ICAPS05

WS4

Workshop on Multiagent Planning and Scheduling

Bradley J. Clement
Jet Propulsion Laboratory, USA

ICAPS 2005
Monterey, California, USA
June 6-10, 2005

CONFERENCE CO-CHAIRS:

Susanne Biundo
University of Ulm, GERMANY

Karen Myers
SRI International, USA

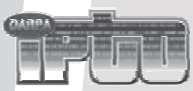
Kanna Rajan
NASA Ames Research Center, USA

Cover design: L.Castillo@decsai.ugr.es

Workshop on Multiagent Planning and Scheduling

Bradley J. Clement
Jet Propulsion Laboratory, USA







Workshop on Multiagent Planning and Scheduling

Table of contents

Workshop on Multiagent Planning and Scheduling

Preface

Multiagent planning is concerned with planning by (and for) multiple agents. It can involve agents planning for a common goal, an agent coordinating the plans (plan merging) or planning of others, or agents refining their own plans while negotiating over tasks or resources. The topic also involves how agents can do this in real time while executing plans (distributed continual planning).

More than ever industry, space, and the military are seeking systems that can solve multiagent planning (and scheduling) problems, such as those in supply chain management, coordinating space missions, and commanding mixtures of vehicles and troops. For many real-world problems, it is hard to motivate multiple agents because centralized decision-making is often most efficient. One goal of this workshop is to identify methods for discerning how and when systems should be decentralized.

Multiagent planning and scheduling seems to fall in the intersection of the fields of planning and scheduling, distributed systems, parallel computing/algorithms, and multiagent systems. However, much of the research appears to build on ideas from either planning or multiagent systems (and usually not both). From the viewpoint of planning, planning for multiple agents means supporting concurrent action, and planning by multiple agents means parallelizing a planning algorithm. One might argue that the former has been done and the latter should be solved using parallel computing techniques and is dependent on hardware. On the other hand, from a multiagent systems perspective, multiagent planning is not about just solving planning problems but also how agents should behave and interact given that they have plans or planning capabilities.

From any point of view, there are many open issues in multiagent planning. While many planners can handle some notion of concurrency, and many plan merging algorithms have been proposed, there has been little work on decentralized planning, competitive planning systems, evaluation of communication costs, and distributed continual planning. We aim for this workshop to foster ideas addressing these issues and suggest other important research questions.

Organizer

- *Bradley J. Clement, Jet Propulsion Laboratory*

Programme Committee

- *K. Suzanne Barber, University of Texas, Austin*
- *Anthony C. Barrett, Jet Propulsion Laboratory*
- *Michael Brenner, Albert-Ludwigs-Universitt Freiburg*
- *Keith S. Decker, University of Delaware*
- *Marie desJardins, University of Maryland, Baltimore County*
- *David V. Pynadath, USC Information Sciences Institute*
- *Katia Sycara, Carnegie Mellon University*
- *Tom Wagner, Defense Advanced Research Projects Agency*
- *Peter R. Wurman, North Carolina State University*

Honeywell's COORDINATORs Project Extended Abstract

David J. Musliner and John Phelps
Honeywell Laboratories
3660 Technology Drive
Minneapolis, MN 55418
{David.Musliner, John.Phelps}@honeywell.com

Introduction

For the past several years, Honeywell has been developing prototype multi-agent coordination technology to help humans coordinate their activities in complex, dynamic environments. For example, we have demonstrated a COORDINATORs concept system to assist emergency "first responders" such as firefighters and police (Wagner *et al.* 2004a; 2004b). Each first responder is paired with a COORDINATOR agent, running on a mobile computing device. COORDINATORs provide decision support to first response teams, helping them reason about who should be doing what, when, and with what resources. COORDINATORs respond to the dynamics of the environment by integrating building system reports, situation assessments, and new information from the human teams to determine the right course of action in constantly-evolving circumstances. Our COORDINATORs concept demonstrations have been implemented using commodity workstations, wireless PDAs, and proprietary first responder location tracking technologies.

Beginning in 2005, DARPA/IPTO has funded a new program to develop this type of Coordinator technology beyond the early concept stage, to provide well-founded technical approaches to the challenge of scheduling and adapting distributed activity in dynamic environments. Complete Coordinator multi-agent solutions will be developed by three independent teams led by Honeywell, ISI, and SRI. In this abstract, we provide a brief overview of the Honeywell team's project.

The Problem

COORDINATORs are intended to address the problem of effective coordination of distributed human activities. COORDINATORs help their human partners to adapt to rapidly-evolving scenarios by clearly identifying coordination alternatives and assisting in their selection, monitoring, and modification. The emphasis in these problems is not on planning from first principles (i.e., building new response plans from low-level models of environmental dynamics and primitive actions), but rather on selecting amongst the numerous possible well-understood, pre-planned alternative tasks that the human teams may perform. In this way, the problem is similar to playbook-based (Miller & Goldman 1997; Miller *et al.* 2002) concepts of team tasking: the team is expected to have trained on suitable alternative activities, and the challenge is making sure that the aggregate activities of all teams are adaptively selected and scheduled to achieve the best overall effect.

Figure 1: The prototype portable COORDINATOR display for a first responder, showing map and task information.

Demonstrating the Concept

Our early investigations into first-responder domains highlighted the multi-agent task coordination problem and its challenges. To demonstrate the concept and provide an early assessment of the potential value of a COORDINATOR system, we developed a rapid concept prototype. Using handheld computing devices and wireless networks, our demonstration system allowed a team of individuals to coordinate their activities using

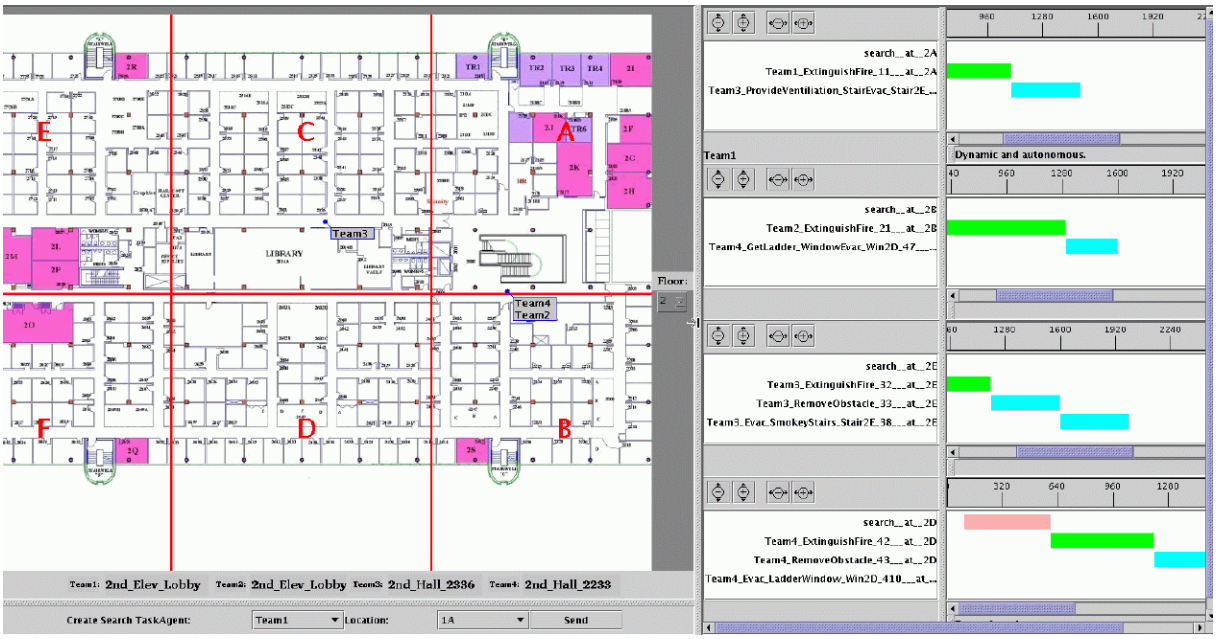


Figure 2: The prototype incident commander interface displays full map information as well as views of what tasks each team is performing.

task-oriented commands and a *centralized* scheduling system. Figure 1 illustrates our prototype handheld interface that lets each first responder see map information and tasks they are assigned to perform, as well as reporting new situation updates and creating new tasks.

Figure 2 illustrates a portion of the prototype interface used by the incident commander to monitor and control each of the distributed response teams. Mimicking the role of an on-site fire incident commander, the central scheduling system was able to properly allocate each individual's efforts in the most effective way.

To test the potential efficacy of the COORDINATOR concept, we ran volunteers through two exercises, with and without the automation support. To drive the exercises, we developed a time-pressured scenario that involves a fictitious oil refinery that has multiple, concurrent emergency situations evolving. In the first exercise, we gave our participants the state-of-the art technology widely used today: walkie-talkies. In the second exercise, we permuted the roles that the participants played in the scenario to diminish learning effects and outfitted them with a location transmitter and networked handheld computers running COORDINATORS. We also allowed them to keep the walkie-talkies. In all of our experiments, the teams performed markedly better with the COORDINATORS, despite the fact that we ran the scenario almost twice as fast as in the first exercise. More information about these results can be found in the references.

The concept demonstration helped frame the COORDINATORS problem and illustrate how a full solution might work, but it was not designed for true distribution and scaling to larger and more difficult problems.

The Honeywell Team and Approach

To address these challenges, Honeywell has teamed with the University of Massachusetts (Dr. Victor Lesser and Dr. Dan Corkill), the University of Michigan (Dr. Edmund Durfee), the University of Southern California (Dr. Milind Tambe and Dr. Sven Koenig), the University of North Carolina at Charlotte (Dr. Anita Raja), Adventium Labs (Dr. Mark Boddy) and SIFT, LLC (Dr. Robert Goldman). Together, our team combines expertise in several underlying technologies that will form the foundation of our new system:

TÆMS — The Task Analysis, Environment Modeling, and Simulation provides a representation for multi-agent hierarchical tasks with probabilistic expectations on their outcomes (characterized by quality, cost, and duration) and complex hard and soft interactions. The existing Design To Criteria (DTC) and Generalized Partial Global Planning (GPGP) software provides a heuristic approach to collaboratively negotiating over the selection and scheduling of TÆMS tasks. These components were used to build the concept demonstration described above, and will form the foundation of the new system. In fact, the TÆMS framework, including its representation and simulation capability, is being shared by all three teams on the Coordinators program.

MDPs and Constraint Optimization — While DTC and GPGP operate on native TÆMS models, the underlying semantics of TÆMS can be viewed as

a distributed MDP problem. Decisions about which agent performs which task may be addressed either as variable assignments in a constraint optimization framework, or as action choices in an MDP model, or as structural changes to local MDP problems. We plan to adapt and improve existing MDP solvers and constraint-based reasoning algorithms to address the need for incremental, time-adjustable coordination algorithms that scale to very large distributed problems.

