototype planner operating within the POFC framework. This planner uses agent-specific action sequences to generate state information and effectively exploits such states to enable efficient evaluation of preconditions and control formulas.

III. A Prototype POFC Planner

We now present an initial prototype planner operating within the framework of partial order forward-chaining. In this planner, goal-directedness is achieved through domain-specific precondition control formulas [3], [4], [7] as explained below. This can be very effective due to the comparatively rich state

information afforded by the POFC plan structure. Means-ends analysis as in standard partial order causal link (POCL [8]) planning, or state-based heuristics as in many forward-chaining planners, could also be explored in the future.

A. Domains and Problem Instances

We assume a typed finite-domain state variable representation. State variables will also be called *fluents*. For example, $\mathsf{loc}(\mathit{crate})$ might be a location-valued fluent taking a crate as its only parameter. For any problem instance, the *initial state* must provide a complete definition of the values of all fluents. The *goal* is typically conjunctive, but may be disjunctive.

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 as the operator $\mathsf{fly}(\mathit{uav}, \mathit{from}, \mathit{to})$. An *action* is a fully instantiated (grounded) operator. Given finite domains, any operator corresponds to a finite set of actions.

Each operator is associated with a *precondition* formula and a set of *precondition control* formulas, all of which may be disjunctive and quantified and must be satisfied at the time when an instance of the operator is invoked. Precondition control represents conditions that are not "physically" required for execution but should be satisfied for an action to be meaningful given the current state and the goal [3], [7]. The construct $\mathsf{goal}(\phi)$ tests whether ϕ is entailed by the goal. For example, flying a loaded carrier to a location far from where its crates should be delivered according to the goal can be prevented using a control formula. We often use "conditions" to refer to both preconditions and control formulas.

Note that given the search method used in this planner, precondition control will not introduce new subgoals that the planner will attempt to satisfy. Instead, control formulas will be used effectively to prune the search space.

An operator has a *duration* specified by a possibly state-dependent temporal expression. We currently assume that the true duration of any action is strictly positive and cannot be controlled directly by the executing agent. We assume no knowledge of upper or lower bounds, though support for such information will be added in the future.

A set of *mutexes* can be associated with every operator. Mutexes are acquired throughout the duration of an action to prevent concurrent use of resources. For example, an action loading a crate onto a carrier may acquire a mutex associated with that crate to ensure that no other agent is allowed to use the crate simultaneously. Mutexes must also be used to prevent actions having mutually inconsistent effects from being executed in parallel. Thus, mutual exclusion between actions is not modeled by deliberately introducing inconsistent effects.

For simplicity, we initially assume single-step operators, where all *effects* take place in a single effect state. Effects are currently conjunctive and unconditional, with the expression $\mathsf{f}(\overline{v}) := v$ stating that the fluent $\mathsf{f}(\overline{v})$ is assigned the value v. Both v and all terms in $\overline{v}$ must be either value constants or variables from the formal parameters of the operator. For example, $\mathsf{fly}(\mathit{uav}, \mathit{from}, \mathit{to})$ may have the effect $\mathsf{loc}(\mathit{uav}) := \mathit{to}$.

B. Plan Structures, Executable Plans and Solutions

For our initial POFC planner, a *plan* is a tuple $\langle A, L, O\rangle$ whose components are defined as follows.

- A is the set of actions occurring in the plan.
- L is a set of ground causal links $a_i \xrightarrow{\mathsf{f}=v} a_j$ representing the commitment that the a_i will achieve the condition that f takes on the value v for a_j. This is similar to the use of causal links in partial order causal link (POCL) planning.
- O is a set of ordering constraints on A whose transitive closure is a partial order denoted by $\preceq$. We define $a_i \prec a_j$ iff $a_i \preceq a_j$ and $a_i \neq a_j$. We interpret $a_i \prec a_j$ as meaning that a_i ends before a_j begins. The expression $a_i \prec_{\text{imm}} a_j$ is a shorthand for $a_i \prec a_j \wedge \nexists a.a_i \prec a \prec a_j$, indicating that a_i is an immediate predecessor of a_j.

By the definition of partial order forward-chaining, the actions associated with any given agent must be totally ordered by O.

As in POCL planning, we assume a special initial action $a_0 \in A$ without conditions or mutexes, whose effects completely define the initial state. Any other action $a_i \neq a_0 \in A$ must satisfy $a_0 \prec a_i$. Due to the use of forward-chaining techniques instead of means-ends analysis, there is no need for an action whose preconditions represent the goal.

A POFC plan such as the one in Figure 1 places certain constraints on when actions may be invoked. For example, uav4 must finish taking off before it begins flying to carrier 4. At the same time, the execution mechanism is free to make choices such as whether robot3 begins going to crate 12 before uav4 begins taking off or vice versa. Additionally, the order in which actions end is generally unpredictable: uav4 may finish taking off before or after robot3 finishes going to crate 12.

An *executable* plan satisfies all executability conditions (preconditions, control formulas and mutexes) regardless of the choices made by the execution mechanism and regardless of the outcomes of unpredictable action durations.

To define executability more formally, we associate each action $a \in A$ with an *invocation node* $\mathit{inv}(a)$ where conditions must hold and mutexes are acquired, and an *effect node* $\mathit{eff}(a)$ where effects take place and mutexes are released. For all actions $a, a' \in A$, we let $\mathit{eff}(a) \prec \mathit{inv}(a')$ iff $a \prec a'$, meaning that a must end before a' is invoked. For all actions $a \in A$, we let $\mathit{inv}(a) \prec \mathit{eff}(a)$: An action must begin before it ends. Then every total ordering of nodes compatible with $\prec$ corresponds directly to one combination of choices and outcomes. For example, Figure 2 shows three partial node sequences compatible with the plan defined in Figure 1, including the special initial action a_0 used in this particular POFC planner.

A plan is executable iff every node sequence compatible with the plan is executable. The executability of a single node sequence is defined as follows. Observe that the first node must be the invocation node of the initial action a_0, which has no preconditions. The second node is the effect node of a_0, which completely defines the initial state. After this prefix, invocation nodes contain preconditions and precondition control formulas that must be satisfied in the "current" state. Effect nodes update the current state and must have internally consistent effects

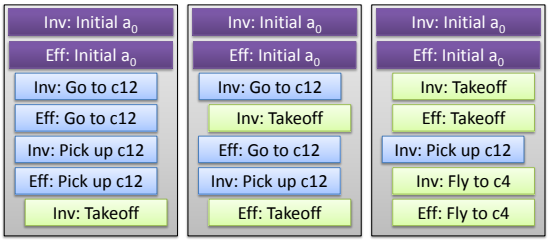

Fig. 2. Three node sequences compatible with the plan in Figure 1

(must not assign two different values to the same fluent). Finally, executability also requires that no mutex is held by more than one agent in the same interval of time.

An executable plan is a *solution* iff every compatible node sequence results in a final state satisfying the goal.

C. Search Space and Main Planning Algorithm

We are currently exploring a search space where adding a new action to an executable plan, together with a new set of causal links and precedence constraints, is only permitted if this results in a new executable plan: The conditions of the new action must be satisfied at the point where it is inserted in the current plan structure, its effects must be internally consistent and must not interfere with existing actions in the plan, and mutexes must be satisfied.

The following is a high-level description of the prototype planner. For simplicity and clarity, we present it as a non-deterministic algorithm. In reality, standard complete search methods such as depth first search with cycle detection can be used. Furthermore, each iteration in the algorithm below generates *all* executable actions for a given agent before selecting one of these actions. This is also a simplification for clarity, where the real planner does not have to generate all executable actions before making a choice.