Next Steps

The new DARPA Coordinators program efforts began in February 2005. Across the program, we are currently defining a revised version of TÆMS that supports the rigorous semantic analysis required for the various team approaches. DARPA has also funded development of a suite of test domain problems, on which each team's algorithms will be evaluated. Our challenge is to build a time-constrained distributed coordination system that produces results comparable to a centralized, non-time-limited scheduling system. We hope that the result will be revolutionary advances in multi-agent coordination technology.

References

Miller, C., and Goldman, R. P. 1997. "Tasking" interfaces; Associates that know who's the boss. In *Proc. Fourth Human Electronic Crew Conference.*

Miller, C.; Funk, H.; Whitlow, S.; and Dorneich, M. C. 2002. A playbook interface for mixed initiative control of multiple unmanned vehicle teams. In *21st Digital Avionics Systems Conf.*, 223–235.

Wagner, T.; Phelps, J.; Guralnik, V.; and VanRiper, R. 2004a. COORDINATORS - Coordination managers for first responders. In *Proceedings of the 3rd International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS04).*

Wagner, T.; Phelps, J.; Guralnik, V.; and VanRiper, R. 2004b. An application view of COORDINATORS: Coordination managers for first responders. In *Proceedings of the Sixteenth Innovative Applications of Artificial Intelligence Conference (IAAI04).*

Managing Communication Limitations in Partially Controllable Multi-Agent Plans

John Stedl and **Brian Williams**
Massachusetts Institute of Technology
Computer Science and Artificial Intelligence Laboratory
32 Vassar St. Room 32-G275, Cambridge, MA 02139
stedl@mit.edu, williams@mit.edu

Abstract

In most real world situations, cooperative multi-agent plans will contain activities that are not under direct control by the agents. In order to enable the agents to robustly adapt to this temporal uncertainty, the agents must be able to communicate at execution time in order to dynamically schedule their plans. However, it is often undesirable or impossible to maintain communication between all the agents, throughout the entire duration of the plan.

This paper introduces a two-layer approach that clusters the tightly coordinated portions of a multi-agent plan into a set of group plans, such that each group plan only requires loose coordination with one another. The key contribution of this paper is a polynomial time, Hierarchical Reformulation (HR) algorithm that combines the properties of strong and dynamic controllability, in order to decouple the partially controllable group plans from one another, while enabling the tightly coordinated activities within each group plan to be scheduled dynamically.

Introduction

The domain of plan scheduling and execution for multiple robots cooperating to achieve a common goal has applications in a wide variety of fields such as cooperative observations of Earth orbiting satellites. These applications often require tight temporal coordination between the agents, which must also be able to robustly adapt to uncontrollable events.

Previous work on dispatching of temporally flexible plans (Muscettola, Morris, & Tsamardinos 1998) (Morris, Muscettola, & Vidal 2001) provided a framework for robust execution of temporal plans. The *executive* consists of a *reformulator* and a *dispatcher*. The reformulator is an off-line compilation algorithm that prepares the plan for efficient execution. The dispatcher is an online dynamic scheduling algorithm that exploits the temporal flexibility of the plan, by waiting to schedule events until the last possible moment. In this least commitment execution strategy, the dispatcher schedules and dispatches the tasks simultaneously, rather than scheduling the tasks prior to execution. This dynamic execution strategy enables the agent to adapt to runtime uncertainty, at the cost of some online constraint propagation. Specifically, the dispatcher must propagate the execution times of each event, through local temporal constraints, towards future events, every time an event is executed. This propagation enables the dispatcher to select consistent execution times for these future events. This work, however, did not tackle the case of a distributed dispatcher for multi-robot plans, for which the challenge resides in the fact that the agents must be cope with communication limitations at execution time.

Previous work on execution of *Simple Temporal Networks with Uncertainty* (*STNUs*) (Vidal & Fargier 1999) (Morris, Muscettola, & Vidal 2001) provided methods to achieve robust execution of plans that contain uncontrollable events. STNUs are an extension of Simple Temporal Networks (STNs) (Dechter, Meiri, & Pearl 1995), in which only some of the events (or *timepoints*) in the plan are fully controllable (or *executable*), while other *contingent* timepoints cannot be scheduled directly, but rather are observed during plan execution. *Links* between pairs of timepoints impose flexible temporal constraints, which express temporal coordination in the plan (in the form of *requirement* links), and model the duration of uncontrollable activities (in the case of *contingent* links).

(Vidal & Fargier) defined a set of controllability properties for STNUs that determines under what conditions an agent can guarantee it successful execution of the plan. Informally, an STNU is controllable if there exists a consistent strategy for scheduling the executable timepoints of the plan, for all possible durations of the uncontrollable activities (subject to the constraints on the contingent and requirement links). There are three primary levels of controllability; a network is *strongly controllable* if there exists a viable execution strategy that does not depend on the outcome of the uncontrollable durations. In this case, it is possible to statically schedule all executable timepoints beforehand. A network is *dynamically controllable* if there exists a viable execution strategy that only depends on the knowledge of outcomes of past uncontrollable events. Finally, a network is *weakly controllable* if there is a viable execution strategy, given that we know the outcomes for all the uncontrollable events beforehand. Furthermore, strong controllability implies dynamic controllability which in turn implies weak controllability (Vidal & Fargier).

(Morris, Muscettola, & Vidal 2001) presented a polynomial time dynamic controllability algorithm that both checks if a plan is dynamically controllable and reformulates the plan for efficient dynamic execution. (Vidal & Fargier 1999) introduced a polynomial time algorithm to check for strong controllability. Waypoint controllability, introduced by (Morris & Muscettola 1999), combines the properties of strong and weak controllability. In this framework, a sub-

set of the timepoints, designated as waypoints, are scheduled prior to knowing the uncertain durations; the remaining timepoints are scheduled once all of the uncertainty in the plan has been resolved. This provides a means to partition a partially controllable plan; however, it does not enable the agents to adapt to the uncertainty at execution time.

Our two layer approach is similar to waypoint controllability; however, in this paper we seek a execution strategy that combines strong with dynamic controllability instead of strong with weak controllability.

In the area of collaborative multi-agent planning and scheduling, (Hunsberger 2002) presented a Temporal Decoupling Algorithm (TDA) that solved the Temporal Decoupling Problems for STNs. The algorithm adds constraints to a Simple Temporal Network in order to ensure that agents working on different tasks may operate independently. In this paper we extends this work to Simple Temporal Networks with Uncertainty.

This paper presents a two layer approach and corresponding hierarchical reformulation (HR) algorithm that mitigates the need for communication at execution time between loosely coupled agents, while enabling tightly coupled agents to dynamically adapt to uncertainty. In particular, the algorithm preserves the flexibility in places where tight coordination is required and decouples the plan in places of loose coordination.

Our two layer approach is a pragmatic solution to what we term the *communication controllability problem* for STNUs. Informally, a plan in communication controllable if there is a viable multi-agent execution strategy in all situations that only depends on the observable past. Due to communication limitations, information of past outcomes and scheduling decisions may be either delayed or completely unobservable. To further complicate matters, each agents observable past is dependent not only on the outcomes of the uncertain durations but also on each agents scheduling decisions.

One main challenge in using our two layer approach is effectively modeling group plans within the mission plan. In this paper we present two approaches. The first approach preserves maximum flexibility in the group plans at the cost of completeness. In the second approach, we enable the group plans to give up some flexibility in order to satisfy the decoupling requirement at the mission layer. In both approaches it is possible to scheduled the start of each group plan off-line, without knowing the uncertain durations; however, in order to be robust to unmodeled uncertainty we keep the start time of each group plan flexible.

Overall Approach

In this paper, we assume the full multi-agent plan is divided into a set of tightly coordinated group plans. Furthermore, we assume that the agents that participate in these group plans are free to communicate with one another; however, the agents may not be capable of communicating outside their group. These group plans are loosely coupled via a higher level mission plan. The mission plan uses a simplified abstraction for each group plan that hides the details of the group plan. This encapsulation enables the reformulation algorithm to reason about group level interactions, without getting into the details of the group plans.

Our overall approach is presented in Figure 1.The multi-agent plan is formulated as a two-layer plan multi-agent plan

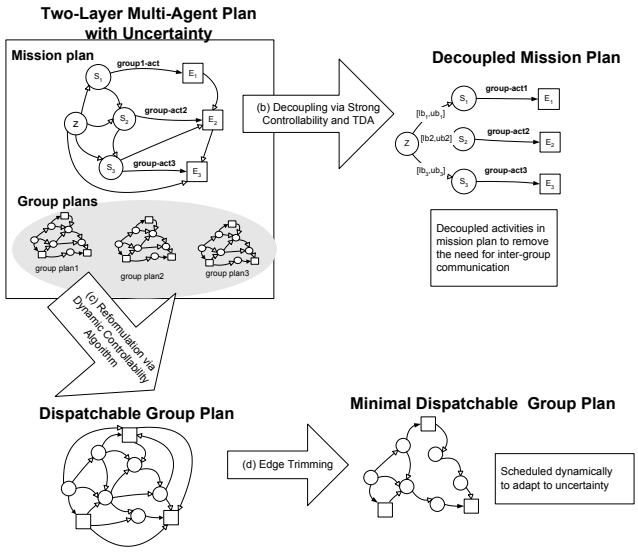


Figure 1: Hierarchical Reformulation Algorithm Overview

with uncertainty as shown in Figure 1a. The Hierarchical reformulation (HR) algorithm, converts the two-layer plan into a set of decoupled, minimal dispatchable group plans. The HR algorithm decouples each group plan, using the mission plan, by applying our novel STNU decoupling algorithm that combines properties the Strong Controllability Algorithm (SCA) introduced by (Vidal 2000) with the STN Temporal Decoupling Algorithm (TDA) introduced by (Hunsberger 2002), as shown in (Figure 1b). The group plans are reformulated using the dynamic controllability algorithm (Morris, Muscettola, & Vidal 2001) as shown in (Figure 1c). We also apply an edge trimming algorithm (Tsamardinos, Muscettola, & Morris 1998) to resulting dispatchable group plan (Figure 1d), in order to remove the redundant constraints. After applying the hierarchical reformulation algorithm, each group plan may be executed independently using the dispatching algorithm presented by (Morris, Muscettola, & Vidal 2001).

In this paper, we first we formalize our notion of a two-layer plan. Then, we present the high level structure of the HR reformulation algorithm followed by a detailed description of the decoupling algorithm. We end with a discussion of the HR algorithm and directions for future work.

Formal Definition of Two-Layer Plans

In this section we formally define a two-layer multi-agent plan with uncertainty and communication constraints. Then we describe our simplifying assumptions used in this paper.