Algorithm POFC-prototype-planner(a_0, g):
  $A \leftarrow \{a_0\}$; $L \leftarrow \varnothing$; $O \leftarrow \varnothing$
  $\pi \leftarrow \langle A, L, O \rangle$
  Generate agent-local initial states as shown in Section III-D
  // If we can ensure the goal g holds after execution
  // by introducing new precedence constraints, we are done
  while adding precedence constraints is insufficient to satisfy g **do**
    // Choose an agent that will be assigned a new action
    agent $\leftarrow$ nondet-choice$(\mathcal{A})$
    // Choose an action a that can be made executable
    // given that the precedence constraints P and causal links C
    // are added to the current plan
    $\langle a, P, C \rangle \leftarrow$ nondet-choice(find-executable$(\pi$, agent, $g))$
    // Apply the action and iterate
    $A \leftarrow A \cup \{a\}$ *// New action*
    $L \leftarrow L \cup C$ *// New causal links*
    $O \leftarrow O \cup P$ *// New precedence constraints*
    // Update and generate states according to Section III-F
    Update existing states: state-update$(\langle A, L, O \rangle, a)$
    Generate new state: generate-state-after$(\langle A, L, O \rangle, a)$

At the highest level, the prototype planner appears quite similar to any other planner based on search. It begins with an initial search node corresponding to the initial executable plan, containing only the special initial action a_0. Given a node

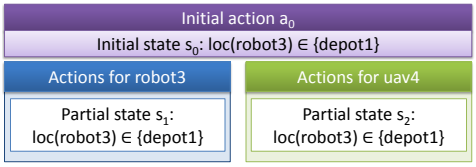

Fig. 3. Partial states for an initial (empty) plan.

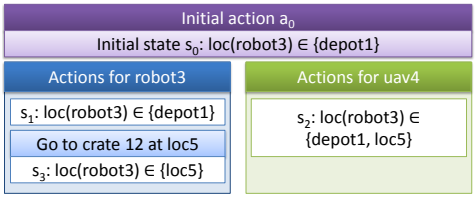

Fig. 4. Partial states after one action has been added.

corresponding to a plan π, it tests whether the goal is satisfied after execution – or rather, whether the goal can be *made* satisfied through the introduction of additional precedence constraints. This is tested using the *make-true* procedure presented in Section III-E. If not, a child node can be generated by deciding which agent should be given a new action in π, and then choosing a new action to be added at the end of that agent's action sequence, together with precedence constraints and causal links ensuring that the new plan is executable.

Many heuristics can be used for choosing an agent, such as using as few agents as possible or distributing actions evenly across all available agents. In the latter case, we can calculate the time at which each agent is expected to finish executing its currently assigned actions and test agents in this order.

D. Partial States and Initial States

A variety of state structures can be used to store partial information about the state at distinct points in a plan. We initially use a simple structure where a finite set of possible values is stored for each fluent: $\mathsf{f} \in \{v_1, \ldots, v_n\}$. The evaluation procedure defined in the next section resolves as much of a formula as possible in such partial states. Should this not be sufficient to completely determine whether the formula is true, the procedure falls back on an explicit traversal of the plan structure for those parts of the formula that remain unknown. This grounds evaluation in the initial state and the explicit effects in the plan for completeness.

The initial plan consists of a single action a_0, which completely defines the initial state s_0 of the planning problem at hand. In preparation for future agent-specific state updates, this state is copied into a partial state structure for each agent as indicated in Figure 3, where s_1 and s_2 initially contain all facts specified in s_0. For example, robot3 is initially at depot1.

A partial state represents facts known to hold over an interval of time. This interval starts at the end of a specific action for a given agent, or at the beginning of the plan if there is no preceding action. It ends at the beginning of the next action for the same agent, or at the end of the plan if no such action exists. For example, state s_1 in Figure 4 represents

what must hold from the beginning of the plan until robot3 begins moving to loc5.

Whenever a new action has been added, existing states must be updated and a new state must be created. This will be discussed in Section III-F.

E. Searching for Applicable Actions

When the planner searches for a new applicable action for a particular agent (such as uav4 in Figure 4), there already exists a partial state describing facts that must hold up to the point where the new action will be invoked (in this case, state s_2). This state can be used for the efficient evaluation of preconditions and control formulas.

Given sufficiently loose coupling, together with the existence of agent-local and static facts, this state will be sufficient to immediately determine the executability of most actions related to the chosen agent. In some cases, though, the partial information in the state will be insufficient. This can be due to our use of simple partial state structures that cannot represent arbitrary sets of possible states, or due to incomplete state update procedures. Another reason is that given partially ordered plans, we cannot know in which order actions will start and end. In Figure 4, for example, we cannot know whether the next action to be added for uav4 will start before or after robot3 moves to loc5. Therefore we cannot predict exactly which value loc(robot3) will have when this new action starts – only that it will be either depot1 or loc5.