In general, a two-layer multi-agent plan consists of a high level mission plan, and a set of lower level group plans. The mission plan consists of a set of uncontrollable activities that corresponds to the set of group plans along with a set of constraints. The mission plan contains a start timepoint, Z, that is always execute at T = 0. The group plans specify the details of each group activity. Specifically, each group plan contains a set of activities (both controllable or uncontrollable) to be performed the group of agents and a set of the

temporal constraints on those activities.

We formalize the our two-layer plans as a two-layer Multi-Agent Temporal Plan Network with Uncertainty (2L-MTPNU). A TPN is a set of activities, A, to be performed, each of which includes a start time, s_i, and end time, e_i, together with a set of simple temporal constraints that specify the valid activity start times, S and end times, E for the activities, A. Hence a TPN is a generalization of a STN (Dechter, Meiri, & Pearl 1995) that also contains of a set of activities A, and a mappings, $T^+ : S \rightarrow N^+$, and $T^- : E \rightarrow N^-$, mapping the start and end times of each activity to the timepoints in the STN. A TPN under uncertainty (TPNU) is analogous, where the temporal constraints are of the plan expressed as a STNU (**?**). In this case, the duration of each activity is either controllable and expressed as a requirement link, or uncontrollable and expressed as a contingent link.

A multi-agent TPNU (MTPNU) extends the TPNU in two ways. First, a MTPNU introduces a set of agents, Q, and a distribution, $D : N \rightarrow Q$, mapping the timepoints to agents. We assume that the start and end timepoints associated with each activity are mapped to the same agent. Second, a MTPNU contains a *communication availability graph* (CAG), which specifies when the agents are capable of communicating with one another. In general, a CAG, $C = < \Pi, T, U >$, consists of a set of states, $\pi_i \in \Pi$ for each agent $q_i \in Q$, a set of state transitions $t_i \in T$ with transition guards that specify how the agents transition between states, and a set of undirected communication availability edges,U, connecting states of different agents. We say that reliable communication exists between states π_i and π_j if there is an edge $u_i \in U$.

A two-layer MTPNU is an extension of a MTPNU. The two-layer MPTNU = $< M, G, B >$, where M is the high level mission plan, G is a set of group plans, and B is a function mapping the group activities, $a_i \in A$ in the mission plan to a group plan $g_i \in G$. Both the mission plan and group plans are modeled as a MPTNU; however, Q, in the mission plan are a set of groups, whereas, Q in the group plans are a set of agents.

In this paper we introduce several simplifying assumptions with respect to the two-layer MTPNU. First, we assume that the mission plan's CAG specifies that each group is unable to communicate with one another during execution. Second, we assume the CAG of each group plan is fully connected, meaning each agent is able to communicate with all other agents that participates in the same group plan. Third, we assume the mission plan consists only of the Z timepoint and timepoints associated with the start and end of each group activity.

Consider the simple two-layer plan illustrated in Figure 2. The mission plan (shown in Figure 2a) contains two group activities: group act1 and group act2. The corresponding group plans are shown Figure 2b,c. Both of the group plans consist of three timepoints and one contingent activity.

Hierarchical Reformulation Algorithm

In this section, we present our novel Hierarchical Reformulation (HR) algorithm. The HR algorithm is a centralized reformulation algorithm that transforms a two-layer MTPNU into a set of decoupled, minimally dispatchable group plans.

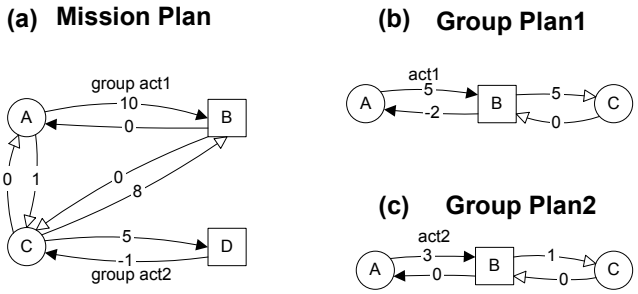


Figure 2: (a) The simple two-layer mission plan, (b) group plan1 (c) group plan2.

After running the HR algorithm, each group is able to execute their plan independently.

The HR algorithm operates on both layers of the two-layer plan. The dynamic controllability algorithm operates on the group plans, whereas, the decoupling algorithm, based on the strong controllability algorithm and STN temporal decoupling algorithm, operates on the mission plan.

The pseudo-code for the HR algorithm is shown in Figure 1. The algorithm takes in a two-layer MTPNU, P = $\langle M, G, B \rangle$, consisting of a mission plan, M, and a set of group plans, G, and mapping B and generates a set of decoupled, dispatchable MTPNUs. The algorithm returns true if the reformulation succeeds; otherwise, false.

The HR algorithm may fail for several reasons. The HR algorithm fails if either the mission plan or group plans are temporally inconsistent. Furthermore, the HR algorithm fails if the group plans are not dynamically controllable or if the mission plan is not strongly controllable.

Lines 1-3 of the HR algorithm, shown in 1, calls the UPDATE_GROUP_ACTIVITIES function and returns false if the update reveals a temporal inconsistency in any of the group plans. This function is called at the beginning of the HR algorithm, in order to synchronize the mission plan with the constraints specified in the group plans.

Alg. 1 HIERARCHICAL_REFORMULATION(P)

```
1: consistent ← UPDATE_GROUP_ACTIVITIES(G,M)
2: if ¬ consistent then
3:    return FALSE
4: end if
5: consistent ← COMPUTE_APSP_GRAPH(M)
6: if ¬ consistent then
7:    return FALSE
8: end if
9: UPDATE_GROUP_PLANS(G,M)
10: for each g ∈ G do
11:    controllable ← DC(g)
12:    if ¬ controllable then
13:       return FALSE
14:    end if
15: end for
16: UPDATE_GROUP_ACTIVITIES(G,M)
17: success ← DECOUPLE(M,G)
18: return success
```

Alg. 2 UPDATE_GROUP_ACTIVITIES(M,G)

```
1: for each group plan g ∈ G do
2:    s ← start timepoint of g
3:    if ¬ BELLMAN_FORD_SSSP(g,s) then
4:       return FALSE
5:    end if
6:    ub ← max( d[n] for each n ∈ N[g])
7:    BELLMAN_FORD_SDSP(g,s)
8:    lb ← -min( d[n] for each n ∈ N[g])
9:    groupactivity ← GET_GROUP_ACTIVITY(g)
10:   UPDATE_EDGE(M,START[m], END[m], ub)
11:   UPDATE_EDGE(M,END[m], START[m], -lb)
12: end for
13: return TRUE
```

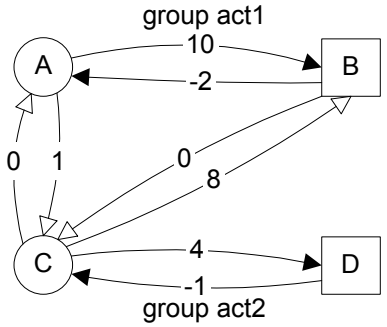


Figure 3: The UPDATE_GROUP_ACTIVITIES function updates the edges associated with the group activities in the mission plan. AB is updated to 9 and CD is updated to 4.

The UPDATE_GROUP_ACTIVITIES function first computes the feasible duration of each group plan. Next, it updates the contingent timebounds of the corresponding contingent bounds in the mission plan. The pseudo code for the UPDATE_GROUP_ACTIVITIES function is shown 2. The feasible durations are computed by calling two Bellman-Ford Single-Source Shortest-Path (SSSP) computations (Dechter, Meiri, & Pearl 1995) (Cormen, Leiserson, & Rivest 1990). If the SSSP computation detects an inconsistency in any of the group plans, the algorithm returns false, otherwise true.

For example, the two-layer plan shown in Figure 2. For group plan1, the maximum SSSP distance is 10 for the path ABC, and the minimum SDSP is -2 for the path CBA. The UPDATE_GROUP_ACTIVITIES function leaves the distance of the contingent edge AB in the mission plan at 10; however, the distance of the contingent edge BA is updated to -2. For group plan2, the maximum SSSP distance is 4 for the path ABC, and the minimum SDSP is 0 for the path CBA. For this group plan, the UPDATE_GROUP_ACTIVITIES function updates the distance of the mission plan's contingent edge CD to 4, while the contingent edge DC remains at -1. Both group plans are temporally consistent; therefore, the UPDATE_GROUP_ACTIVITIES function returns true. Figure 3 shows the updated mission plan after calling the UPDATE_GROUP_ACTIVITIES in the HR algorithm.

Lines 4-7 of the HR algorithm computes the All-Pairs Shortest-Path graph (APSP-graph) of the mission plan's distance graph (returning false if the mission plan is temporally inconsistent). Then the HR algorithm updates the timebounds of the group plans if the edges associated with the group activities are tightened by the APSP-graph. The COMPUTE_APSP_GRAPH function in Line 4 computes the APSP-graph, given the mission plan's distance graph. This APSP-graph is maintained separate from the mission plan's distance graph. The APSP computation is performed by either Johnson's algorithm or Floyd-Warshall's algorithm (Cormen, Leiserson, & Rivest 1990).

The APSP-graph is computed for two purposes. First, it checks if the mission plan is temporally consistent. If the mission plan is inconsistent, then the algorithm returns false in Line 6. Second, the APSP-graph is used to deduce any tightenings on the group activities implied by the constraints in the mission plan's distance graph. If the edges in the APSP-graph, corresponding to the group activity edges, are tightened, then the HR algorithm updates the corresponding group plan. The group plans are updated by calling the UPDATE_GROUP_PLANS function, in Line 7 of the HR algorithm.

Alg. 3 UPDATE_GROUP_PLANS(M,G)

```
1: for each group activity m ∈ Group_activities[M] do
2:    s ← START(m)
3:    e ← END(m)
4:    ub ← M.APSPgraph[s, e]
5:    lb ← -M.APSPgraph[e, s]
6:    UPDATE_EDGE(M,s,e,ub)
7:    UPDATE_EDGE(M,e,s,−lb)
8:    g ← GROUP(m)
9:    s ← START(g)
10:   e ← END(g)
11:   UPDATE_EDGE(g,s,e,ub)
12:   UPDATE_EDGE(g,e,s,−lb)
13: end for
```

The pseudo code for the UPDATE_GROUP_PLANS function is shown in Figure 1-24. This function loops through each group activity in the mission plan and updates the bounds in the corresponding group plan.

For example, the mission plan's APSP-graph is shown in Figure 4. The APSP-graph edge AB is smaller than the edge AB in the mission plan's distance graph. This edge AB is associated with the upper bound on the group act1. The edge AB is tightened from 10 to 9. The UPDATE_GROUP_PLANS function updates the mission plan's distance graph accordingly, as shown in Figure 4b. The UPDATE_GROUP_PLAN function then updates the group plans. The updated group plans are shown in Figure 4(c-d). The UPDATE_GROUP_PLANS function adds the edge AC = 9 to group plan1, corresponding to the edge AB = 9 in the mission plan, and adds the edge CA = -1 to group plan2, corresponding to the edge DC = -1 in the mission plan. Note that the edge DC = -1 was present in the original mission plan, whereas the edge AB = 9 was derived by the APSP-graph.

After the group plan's timebounds are updated, the HR algorithm calls the dynamic controllability (DC) algorithm

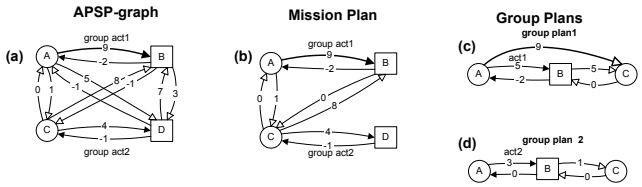


Figure 4: (a) mission plan's APSP-graph (b) Updated mission plan (c) Updated group plan 1 (d) Updated group plan 2.

(Morris, Muscettola, & Vidal 2001), in order to reformulate each group plan into a minimal dispatchable group plan, on Line 9. If this reformulation succeeds, then the group plan is dynamically controllable and the HR algorithm continues. However, if the DC algorithm fails (for any group plan), then the HR algorithm terminates and returns FALSE.

The complete description of the DC algorithm is presented in (Morris, Muscettola, & Vidal 2001). For now the reader only needs to understand that the DC algorithm is a reformulation algorithm that either adds or tightens the constraints of the group plan. These additional constraints may alter the range of feasible durations of the group plan. If the range of feasible durations of a group plan is tightened (the lower bound is increased or the upper bound is decreased), then the HR updates the edges of the corresponding group activity by once again calling UPDATE_GROUP_ACTIVITIES. This is done in Line 14. Note that tightening the constraints of the group activities only serves to remove uncertainty from the mission plan. Thus, the update performed in Line 14 only serves to make the decoupling algorithm more likely to succeed.

In our simple example, both of the group plans are dynamically controllable. Furthermore, feasible durations of the group plans are unchanged by the DC algorithm; therefore, the UPDATE_GROUP_ACTIVITIES call in Line 14 of the HR algorithm does not change the mission plan.

In Line 15, the HR algorithm calls the decoupling algorithm on the mission plan. The decoupling algorithm fixes the schedule for the start of each group plan. If the decoupling algorithm succeeds, then the HR algorithm returns true; otherwise, the HR algorithm returns false.

The Decoupling Algorithm

In this section we describe the *decoupling algorithm*, which temporally decouples each group activity in the mission plan. The effect of decoupling the group activities in the mission plan is that each group plan may be scheduled independently. The simplest method to perform this decoupling is to use a slight variation of the strong controllability algorithm, introduced by (Vidal 2000). Figure (**?**) shows the decoupling procedure. First, the strong controllability algorithm decouples the executable timepoint from the contingent timepoints, by making all requirement edges, connecting contingent timepoints, dominated (redundant). Next, the decoupling algorithm selects a consistent assignment to the executable timepoints in the mission plan.

This decoupling algorithm operates on the mission plan, in order to generate a fixed schedule for the start timepoint of each group activity. These fixed start times are then passed

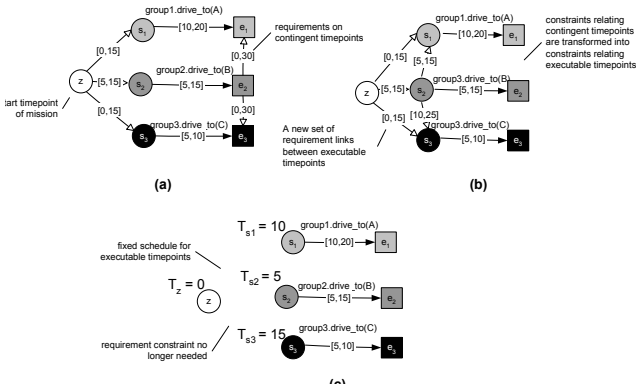


Figure 5: (a) The original mission plan containing requirement edges connecting contingent timepoints. (b) The mission plan after the contingent timepoints are decoupled by the strong controllability algorithm. Note, all requirement edge connecting contingent timepoints are removed. (c) The decoupling algorithm fixes the start time for each executable timepoints. This eliminates the need to propagate scheduling times during execution.

to their respective group plans. The resulting group plans can be scheduled independently. The decoupling builds upon the strong controllability checking algorithm (Vidal & Fargier 1999). The decoupling algorithm transforms the distance graph of the mission plan using the strong controllability transformation rules. If this transformed graph is consistent, the decoupling algorithm generates a schedule for the timepoints of the transformed graph. Note that any consistent schedule would work; however, we elect to schedule the group activities as early as possible. This schedule is used to fix the time of the corresponding group plans.

The pseudo-code for the decoupling algorithm is shown in Figure 6. The algorithm takes in a two-layer plan, consisting of a mission plan, M, and a set of group plans, G and fixes the schedule for the mission plan.

The decoupling algorithm runs in polynomial time. Lines 1-13 run in the same time as the strong controllability algorithm (i.e. O(NE)). In Lines 14-20, the decoupling algorithm loops through each timepoint and fixes the start time of each group plan. Using a simple lookup, the GET_GROUP_ACTIVITY and GET_GROUP_PLAN run in time linear in the number of group plans. Therefore, Lines 14-20 run in O(NG), where G is the number of group plans. The number of group plans G is less than the number of edges in the distance graph; therefore, the decoupling algorithm is dominated by the Bellman-Ford SDSP computation. The running time of the decoupling algorithm is O(NE).

For our simple example, the decoupling algorithm succeeds. The decoupling algorithm is applied to the updated mission plan, as shown in Figure 14(b). The distance graph of the mission plan is converted into the transformed STN, as shown in Figure 15(a). The decoupling algorithm first copies over the executable timepoints, A and C, then it transforms the edges, using the strong controllability transformation rules. The decoupling algorithm copies over the requirement edges AC = 1 and CA = 0 from the mission plans

Alg. 4 DECOUPLE(M,G) $Input$: A mission plan M and a set of group plans G. $Effects$: Decouples the group plans by fixing the start time of each group plan. $Output$: $True$ mission plan is strongly controllable; otherwise, $False$.

```
1: G_m ← get distance graph of mission plan
2: copy all executable timepoints of G_m to T
3: initialize all edges of T to NIL
4: for each requirement edge (u,v) ∈ E[G_m] do
5:    transform the edge (u,v) using SC transformation
      rules to and edge (u',v') with d(u',v') = x
6:    UPDATE_EDGE(T,u',v',x)
7: end for
8: s ← start timepoint of T
9: consistent ← BELLMAN_FORD_SDSP(T,s)
10: if ¬ consistent then
11:    return FALSE
12: else
13:    for each timepoint n ∈ N[T] do
14:       m ← GET_GROUP_ACTIVITY(n)
15:       if m ¬ NIL then
16:          g ← GROUP_PLAN(m)
17:          fix start time of g to -d[n] as computed by line
             10
18:       end if
19:    end for
20: end if
21: return TRUE
```

distance graph. The edge CB = 8 is transformed into an edge CA = -1, which relaxes the edge CA in the transformed STN. The edge BC = 0 is transformed into an edge AC = 2, which is greater than the existing edge, so there is no change in the transformed STN. Finally, the decoupling algorithm computes the earliest execution time for each timepoint, using an SDSP computation. The earliest execution time for A = 0 and B = 1, and, therefore, the start timepoint associated with group plan1, is fixed at 0, and the start timepoint for group plan2 is fixed at 1. The decoupled group plans are shown in Figure 6(b,c).

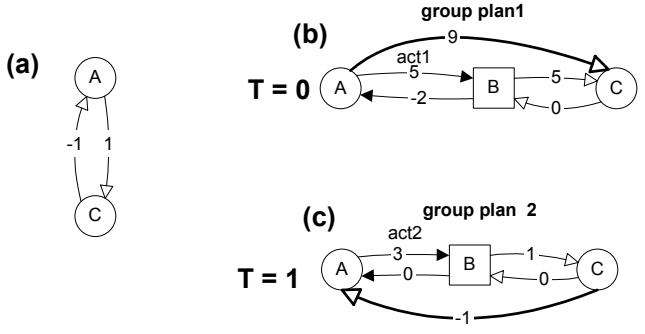


Figure 6: (a) The transformed STN (b) The start time of group plan1 is fixed at T = 0 (c) The start time of group plan2 is fixed at T = 1.

Discussion

After running the HR algorithm on the two-layer MTPNU, each group is able to efficiently execute the plan by using the dispatching algorithm presented by (Morris, Muscettola, & Vidal 2001).

The HR algorithm is a polynomial time algorithm. It gains efficiency by dividing the reformulation problem into a set of smaller sub-problems.

Consider the runtime complexity of the HR algorithm. In this discussion, we use the following notation.

- G = number of group plans.
- N_m = number of timepoints in the mission plan.
- E_m = number of edges in the mission plan.
- N_g = maximum number of timepoints in any group plan.
- E_g = maximum number of edges in any group plan.

In Line 1, HR calls the UPDATE_GROUP_ACTIVITIES function. The UPDATE_GROUP_ACTIVITIES function loops through each group plan and the time of each loop is dominated by the Bellman-Ford algorithm. Therefore, the UPDATE_GROUP_ACTIVITIES runs in O($G * N_g * E_g$). Lines 2-3 of the HR algorithm run in constant time. In Line 4, the HR algorithm calls COMPUTE_APSP_GRAPH. The Floyd-Warshall algorithm is used, which runs in (N_m^3). Lines 5-6 run in constant time. Line 7 calls the UPDATE_GROUP_PLANS function. The UPDATE_GROUP_PLANS function loops through each group activity and each loop is performed in constant time. Therefore, the UPDATE_GROUP_PLANS runs in O(G) time. Lines 8-13 of the HR algorithm loop through each group plan and calls the DC algorithm.

The time complexity of the DC algorithm is polynomial (Morris, Muscettola, & Vidal 2001) however, experimental results exhibit a running time of O(N_g^3). Given this, the running time of Lines 8-13 is experimentally shown to be O($G * N_g^3$). Line 14 calls the UPDATE_GROUP_ACTIVITIES function. Finally, in Line 15, the HR algorithm calls DECOUPLE, which runs in O($G * N_g$) time.