Nevertheless, completeness requires a complete evaluation procedure. This procedure also needs the ability to introduce new precedence constraints to ensure that a precondition holds. For example, suppose that the precondition of a potential new action for uav4 requires robot3 to be at loc5. This can be satisfied by ensuring that the new action occurs not only after all existing actions for uav4, but also after robot3 goes to loc5. Such alternatives must also be considered by the planner.

For this purpose we define replace plain formula evaluation in a partial state with the procedure *make-true*(α, a, s, g, π). This procedure assumes that the action a whose conditions α should be tested has temporarily been added to the plan π, and that the last known partial state before a is s. It recursively determines whether α can be made to hold when a is invoked, possibly through the addition of new precedence constraints. When necessary due to incomplete state information, it explicitly searches π for actions having suitable effects. The goal g is provided to enable evaluation of goal() constructs.

The procedure returns a set of *extensions* corresponding to the minimally constraining ways in which the precedence order can be constrained to ensure that α holds when a is invoked. Each extension is a tuple $\langle P, C\rangle$ where P is a set of precedence constraints to be added to O and C is a set of causal links to be added to L. Thus, if α is proven false regardless of which precedence constraints are added, $\varnothing$ is returned: There exists no valid extension. If α is proven true without the addition of new constraints, $\{\langle \varnothing, C\rangle\}$ is returned for some suitable set of causal links C. In this case, s can be updated accordingly, providing better information for future formula evaluation.

Below, certain aspects of the *make-true* procedure have been simplified to improve readability while retaining correctness. A number of optimizations to this basic procedure can and have been applied. For example, suppose one is evaluating a formula such as $\alpha \wedge \beta$ and it cannot be determined using the current partial state alone whether the first subformula α holds. Then the attempt to "make" it true by the introduction of new precedence constraints can be deferred while β is evaluated. If β turns out to be false, the entire formula must be false and there is no need to return to the deferred subformula.

Algorithm *make-true*$(\alpha, a, s, g, \pi = \langle A, L, O\rangle)$
  // Returns a set of $\langle$precedence, causallink$\rangle$ tuples
  if α is $\mathsf{f} = v$ **then**
    if $s \models \alpha$ **then**
      // Formula known to be true. Need to find an action that can
      // support a causal link: Must assign the right value, occur
      // before a, and no action must be able to interfere. At least
      // one possibility will exist without the need for additional
      // precedence constraints!
      $S \leftarrow \{a_i \in A \mid a_i \prec a \ \wedge \ a_i$ assigns $f := v$
          $\wedge$ no action a_j can interfere by assigning another value between a_i and $a\}$
      // All the tuples below are minimally constraining extensions
      return $\{\ \langle \varnothing, \{a_i \xrightarrow{\mathsf{f}=v} a\}\rangle \mid a_i \in S\ \}$
    else if $s \models \neg\alpha$ **then**
      // Formula known to be false. No support possible.
      return $\varnothing$
    else
      // Insufficient information in s. Formula could already be true,
      // in which case $P = \varnothing$ will be found below. Or we may be
      // able to "make" it true given new precedence constraints.
      $S \leftarrow \{a_i \in A \mid a_i \prec a \ \wedge \ a_i$ assigns $f := v\}$
      $E \leftarrow \varnothing$
      for all $a_i \in S$ **do**
        for all minimally constraining new precedence constraints P that would ensure that the relevant effect of a_i cannot be interfered with between a_i and a **do**
          $E \leftarrow E \cup \{\langle P, \{a_i \xrightarrow{\mathsf{f}=v} a\}\rangle\}$
      return E
  else if α is $\neg(\mathsf{f} = v)$ **then**
    // Handled similarly
  else if α is goal(ϕ) **then**
    if $g \models \alpha$ **then return** $\{\langle \varnothing, \varnothing\rangle\}$ **else return** $\varnothing$
  else if α is $\neg$goal(ϕ) **then**
    if $g \models \alpha$ **then return** $\varnothing$ **else return** $\{\langle \varnothing, \varnothing\rangle\}$
  else if α is $\neg\beta$ **then**
    // Push negations in, until they appear before an atomic formula.
    // For example, $\neg(\beta \wedge \gamma) = (\neg\beta) \vee (\neg\gamma)$.
    $\gamma \leftarrow$ push negations in using standard equivalences
    return *make-true*(γ, a, s, g, π)
  else if α is $\beta \wedge \gamma$ **then**
    // Both conjuncts must be satisfied. For every way we can satisfy
    // the first, find every way in which the second can be satisfied.
    // May result in non-minimal extensions that can be filtered out.
    $E \leftarrow \varnothing$
    for all $\langle P, C\rangle \in$ *make-true*(β, a, s, g, π) **do**
      $E \leftarrow E \cup$ *make-true*$(\gamma, a, s, g, \langle A, L \cup C, O \cup P\rangle)$
    Remove extensions not minimal in terms of precedence
    return E
  else if α is $\beta \vee \gamma$ **then**
    // It is sufficient that one of the disjuncts is satisfied. Calculate
    // all ways of satisfying either disjunct, and retain the minimal
    // extensions.
    $E \leftarrow$ *make-true*$(\beta, a, s, g, \pi) \cup$ *make-true*(γ, a, s, g, π)