Adding the terms together, we get an expression for the running time of the HR algorithm as O($G * N_g * E_g$) + O(N_m^3) + O(G) + O($G * N_g^3$) + O($G * N_g$) + O(1), which can be simplified to O($G * N_g^3 + N_m^3$). The N_g^3 term is derived by an All-Pairs Shortest-Path (APSP) computation, applied to the group plan used in the DC algorithm. The N_m^3 term is due to the APSP computation on the mission plan.

The HR algorithm, presented in this paper, is unique in its ability to cope with both communication limitations and temporal uncertainty, by combining properties of strong and dynamic controllability. We believe this paper lays out a framework that will enable multi-agent systems to manage communication limitations in a pragmatic way.

References

Cormen, T. H.; Leiserson, C. E.; and Rivest, R. L. 1990. *Introduction to Algorithms*. Cambridge, MA: MIT Press.

Dechter; Meiri; and Pearl. 1995. Temporal constraint newworks. In *Artificial Intelligence*, 61–95.

Hunsberger, L. 2002. Algorithms for a temporal decoupling problem in multi-agent planning.

Morris, P. H., and Muscettola, N. 1999. Managing temporal uncertainty through waypoint controllability. In *IJCAI*, 1253–1258.

Morris, P.; Muscettola, N.; and Vidal, T. 2001. Dynamic control of plans with temporal uncertainty. In *IJCAI*, 494–502.

Muscettola, N.; Morris, P.; and Tsamardinos, I. 1998. Reformulating temporal plans for efficient execution. In *Principles of Knowledge Representation and Reasoning*, 444–452.

Tsamardinos, I.; Muscettola, N.; and Morris, P. 1998. Fast transformation of temporal plans for efficient execution. In *AAAI/IAAI*, 254–261.

Vidal, and Fargier. Dealing with uncertain durations in temporal constraint networks dedicated to planning. In *In Proc. of 12th European Conference on Artificial Intelligence (ECAI-96)*, 48–52.

Vidal, and Fargier. 1999. Handling contingency in temporal constraint networks: from consitency to controllabilities. In *Journal of Experimental and Theoretical Artificial Intelligence*, 23–45.

Vidal, T. 2000. Controllability characterization and checking in contingent temporal constraint newtorks. In *Proc. of Seveth Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'2000)*.

Computing the Communication Costs of Item Allocation

Timothy W. Rauenbusch
Artificial Intelligence Center
SRI International
Menlo Park, CA 94025
rauenbusch@ai.sri.com

Stuart M. Shieber and Barbara J. Grosz
Division of Engineering and Applied Sciences
Harvard University
Cambridge, MA 02138
shieber,grosz@eecs.harvard.edu

Abstract

Multiagent systems require techniques for effectively allocating resources or tasks to among agents in a group. Auctions are one method for structuring communication of agents' private values for the resource or task to a central decision maker. Different auction methods vary in their communication requirements. This paper makes three contributions to the understanding the types of group decision making for which auctions are apprpriate methods. First, it shows that entropy is the best measure of communication bandwidth used by an auction in messages bidders send *and* receive. Second, it presents a method for measuring bandwidth usage; the dialogue trees used for this computation are a new and compact representation of the probablity distribution of every possible dialogue between two agents. Third, it presents new guidelines for choosing the best auction, guidelines which differ significantly from recommendations in prior work. The new guidelines are based on detailed analysis of the communication requirements of Sealed-bid, Dutch, Staged, Japanese, and Bisection auctions. In contradistinction to previous work, the guidelines show that the auction that minimizes bandwidth depends on both the number of bidders and the sample space from which bidders' valuations are drawn.

1 Introduction

Multiagent system designers can achieve significant cost savings by making the correct choice of algorithm for team decision making. The results in this paper show that no single auction type minimizes bandwidth usage for all team sizes or for all possible valuations for the resource. For instance, Sealed-bid auctions require the least communication for small problems. The Dutch, Staged, and Bisection auctions each require least communication in some situations.

A Sealed-bid auction requires each bidder and the auctioneer to exchange 5 bits of information in a system with 60 agents where each agent's valuation is drawn independently and uniformly from the range \$1 to \$32. A Dutch auction requires an exchange of approximately one bit on average under the same assumptions. A difference of four bits of information may seem insignificant by today's standards but modern systems may make millions or billions of related team decisions every second. While sacrificing no team decision quality, a system designer could save over 80 percent of its communication bandwidth just by implementing a different set of auction rules.

Previous work has made recommendations for the best choice of auction for making group decisions. However, the assumptions that led to those recommendations are incompatible with real systems in which communication bandwidth is costly, such as those using Internet-like networks.

This paper makes three main contributions to the understanding of communication for decision making in multiagent systems. First, we argue for entropy as the metric of communication bandwidth used by all messages exchanged. Communication in any multiagent system is made up of a series of messages that one agent sends to another. System designers need to choose an encoding for messages. For example, the number nine is commonly given the binary encoding "1001" but in ASCII code it is assigned the binary encoding "0011 1001". Measuring communication in decision-making algorithms using a particular message encoding could lead to results that are applicable only for that encoding. This paper uses principles of Information Theory to measure information in a coding-independent way. The receiver of a message can generate a probability distribution over the set of possible messages it can receive. The entropy of that distribution is a lower bound on the average size of the encoding for each message.

Second, we provide details of a three-step method for measuring bandwidth used by an algorithm. In the first step, the analyst builds a dialogue tree that represents all possible sequences of messages exchanged between the auctioneer and each bidder. In the second step, the edges of the dialogue tree are labeled with the probability associated with each message. Finally, in the third step, the expected information in the dialogue is calculated using the tree representation.

Third, we apply the analysis to Sealed-bid, Dutch (descending), Japanese (ascending), Staged (ascending), and Bisection auctions and provides system designers with the knowledge necessary to choose the auction that minimizes communication bandwidth. Auctions are particularly attractive for multiagent decision making because they provide a

way to structure the allocation of a resource or task to the member of a multiagent system that values it most, when the resource's value is private to each group member. Equivalently, auctions are used to assign a task to the member of a group that is best suited to perform it when the suitability of each group member to the task is private [Hunsberger and Grosz, 2000; Rauenbusch, 2004].

Our recommendations, based on a minimizing communication requirements, differ from those of economists and computer scientists. Economic analysis typically ignores communication costs entirely. Some computer scientists [Shoham and Tennenholtz, 2001] have focused on preference revelation, which concerns the willingness to disclose information. They consider only those messages sent from a bidder to an auctioneer and ignore message sent in the opposite direction. Some researchers [Grigorieva *et al.*, 2002] have used communication complexity or other metrics that assume a particular message encoding. Their results may be misleading for measuring bandwidth requirements in systems that employ more efficient encodings—our results are coding-independent.

This paper is organized as follows. In Section 2 the single item allocation problem is formally defined, and the five auctions are described. Next, Section 3 details the process for measuring communication in a dialogue using Dialogue Trees. Section 4 describes the application of dialogue trees to auctions. Guidelines for system designers choosing auction rules that minimize communication are given in Section 5. Section 6 highlights important related work and Section 7 gives conclusions and suggests areas for future work.

2 Item Allocation and Auctions

A single-item allocation problem is characterized by a group of n bidder agents and a seller agent (also called the auctioneer) that possesses a single, atomic item. Each bidder has a value for the item that is private and drawn independently and uniformly from the set of integers from 0 to $2^R - 1$ inclusive. Another way to look at a bidder's value is that it is being drawn from one of 2^R bins. The distribution from which each bidder's value is drawn is common knowledge. Bidder i's value is denoted by x_i. The goal of the seller is to allocate the task to the bidder with the highest value. If there is a tie for the highest value, the task may be allocated to any of the bidders with the highest value. A *solution* to a single-item allocation problem is the index i, where x_i is the maximum value among all n bidders.

We analyze five auction types: Sealed-bid, Japanese, Staged, Dutch, and Bisection. This particular list of five auction types is representative of the range of auctions typically used to allocate a single item and is not intended to be exhaustive. For reference, the rest of this section provides a description of each auction type. Rauenbusch [2004] provides more detail, including pseudocode for each. In each auction, we assume bidders are honest. Prices are used to structure communication with the bidders and not as a tool for building in incentives for honesty.

Sealed-bid. All bidders send their value to the auctioneer. The winner is the bidder that sends the highest value.

Japanese (Ascending). The auctioneer maintains a *current price*, initially set to 0. The auctioneer sends each bidder in turn the current price. If a bidder's value is greater than or equal to the current price, it sends a message affirming its continued participation in the auction. Otherwise, it sends a message indicating its desire to leave the auction. The auctioneer then increments the current price, and repeats the process. If only one participating bidder remains in the auction after a round, the auction ends and that remaining bidder is the winner. If no participating bidders remain, the winner is chosen from the bidders in the previous round. Once a bidder leaves the auction, it may not rejoin.

Staged (Ascending). The auctioneer maintains a *current price*, initially set to 0. In Stage 1, the auctioneer sends bidder 1 the current price. If the bidder's value is greater than or equal to the current price, it sends its value and the current price is updated to this value. Otherwise, it sends a message indicating its desire to leave the auction. The auctioneer then moves on to Stage 2, sends the current price to bidder 2, and the process repeats. The auctioneer continues in this way with each bidder and the process ends after the nth stage. The winner is the last bidder that did not leave the auction.

Dutch (Descending). The auctioneer maintains a *current price*, initially set to $2^R - 1$. The auctioneer sends each bidder in turn the current price. The bidder sends a message indicating whether its value is equal to the current price. If no bidder's value is equal to the current price, the auctioneer decrements the price and repeats. If one or more bidder has value equal to the current price, the auctioneer chooses one as the winner.

Bisection. The auctioneer maintains an *lower bound* denoted l and *upper bound* denoted u, initially set to 0 and 2^R, respectively. The auctioneer also maintains a list of active bidders, initially the set of all bidders. The auctioneer calculates the *current price* as $u - \frac{u-l}{2}$. The auctioneer sends each bidder in turn the current price. Each bidder sends a message of either "Yes" or "No" to indicate whether its value is greater than or equal to the current price. If there are two or more bidders that sent a "Yes" message, the lower bound is updated to the current price, the set of active bidders updated to include only those that sent a "Yes" message, and the process repeats. If no bidder sent a "Yes" message, the upper bound is updated with the current price and the procedure repeats. If one bidder sent a "Yes" message, that bidder is declared the winner and the procedure ends. If the upper bound and lower bound differ by only one, one of the active bidders is chosen as the winner. After finding a winner, typically the bisection auction may proceed into a "price determination" phase that provides incentives for honesty. Because we assume honesty, the price determination phase is omitted from our analysis.

	Encoding		Probability	
Message	Enc1	Enc2	AlgA	AlgB
a	0000	0	0.0625	0.99
b	0001	10001	0.0625	0.000333
c	0010	10010	0.0625	0.000333
d	0011	10011	0.0625	0.000333
...	...	...	...	...
p	0011	10011	0.0625	0.000333

Table 1: Two encodings for sixteen messages used by Algorithms AlgA and AlgB

3 Communication Properties of a Dialogue

This section serves three main purposes. First, it presents an argument for the use of entropy and information theory to measure communication for team decision making. Second, it highlights the need to consider all communication. In auctions, this means that complete analysis requires evaluating communication in two directions: both from the bidders to the auctioneer and from the auctioneer to the bidders. Third, it presents dialogue trees—a tool for using entropy to measure the expected information transmitted in successive messages between agents. It details the use of dialogue trees in measuring communication for team decision making.

3.1 Entropy: Metric for Measuring Communication

A metric for measuring communication is required to compare auction rules by their communication cost. In each auction, information is exchanged between the auctioneer and each bidder by sending and receiving messages. In any implementation of an auction, the center and the bidders must agree to an encoding of messages.

Measuring information required by a multiagent algorithm using a particular encoding for messages may lead to misleading results. To illustrate why, we refer to the example given in Table 1. The columns labeled Enc1 and Enc2 shows two possible encodings for each of sixteen messages labeled *a* through *p*. Two algorithms, labeled AlgA and AlgB, each require one of sixteen messages to be sent from one agent to another but they differ in the frequency with which each message is sent. The probability associated with each message for each algorithm is shown in the two rightmost columns of the table.

With encoding Enc1, both AlgA and AlgB require four bits to transmit the message. But with encoding Enc2, AlgA requires 4.75 bits and AlgB requires 1.04 bits in expectation. Therefore, the algorithm that requires the least communication depends on the encoding chosen. Just as in this toy example, conclusions about the communication properties of auctions using a particular encoding are misleading because it is not clear whether those conclusions hold for other possible encodings. Work in Information Theory [Shannon, 1948; Cover and Thomas, 1991] has shown that the entropy of a random variable describing a message is a lower bound on the average size of the encoding for that message. Rather than evaluate an algorithm using a particular encoding, we therefore use entropy to measure expected information communicated.

3.2 Direction of Communication

It is convenient to distinguish between *coordination messages*, which are those sent by the auctioneer to a bidder, from *revelation messages*, which are those sent by the bidder to the auctioneer. In this paper, the communication costs associated with coordination and with revelation are considered when calculating the expected information transmitted in an auction. In particular, the results provided are for the sum of coordination and revelation costs. This assumption is supported by Internet-like computer networks in which increased bandwidth requires costs associated with increased infrastructure for both directions of communication.

In a Sealed-bid auction, each bidder always reveals its value. Therefore, Sealed-bid auctions have the highest bandwidth requirements for revelation messages. As the results in Section 5 indicate, it would be misleading to rely on revelation messages alone when choosing an auction. Even though Sealed-bid auctions require more information transmitted in revelation than any other auction, they require no coordination. For that reason, they have low communication requirements in settings with small teams and coarse distributions from which bidders' values are drawn.

3.3 Dialogue Trees

A *dialogue* is a sequence of messages sent from one agent to another agent, in which the agent that sends the odd-numbered messages receives the even-numbered messages. Dialogue trees simplify the construction of a probabilistic model of the messages. In this section, we describe dialogue trees and provide a detailed method for calculating the expected information in a dialogue. We use dialogue trees to measure expected information in an auctions by analyzing the dialogue between the auctioneer and each bidder. Dialogue trees apply equally to other dialogues and are not limited to analysis of auctions.

A *dialogue tree* is a tree data structure with labeled edges. Each node represents a message, and is labeled with the message it represents. *Query messages* are those sent by the auctioneer to request a message from the bidder; *reply messages* are those sent by the bidder. *Status messages* are those sent by the auctioneer to which no reply is expected. Figures that represent dialogue trees (such as Figure 1) show query nodes, reply nodes, and status nodes enclosed by circles, boxes, and diamonds, respectively. $\mathrm{Nodes}(d)$ denotes the set of all nodes in dialogue tree d.

The children of a node in a dialogue tree represent the sample space from which the next message is drawn, given that the message represented by the parent node has been sent. $\mathrm{Children}(m)$ denotes the set of child nodes of node m. $\mathrm{Parent}(m)$ denotes the parent node of node m.

A label on an edge between a parent and child node indicates the receiver's belief, prior to receiving the message, that the message represented by the child node is the one that the sender will send. $\mathrm{In}(m)$ denotes the edge label that is incident on node m in a dialogue tree.

The edge labels define a probability distribution over the sample space represented by the children. The probability distribution and sample space together define a probabilistic model for messages in a dialogue.

In the auctions described in this paper, a bidder always sends a reply after receiving a query; therefore, a query node is never a leaf in a dialogue tree. A reply node may be either a leaf or a non-leaf node, depending on whether the center may follow the corresponding reply message with a message. A status node is always a leaf in a dialogue tree.

The remainder of this section details how a dialogue tree is used to calculate the expected information in a dialogue. The procedure uses edge labels for two purposes: to calculate the information content of a node and to calculate the probability of visiting a node.

The *information content* (IC) of node m is the entropy of the random variable represented by the labels of all edges originating at the node. Formally,

$$\mathrm{IC}(m) = -\sum_{c \in \mathrm{Children}(m)} \mathrm{In}(c) \log \mathrm{In}(c) \qquad (1)$$

A leaf node therefore has information content of 0.

A path from the root node to each leaf node represents every possible dialogue between the two agents. The amount of information in a dialogue is the sum of the information content in each node on the path. Each of the possible dialogues represented by a tree has a different probability of occurring. This probability is the product of the edge labels along the path of the dialogue from the root of the tree to a leaf. The *probability of visiting* (PV) node m (that is, the probability that a message represented by a particular node will be sent in a dialogue) is the product of the probability of the message represented by its parent node and the label on its incident edge. There is unit probability of visiting the root node. Formally,

$$\mathrm{PV}(m) = \begin{cases} 1 & \text{if } m \text{ is root} \\ \mathrm{PV}(\mathrm{Parent}(m)) \cdot \mathrm{In}(m) & \text{otherwise} \end{cases} \qquad (2)$$

The *contributed information* (CI) of a node m is the product of the amount of information represented by the node and the probability the node is visited. Formally,

$$\mathrm{CI}(m) = \mathrm{PV}(m)\mathrm{IC}(m) \qquad (3)$$

We use expected information in a dialogue as the metric for communication. Expected information of a dialogue (EI) represented by dialogue tree d is the sum of the contributed information of each node in d. Formally,

$$\mathrm{EI}(d) = \sum_{m \in \mathrm{Nodes}(d)} \mathrm{CI}(m) \qquad (4)$$

Contributed information provides a straightforward way to separate the information contribution of messages sent by the center from those sent by the bidder. The child nodes of a reply node represent messages sent by the center and the child nodes of a query node represent messages sent by the bidder. The amount of information sent by the bidder is the sum of the contributed information of all query nodes and the amount of information sent by the center is the sum of the contributed information of all reply nodes. This is counter-intuitive and arises because contributed information of each node is derived from the probabilities associated with the edges *originating* at that node, which define the information content of the messages represented by its child nodes. Section 4 describes the dialogue tree in Figure 1 and how it is used to analyze the Bisection auction.

4 Analysis of Auctions

Using dialogue trees as a tool, in each auction we first determine the structure of the tree, then calculate the appropriate edge labels. To aid in determining the structure of the tree, the messages in each of the five auctions are divided into the following two types of query/response pairs: (1) best response, and (2) value. In a best response query, the auctioneer sends the bidder a message that includes a price. The bidder then responds with the message *Yes* if its value is higher than the price and the message *No* otherwise. In a value query, the auctioneer sends a message, and the bidder responds by sending a message containing its value.

Decomposing these algorithms into two types of constituent query/response pairs is a tool used to simplify of the analysis. The measurement of the expected information in a dialogue for each auction is independent of this decomposition. For example, if a bidder in the Staged auction responds *Yes* when sent the first message, it always sends its value. It is therefore not necessary to send a query message for the bidder's value after receiving the response. But, there is zero communication cost for the value query (because the probability of sending it given a *Yes* response is 1).

Two methods are used to determine the edge labels. The first and simplest way to determine the edge labels is by simulation. An auction is run many times in simulation, and the frequency of each message is recorded and used for the edge labels. The main advantage of this approach is that it requires little labor, after coding the algorithm. One disadvantage of the simulation method is that the time required to run the many simulations needed to accurately estimate the frequency of low-probability messages usually found near the leaves of the dialogue tree may be prohibitive. In addition, this method requires a different simulation for each setting of parameters of interest. For example, the results given in Figure 2 would require 1220 sets of simulations: one for each of 122 team sizes and 10 settings for the number of bins.

The second method is to calculate the edge labels analytically. This approach uses the common knowledge from which the bidder's value is drawn, and the knowledge acquired through messages represented by higher levels of the tree. The main drawback with this approach is that it is labor intensive because an analyst must reason about the receiver's mental model for each message in each algorithm. The main advantage of this approach is that the procedure for generating edge labels in one particular setting (e.g., for a team of 20 agents and 4 bins) applies equally well to other settings (e.g., 21 agents and 8 bins) by substituting appropriate parameters. An additional advantage is that the edge labels are calculated precisely rather than estimated.

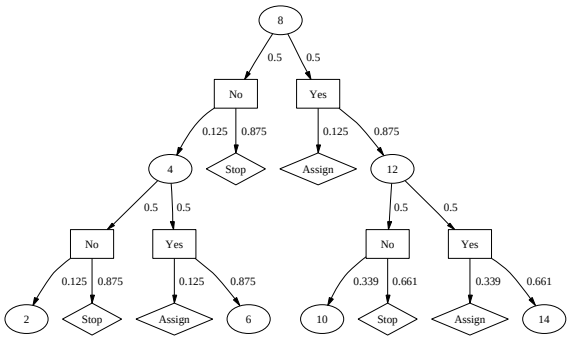


Figure 1: Highest three levels of a dialogue tree for Bisection auction with four bidders and sixteen bins

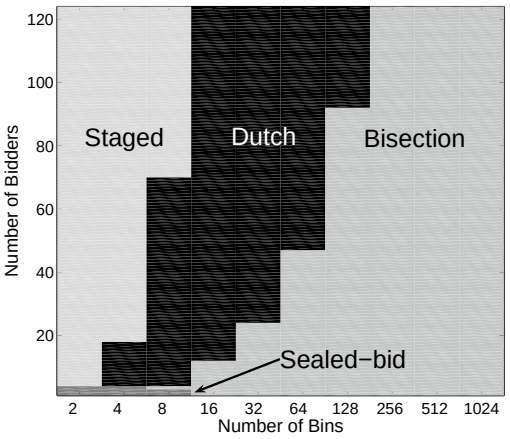


Figure 2: Algorithm with lowest expected information transmitted for varying numbers of bidders and bins

The results presented in this paper were based on generating edge labels using the second method. The first method was used to verify the results. The rest of this section provides an example of a dialogue tree for the Bisection auction to illustrate the use of dialogue trees to measure the expected amount of information transmitted in the five auctions. Details of the analysis have been omitted due to lack of space. Rauenbusch [2004] provides the details of the analysis of the dialogue trees for each auction.