Remove extensions not minimal in terms of precedence
return E
else if α is $\forall v.\beta(v)$ **then**
Treat as conjunction over all values in the finite domain of v
else if α is $\exists v.\beta(v)$ **then**
Treat as disjunction over all values in the finite domain of v

Though *make-true* is important for the procedure of finding new executable actions, it only considers preconditions and control formulas. For each extension returned by *make-true*, we may have to add further precedence constraints to ensure that no mutex is used twice concurrently. Additional constraints may be required to ensure that the new potential action does not interfere with existing causal links in the plan. This results in the following procedure.

Algorithm find-executable($\pi = \langle A, L, O\rangle$, agent, g)
executable $\leftarrow \varnothing$
lastact $\leftarrow$ the last action associated with the given agent in π, or a_0 if no such action exists
laststate $\leftarrow$ the last partial state for the given agent in π
for all potential actions a associated with the given agent **do**
// Temporarily add the potential action to the plan
$\pi' \leftarrow \langle A \cup \{a\}, L, O \cup \{\text{lastact} \prec a\}\rangle$
for all $\langle P, C\rangle \in$ *make-true*(conditions(a), a, laststate, g, π') **do**
for all $\langle P', C\rangle$ minimally extending $\langle P, C\rangle$ so that no mutex is used twice concurrently **do**
for all $\langle P'', C\rangle$ minimally extending $\langle P', C\rangle$ so that a cannot interfere with existing causal links in L **do**
executable $\leftarrow$ executable $\cup \{\langle a, P'', C\rangle\}$

It may seem like this procedure results in a combinatorial explosion of alternative extensions. However, standard POCL planners have essentially the same alternatives to choose from, the main difference being that alternatives not generated in a single procedure but selected through a long sequence of plan refinement steps. POFC implementations can easily generate extensions incrementally as needed.

Note also that every new precedence constraint generated leads to fewer options available in the next step, which provides a natural limit to the potential complexity. Searching for existing support for all conditions *in the current plan*, as opposed to leaving open "flaws" to be treated later, also tends to reduce the number of consistent extensions. Additionally, the initial filtering based on the partial state quickly filters out most candidate actions.

Finally, we would like to note that evaluation performance can be improved by analyzing preconditions and control formulas in order to extract common parts that only depend on some or none of the parameters of a particular operator.

F. Generating and Updating States

Updating Existing States. When a new action is added to a plan, some of the existing partial states must be updated. As an example, let us expand the plan in Figure 3 with the action of robot3 moving to loc5, resulting in the plan in Figure 4. Recall that state s_2 in this figure should represent what we know about the state of the world from the beginning of the plan up to the first action that will eventually be performed by uav4. Given the action that was just added, we no longer know whether robot3 will remain at depot1 throughout this interval of time. It might, but it may also finish going to loc5 before uav4 begins executing its first action. What we can say with certainty is that at any point of time in the relevant interval, robot3 will be either at depot1 or at loc5.

State information must always be sound, but since formula evaluation will be able to fall back on explicit plan traversal, it does not have to be complete. Therefore, updates do not have to yield the *strongest* information that can be represented in the state structure, and a tradeoff can be made between the strength and the efficiency of the update procedure.

A simple but sound state update procedure could weaken the states of *all* existing nodes in the plan: 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 $\mathsf{f} \in V \cup \{v\}$.

However, suppose that a state s is followed by an action a that in turn precedes the newly added action. Clearly, the new action cannot interfere with s, as this would require interference backwards in time. In Figure 1, for example, the action of picking up carrier 4 has ancestors belonging to robot3 as well as uav4 and cannot interfere with the states of these nodes. Therefore, the following procedure is sufficient.

Algorithm state-update($\pi = \langle A, L, O\rangle$, newact)
for all partial states s stored in π **do**
if s is followed by an action a such that $a \preceq$ newact **then**
// No update needed: New effects cannot interfere with s
else
for all $\mathsf{f}(\overline{v}) := v \in$ effects(newact) **do**
add v to the set of values for $\mathsf{f}(\overline{v})$ in s

Generating New States. In addition to updating new states, a new partial state must be created representing facts known to hold from the end of the newly added action. For example, when the action of going to crate12 was added to Figure 3, a new state s_3 had to be generated.

Any new action a always has at least one immediate predecessor p such that $p \prec_{\text{imm}} a$. For example, the action of going to crate12 has a single immediate predecessor: a_0.

Let a be a new action and p one of its immediate predecessors. Clearly, the facts that hold after p will still hold when a is invoked except when there is interference from intervening effects. Therefore, taking the state associated with p and "weakening" it with all effects that *may* occur between p and a, in the same manner as in the state update procedure, will result in a new partial state that is valid when a is invoked. For example, let a be the action of robot3 going to crate12. We can then generate a state that is valid when a is invoked by taking the initial state and weakening it with the effects associated with uav4 taking off and flying to carrier 4, since this is the only action that might intervene between a_0 and a.

Now suppose that we apply this procedure to two immediate predecessors p_1 and p_2. This would result in two states s_1 and s_2, *both* describing facts that must hold when the new action a is invoked. If s_1 claims that $\mathsf{f} \in V_1$ and s_2 claims that $\mathsf{f} \in V_2$ for some fluent f, then both of these claims must be true. We therefore know that $\mathsf{f} \in V_1 \cap V_2$. This can be extended to an arbitrary number of immediate predecessors.

Conjoining information from multiple predecessors often

results in gaining "new" information that was not previously available for the current agent. For example, if robot3 loads crates onto a carrier, incrementally updating a total-weight fluent, other agents will only have partial information about this fluent. When uav4 picks up the carrier, this action must have the last load action of robot3 as an immediate predecessor. The UAV thereby gains complete information about weight and can use this efficiently in future actions.

This results in a state that is valid when the new action a is invoked. To generate a state valid when a ends, the effects of a must also be applied to the new state.

Algorithm generate-state-after($\langle A, L, O\rangle$, newact)
  newstate $\leftarrow$ a completely undefined state
  for all $p \in A$: $p \prec_{\text{imm}}$ newact **do**
    for all fluents f **do**
      values $\leftarrow$ the values for f in the state immediately after p
      for all $a \in A$ that can interfere between p and newact **do**
        $v \leftarrow$ the value assigned to f by a
        values $\leftarrow$ values $\cup \{v\}$
      newstate$[f] \leftarrow$ newstate$[f] \cap$ values
  apply the effects of newact to newstate
  return newstate

G. Completeness

Requiring every intermediate search node to correspond to an executable plan does not result in a loss of completeness. Intuitively, the reason is that there can be no circular dependencies between actions, where adding several actions at the same time could lead to a new executable plan but adding any single action is insufficient.

More formally, let $\pi = \langle A, L, O\rangle$ be a non-empty executable plan. Let $a \in A$ be an action such that there exists no other action $b \in A$ where $a \prec b$ (for example, the action of going to crate 5 in Figure 1). Such an action must exist, or the precedence relation would be circular and consequently not a partial order, and π would not have been executable.