The calculation of the edge labels in any dialogue tree involves reasoning about the knowledge of the receiver of each message: the distribution from which the bidder's value is drawn and all messages represented in higher levels of the tree. Figure 1 shows the dialogue tree that represents the first five messages exchanged between the auctioneer and one bidder in a Bisection auction. In the tree, the message containing the best response query with value b is represented by a query node with label b.

To provide an example of the reasoning involved in computing edge labels, we specifically consider the edges on the path from the root node labeled *8* to the leaf node labeled *14*. Calculation of edge labels in the figure assumes that there are four bidders, with values drawn from 16 bins—0 through 15 inclusive.

The root of the tree corresponds to the best response query with value 8. The bidder replies to this query with *Yes* if its value is greater than or equal to 8, and *No* otherwise. The receiver of the *Yes* or *No* message—the auctioneer—believes that the *Yes* message will be sent with probability 0.5 because it knows the distribution from which the bidder's value is drawn. Therefore, the edge into the *Yes* node is labeled 0.5.

To compute the next edges, labeled 0.125 and 0.875, we first assume that the bidder sent a *Yes* response to the first query. The bidder will win the auction (and will be sent a message indicating that it is assigned to the item) if and only if no other bidder sent a *Yes* response to the first query. Given the common knowledge that bidders' values are distributed uniformly between 0 and 15, the probability that all three other bidders sent a *Yes* query is $(0.5)^3 = 0.125$. Therefore, the edge incident on the *Assign* node is labeled 0.125, and the edge incident on the 12 query node is labeled with its complement 0.875.

The edges incident on the next reply nodes are labeled 0.5. The auctioneer knows (1) that the bidder's value was drawn uniformly from 0–15 by common knowledge; and (2) the bidder's value is greater than 8 by virtue of the *Yes* response represented in a higher level of the tree. Therefore, the auctioneer's believes that there is a probability of 0.5 that the bidder's value is higher than 12.

The calculation of the edge labeled 0.661, incident on the node labeled 14 in the tree, is complex and full details are omitted. To get a feeling for why, the analysis begins with the knowledge that given that the bidder sent the *Yes* message represented by the top of the edge, the message represented by the node labeled 14 will be sent if and only if at least one other bidder also has value greater than 12. But the bidder knows that at least one other bidder had value greater than 8. The calculation involves the bidder assigning a belief vector representing is belief that each of one, two, and three other bidders still remain in the auction. The value 0.661 is then computed using this vector.

5 Results

Figure 2 indicates the algorithm that has lowest expected information transmitted for increasing numbers of bidders and for increasing numbers of bins. It clearly shows that choosing the algorithm that needs least expected information transmission is highly dependent on the two parameters of the environment. For large numbers of bidders and bins, Bisection requires the least communication. Sealed-bid, Dutch, and Staged auctions each require the least communication for particular parameter settings.

For a very small number of bidders and bins (fewer than five bidders with two or four bins, and fewer than three bidders with eight bins) the Sealed-bid auction performs best. A sealed-bid auction by definition requires the maximum amount or revelation and no coordination. Therefore, for very small problems, the savings in revelation from any other auction method are outweighed by the cost of coordination.

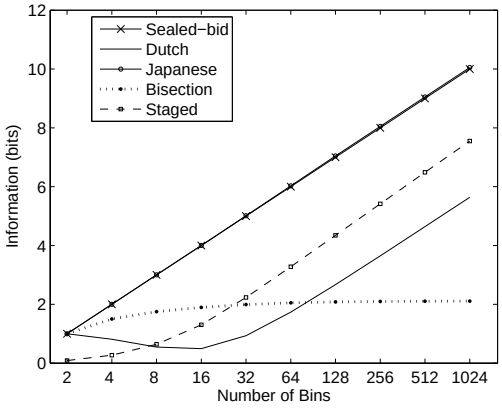


Figure 3: Expected information transmitted per bidder for varying numbers of bins with 60 bidders

When there are two bins, the Japanese auction has the same communication properties as the Sealed-bid auction because the first and only query in the Japanese auction is always sent and the bidder reveals its value (by its response that indicates whether its value is in the higher or lower bin).

For all but the smallest numbers of bidders and bins, the Bisection, Dutch, and Staged Japanese auctions perform well. The graph in Figure 3 shows the expected amount of information transmitted between the center and each bidder for a varying number of bins for a constant 60 bidders.

The first thing of note on the graph is that the communication requirements of the Sealed-bin auction increase linearly as the number of bins increases exponentially. The Sealed-bid auction has zero coordination cost and a revelation cost that is logarithmic in the number of bins.

The graph shows that as the number of bins increases exponentially, the expected amount of communication required by the Bisection auction rises then levels off. For a small number of bins, the Staged auction has very low communication requirements. For small numbers of bins, the Dutch auction's communication requirements actually decrease as the number of bins increase. Therefore, as the number of bins increases, the auction with the lowest communication costs is first the Staged auction, then the Dutch auction and finally the Bisection auction.

6 Related Work

Economic analysis of auctions [Rasmussen, 1989, inter alia] focuses on the effect of auction rules and prices on the strategies of non-cooperative bidders. While this paper is concerned with systems in which strategies can be imposed by methods external to the auction itself, dialogue trees can be used to measure communication requirements of all types of auctions. In multiagent systems where the assumption of externally imposed incentives does not hold, dialogue trees can be used to compare the communication costs of auctions that impose desirable incentives on the bidders.

Researchers in computer science have used several alternatives to entropy for measuring communication in multiagent decision making. One such approach counted the number of messages required to arrive at a team decision [Ortiz *et al.*, 2003], which is equivalent to assuming that each message has a fixed length. In systems with communication channels that carry encoded messages, the assumption that each message has a fixed length does not hold. Under a fixed length assumption, the Sealed-bid auction would always be preferred. Thus, such analyses may be misleading because an algorithm with fewer fixed-length messages will not always be the cheaper algorithm in terms of expected information transmitted.

Sunderham and Parkes [2003] measure the volume remaining in the space of feasible private information after bidders have sent the auctioneer constraints on their private information in a multi-attribute auction. They use this metric to compare the amount of revelation in auctions. For our purposes, entropy is a preferred metric because it provides a direct measure of bandwidth required by an auction and it provides the common currency of bits to measure both coordination and revelation.

Communication complexity [Kushilevitz and Nisan, 1996] provides an alternative method for analyzing communication between agents. Grigorieva et al. [2002] use communication complexity to analyze the bisection auction. Communication complexity evaluates the worst case amount of communication required for two agents to compute a function. The communication complexity model assumes that sending each binary message costs one bit. If any prior information is available, it is ignored for the purposes of calculating communication complexity. As long as there is *some* arbitrarily small possibility that an agent will send a '0', that communication costs one bit. Protocol trees [Yao, 1979] are used as a tool to evaluate communication complexity of an algorithm while dialogue trees are used to calculate expected information in a dialogue that represents messages sent in an algorithm.

The main benefit of this assumption is that there is no need to assume a prior distribution, and that simplifies the analysis. The main drawback is that it assumes a particular encoding of messages and therefore no savings can be attained by alternative encodings. A system designer that relies on communication complexity in choosing an auction will select an auction that performs well under a worst case assumption of the encoding cost *of each message*. In this paper, we assume that system designers prefer choosing an auction based on the expected information transmitted.

Shoham and Tennenholtz [2001] use a method related to communication complexity for the analysis of the functions computed in team decision-making mechanisms. They define f as the maximum value of n bidders' willingness to pay for an item, where each bidder i has a willingness to pay of x_i. They imply that the domain of x_i is continuous on the interval $(0, \mathit{maxprice})$ and assume that each bidder i can communicate x_i to the auctioneer with one bit by making use of a common clock. They claim that by using an auction similar to the Dutch auction, the function f can be computed by a single bidder communicating a single bit.

In both Yao's theory of communication complexity and Shannon's theory of information [Shannon, 1948], the cost of communicating an arbitrary value drawn from a continuous interval is infinite, not a single bit, because there is an

infinite number of messages that the bidder can send to the center. The theory of information makes assumptions that are consistent with modern wired and wireless computer networks, in which messages can be encoded. Shoham and Tennenholtz' critical assumption that a continuous value may be communicated in one bit does not hold in modern multiagent systems.

Relying on Shoham and Tennenholtz' assumptions would lead a system designer to always choose their version of the Dutch auction to minimize the amount of communication from the bidder to the center. This paper shows that the expected amount of information communicated by an algorithm is highly dependent on the number of bidders and the distribution of bidders' private values. The Dutch auction is often not the algorithm that minimizes the expected amount of communication from the bidder to the center. Therefore, a system designer that relies on Shoham and Tennenholtz' assumption may incur unnecessary costs.

Much prior work [Shoham and Tennenholtz, 2001; Sunderam and Parkes, 2003, inter alia] has centered around measuring how much of a bidder's preferences are revealed by an algorithm instead of how much bandwidth is used by an algorithm. Therefore, a common assumption has been that coordination messages are free while revelation messages are costly. Under that assumption, it is desirable to select an algorithm with low revelation costs, even if it has high coordination costs. The results presented in Section 5 are for the sum of revelation and coordination costs and differ from such prior work for several reasons. However, situations in which only one direction of communication is important can be handled easily by the models described in this paper by ignoring the other direction in the analysis.

7 Conclusion and Future Work

In this paper, we presented three major contributions. First, we presented an argument for measuring expected information transmitted in a dialogue to determine the bandwidth need by multiagent algorithms. Second, we provided a method for measuring expected information using dialogue trees. Third, we showed that using that method to analyze five auctions leads to recommendations for multiagent system design that differ from recommendations made in previous work. The results of the analysis indicated that the correct choice of auction depends on the number of bidders and the size of the sample space from which bidders' values for the item are drawn. The Staged, Dutch, and Bisection auctions are each appropriate for different situations, and the Sealed-bid auction is best for very small problems. The guidelines presented in this paper could lead to real savings in communication bandwidth with no loss in decision quality.

In future work, we plan to use dialogue trees to analyze algorithms for more general team decision problems than single-item assignment and for more general algorithms than auctions. Auctions are commonly suggested for item or task assignment in multiagent systems because they are a convenient method for structuring communication between agents. We plan to compare other methods for allocating a single item, such as inter-agent exchange, to auctions. We assumed that agents were honest—small adjustments to the auctions rules instead allow us to build incentives into an auction directly. We plan to evaluate the communication costs incurred by auctions with built-in incentives and analyze the impact of those incentives on the correct choice of auction method.