Since a does not precede any other action, it cannot be required in order to support the preconditions or control formulas of other actions in π. Similarly, a clearly cannot be required for mutual exclusion to be satisfied. Consequently, removing a from π must lead to an executable plan π'. By induction, any finite executable plan can be reduced to the initial plan through a sequence of reduction steps, where each step results in an executable plan. Conversely, any executable plan can be constructed from the initial plan through a sequence of action additions, each step resulting in an executable plan.

Given finite domains, solution plans must be of finite size and can be constructed from the initial plan through a finite number of action additions. The action set must also be finite. Furthermore, when any particular action is added to a plan, there must be a finite number of ways to introduce new precedence constraints and causal links ensuring that the plan remains executable. Any search node must therefore have a finite number of children, and the search space can be searched to any given depth in finite time.

Thus, given a method for finding all applicable actions for a given agent, we can incrementally construct and traverse a search space. Given a complete search method such as iterative deepening or depth first search with cycle detection, we have a complete planner.

Fig. 5. The UASTech Yamaha RMAX helicopter system

IV. Planning for Collaborative UAV Systems

The research context in which this planning framework is being developed focusses on the topic of mixed-initiative decision support for collaborative Unmanned Aircraft Systems [9], [10]. In the broader context, we are developing a delegation framework [11] which formally characterizes the predicate *Delegate*$(A, B, \textit{Task}, \textit{Constraints})$, where an agent A delegates a *Task* to agent B in a context characterized as a set of *Constraints*. It is often the case that a *Task* is in fact a goal statement. In this case, in order for agent B to accept the task and for delegation to take place, it must find a plan which satisfies the goal statement. A recursive process may then ensue where agent B makes additional calls for delegation of additional tasks to other agents. The character of these tasks might be new goal statements or sequences of abstract actions in a loosely coupled plan already generated by agent B.

In the latter case, agent B would broadcast for agents with specific capabilities and roles associated with the goal statement. For example, in the logistics example described in the introduction, agent B would look for a number of agents capable of lifting food and medical supplies onto carriers. It would also look for agents such as our modified Yamaha RMAX helicopters (Figure 5) capable of lifting carriers with supplies already loaded and taking them to locations where injured inhabitants have been geo-located. Given this set of agents as input, agent B would then use the POFC planner described here to generate a loosely coupled distributed plan for the given agents. The output would be sequences of abstract actions which would then be delegated to these agents by agent B. The agents would then have to check whether they could instantiate these abstract actions in their specific action repertoires while taking into account the dependency constraints associated with the larger loosely coupled plan. If these recursive calls are successful, then the original delegation from agent A to B will also be successful. This integration of the POFC planner with the delegation framework has in fact been done and a prototype system is now being integrated with our UAV systems.

V. Related work

A variety of planners creating temporal partially ordered plans exist in the literature and could potentially be applied in multi-agent settings. Some of these planners also explicitly focus on multi-agent planning. For example, Boutilier and Brafman [12] focus on modeling concurrent interacting actions, in a sense the opposite of the loosely coupled agents we aim at.

However, very little appears to have been done in terms of taking advantage of *forward-chaining* when generating partially ordered plans for multiple agents. An extensive search through the literature reveals two primary examples.

First, a multi-agent planner presented by Brenner [13] does combine partial order planning with forward search. However, the planner does not explicitly separate actions by agent and does not keep track of agent-specific states. Instead, it evaluates conjunctive preconditions relative to those value assignments that must hold after *all* actions in the current plan have finished. The evaluation procedure defined in this paper is significantly stronger. In fact, as Brenner's evaluation procedure cannot introduce new precedence constraints, the planner is incomplete.

Second, the FLECS planner [14] uses means-ends analysis to add relevant actions to a plan. A FLExible Commitment Strategy determines when an action should be moved to the end of a totally ordered plan prefix, allowing its effects to be determined and increasing the amount of state information available to the planner. Actions that have not yet been added to this prefix remain partially ordered. Though there is some superficial similarity in the combination of total and partial orders, FLECS uses a completely different search space and method for action selection. Also, whereas we strive to generate the weakest partial order possible between actions performed by different actions, any action that FLECS moves to the plan prefix immediately becomes totally ordered relative to all other actions. FLECS therefore does not retain a partial order between actions belonging to distinct agents.

Thus, we have found no planners taking advantage of agent-specific forward-chaining in the manner described in this paper.

VI. Conclusion

We have presented a hybrid planning framework applicable to centralized planning for collaborative multi-agent systems. We have also described one of many possible planners operating within this framework. Though this work is still in the early stages, a prototype implementation is in the process of being integrated with the UASTech UAV architecture and will be tested in flight missions in the near future. Interesting topics for future research include integration with existing techniques for execution monitoring and recovery from execution failures [15] and the extension of these techniques to handle dynamic reconfiguration after execution failures.

Acknowledgment

This work is partially supported by grants from the Swedish Research Council (2009-3857), the CENIIT Center for Industrial Information Technology (06.09), the ELLIIT network organization for Information and Communication Technology, the Swedish Foundation for Strategic Research (SSF) Strategic Research Center MOVIII, and the Linnaeus Center for Control, Autonomy, Decision-making in Complex Systems (CADICS).

References

[1] B. Bonet and H. Geffner, "HSP: Heuristic search planner," *AI Magazine*, vol. 21, no. 2, 2000.

[2] J. Hoffmann and B. Nebel, "The FF planning system: Fast plan generation through heuristic search," *Journal of Artificial Intelligence Research*, vol. 14, pp. 253–302, 2001.

[3] F. Bacchus and F. Kabanza, "Using temporal logics to express search control knowledge for planning," *Artificial Intelligence*, vol. 116, no. 1-2, pp. 123–191, 2000.

[4] J. Kvarnström and P. Doherty, "TALplanner: A temporal logic based forward chaining planner," *Annals of Mathematics and Artificial Intelligence*, vol. 30, pp. 119–169, Jun. 2000.

[5] J. Kvarnström, "Planning for loosely coupled agents using partial order forward-chaining," in *Proceedings of the 26th Annual Workshop of the Swedish Artificial Intelligence Society (SAIS)*, Uppsala, Sweden, May 2010, pp. 45–54.

[6] R. I. Brafman and C. Domshlak, "From one to many: Planning for loosely coupled multi-agent systems," in *Proceedings of the 18th International Conference on Automated Planning and Scheduling (ICAPS)*, Sydney, Australia, 2008, pp. 28–35.

[7] F. Bacchus and M. Ady, "Precondition control," 1999, available at http://www.cs.toronto.edu/∼fbacchus/Papers/BApre.pdf.

[8] D. S. Weld, "An introduction to least commitment planning," *AI magazine*, vol. 15, no. 4, p. 27, 1994.

[9] P. Doherty, P. Haslum, F. Heintz, T. Merz, P. Nyblom, T. Persson, and B. Wingman, "A distributed architecture for autonomous unmanned aerial vehicle experimentation," in *Proc. DARS*, 2004.

[10] P. Doherty, "Advanced research with autonomous unmanned aerial vehicles," in *Proc. KR*, 2004.

[11] P. Doherty and J.-J. C. Meyer, "Towards a delegation framework for aerial robotic mission scenarios," in *Proc. 11th International Workshop on Cooperative Information Agents (CIA-07)*, 2007.

[12] C. Boutilier and R. I. Brafman, "Partial-order planning with concurrent interacting actions," *Journal of Artificial Intelligence Research*, vol. 14, pp. 105–136, 2001.

[13] M. Brenner, "Multiagent planning with partially ordered temporal plans," in *Proc. IJCAI*, 2003.

[14] M. Veloso and P. Stone, "FLECS: Planning with a flexible commitment strategy," *Journal of Artificial Intelligence Research*, vol. 3, pp. 25–52, 1995.

[15] P. Doherty, J. Kvarnström, and F. Heintz, "A temporal logic-based planning and execution monitoring framework for unmanned aircraft systems," *Journal of Autonomous Agents and Multi-Agent Systems*, vol. 19, no. 3, pp. 332–337, Feb. 2009.