8 Acknowledgments

This material is based on work supported by the National Science Foundation under Grant No. IIS-9978343.

References

[Cover and Thomas, 1991] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. Wiley-Interscience, 1991.

[Grigorieva *et al.*, 2002] Elena Grigorieva, P. Jean-Jacques Herings, Rudolf Müller, and Dries Vermeulen. The private value single item bisection auction. In *METEOR Research Memoranda*, number RM/02/035. Maastricht University, 2002.

[Hunsberger and Grosz, 2000] Luke Hunsberger and Barbara J. Grosz. A combinatorial auction for collaborative planning. In *Proceedings of the Fourth International Conference on Multi-Agent Systems (ICMAS-2000)*, 2000.

[Kushilevitz and Nisan, 1996] Eyal Kushilevitz and Noam Nisan. *Communication Complexity*. Cambridge University Press, 1996.

[Ortiz *et al.*, 2003] Charles L. Ortiz, Timothy W. Rauenbusch, and Eric Hsu. Dynamic resource-bounded negotiation in non-additive domains. In Victor Lesser, Charles L. Ortiz, Jr., and Milind Tambe, editors, *Distributed Sensor Networks: A Multiagent Perspective*. Kluwer Academic Publishers, 2003.

[Rasmussen, 1989] Eric Rasmussen. *Games and Information*. Blackwell, 1989.

[Rauenbusch, 2004] Timothy W. Rauenbusch. *Measuring Information Transmission for Team Decision Making*. PhD thesis, Harvard University, 2004.

[Shannon, 1948] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27, 1948.

[Shoham and Tennenholtz, 2001] Y. Shoham and M. Tennenholtz. Rational computation and the communication complexity of auctions. *Games and Economic Behavior*, 35(1–2):197–211, 2001.

[Sunderam and Parkes, 2003] Aditya V. Sunderam and David C. Parkes. Preference elicitation in proxied multiattribute auctions. In *Proceedings of the Fourth ACM Conference on Electronic Commerce*, 2003.

[Yao, 1979] Andrew Chi-Chih Yao. Some complexity questions related to distributive computing (preliminary report). In *Proceedings of the eleventh annual ACM symposium on Theory of computing*, pages 209–213, Atlanta, 1979.

Coordinating Agile Systems Through The Model-based Execution of Temporal Plans*

Thomas Léauté and **Brian Williams**
MIT Computer Science and Artificial Intelligence Laboratory (CSAIL)
Bldg. 32-G273, 77 Massachusetts Ave., Cambridge, MA 02139
{thleaute, williams}@mit.edu

Abstract

Agile autonomous systems are emerging, such as unmanned aerial vehicles (UAVs), that must robustly perform tightly coordinated time-critical missions; for example, military surveillance or search-and-rescue scenarios. In the space domain, execution of temporally flexible plans has provided an enabler for achieving the desired coordination and robustness. We address the challenge of extending plan execution to non-holonomic systems that are controlled indirectly through the setting of continuous state variables.

Our solution is a novel model-based executive that takes as input a temporally flexible *state plan*, specifying intended state evolutions, and dynamically generates an optimal control sequence. To achieve optimality and safety, the executive plans into the future, framing planning as a disjunctive programming problem. To achieve robustness to disturbances and tractability, planning is folded within a receding horizon, continuous planning framework. Key to performance is a problem reduction method based on constraint pruning. We benchmark performance through a suite of UAV scenarios using a hardware-in-the-loop testbed.

Introduction

Autonomous control of dynamic systems has application in a wide variety of fields, from managing a team of agile unmanned aerial vehicles (UAVs) for fire-fighting missions, to controlling a Mars life support system. The control of such systems is challenging for several reasons. First, they are non-holonomic systems, which means they are underactuated (not all state variables are directly controllable); second, their models involve continuous dynamics described by differential equations; third, controlling these systems usually requires tight synchronization; and fourth, the controller must be optimal and robust to disturbances.

An autonomous controller for agile systems must, therefore, provide three capabilities: 1) to handle tight coordination, the system should execute a temporal plan specifying time coordination constraints. 2) To deal with the underactuated nature of the system, it should elevate the interaction with the system under control (or *plant*) to the level at which the human operator is able to robustly program the plant in terms of desired state evolution, including state variables that are not directly controllable. 3) To deal with the non-holonomic dynamics of the plant, the intended state evolution must be specified in a temporally flexible manner, allowing robust control over the system.

Previous work in model-based programming introduced a *model-based executive*, called *Titan* (Williams 2003), that elevates the level of interaction between human operators and hidden-state, non-holonomic systems, by allowing the operator to specify the behavior to be executed in terms of intended plant state evolution, instead of specific command sequences. The executive uses models of the plant to map the desired state evolution to a sequence of commands driving the plant through the specified states. However, Titan focuses on reactive control of discrete-event systems, and does not handle temporally flexible constraints.

Work on dispatchable execution (Vidal & Ghallab 1996; Morris, Muscettola, & Tsamardinos 1998; Tsamardinos, Pollack, & Ramakrishnan 2003) provides a framework for robust scheduling and execution of temporally flexible plans. This framework uses methods based on distance graphs to both tighten time constraints in the plan, in order to guarantee dispatchability, and propagate occurrence of events during plan execution. However, this work was applied to discrete, directly controllable, loosely coupled systems, and, therefore, must be extended to non-holonomic plants.

Previous work on continuous planning and execution (Ambros-Ingerson & Steel 1988; Wilkins & Myer 1995; Chien *et al.* 2000) also provides methods to achieve robustness, by interleaving planning and execution, allowing on-the-fly replanning and adaptation to disturbances. These methods, inspired from model predictive control (*MPC*) (Propoi 1963; Richalet *et al.* 1976), involve planning and scheduling iteratively over short horizons, while revising the plan when necessary during execution. This work, however, needs to be extended to deal with temporally flexible plans and non-holonomic systems with continuous dynamics.

We propose a model-based executive that unifies the three previous approaches and enables coordinated control of agile systems, through model-based execution of *temporally flexible state plans*. Our approach is novel with respect to three aspects. First, we provide a general method for encod-

*This research is supported in part by The Boeing Company under contract MIT-BA-GTA-1, and by the Air Force Research Lab award under contract F33615-01-C-1850

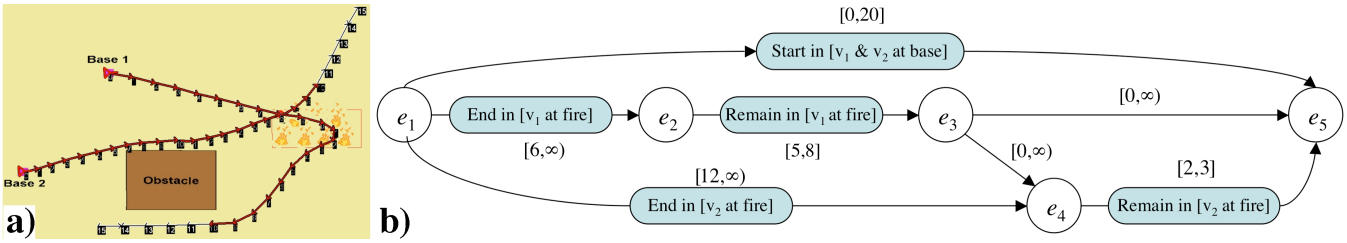


Figure 1: **a)** Map of the terrain for the fire-fighting example; **b)** Corresponding temporally flexible state plan.

ing both the temporal state plan and the dynamics of the system as a mixed discrete-continuous mathematical program. Solving this program provides time-optimal trajectories in the plant state space that satisfy the system dynamics and the state plan. Second, to achieve efficiency and robustness, we apply MPC for planning of control trajectories, in the context of continuous temporal plan execution for non-holonomic dynamical systems. MPC allows us to achieve tractability, by reasoning over a limited receding horizon. Third, in order to further reduce the complexity of the program and solve it in real time, we introduce pruning policies that enable us to ignore some of the constraints in the state plan outside the current planning horizon.

Problem Statement

Given a dynamic system (a *plant*) described by a *plant model*, and given a *temporally flexible state plan*, specifying the desired evolution of the plant state over time, the *continuous model-based execution* (*CMEx*) problem consists in designing a control sequence that produces a plant state evolution that is consistent with the state plan. In this section we present a formal definition of the CMEx problem.

Multiple-UAV Fire-fighting Example

This paragraph introduces the multiple-UAV fire-fighting example used in this paper. In this example, the plant consists of two fixed-wing UAVs, whose state variables are their 2-D Cartesian positions and velocities. The vehicles evolve in an environment (Fig. 1a) involving a reported fire that the team is assigned to extinguish. To do so, they must navigate around unsafe regions (e.g. obstacles) and drop water on the fire. They must also take pictures after the fire has been extinguished, in order to assess the damage. An English description for the mission's state plan is:

> *Vehicles v_1 and v_2 must start at their respective base stations. v_1 (a water tanker UAV) must reach the fire region and remain there for 5 to 8 time units, while it drops water over the fire. v_2 (a reconnaissance UAV) must reach the fire region after v_1 is done dropping water and must remain there for 2 to 3 time units, in order to take pictures of the damage. The overall plan execution must last no longer than 20 time units.*

Definition of a Plant Model

A *plant model* $\mathcal{M} = \langle \mathbf{s}, S, \mathbf{u}, \Omega, \mathcal{SE} \rangle$ consists of a vector $\mathbf{s}(t)$ of *state variables*, taking on values from the *state space* $S \subset \mathbb{R}^n$, a vector $\mathbf{u}(t)$ of *input variables*, taking on values from the *context* $\Omega \subset \mathbb{R}^m$, and a set $\mathcal{SE}$ of *state equations* over $\mathbf{u}$, $\mathbf{s}$ and its time derivatives, describing the plant behavior with time. S and Ω impose linear safety constraints on $\mathbf{s}$ and $\mathbf{u}$.

In our multiple-UAV example, $\mathbf{s}$ is the vector of 2-D coordinates of the UAV positions and velocities, and $\mathbf{u}$ is the acceleration coordinates. $\mathcal{SE}$ is the set of equations describing the kinematics of the UAVs. The unsafe regions in S correspond to obstacles and bounds on nominal velocities, and the unsafe regions in Ω to bounds on accelerations.

Definition of a Temporally Flexible State Plan

A *temporally flexible state plan* $P = \langle \mathcal{E}, \mathcal{C}, \mathcal{A} \rangle$ specifies a desired evolution of the plant state, and is defined by a set $\mathcal{E}$ of *events*, a set $\mathcal{C}$ of *coordination constraints*, imposing temporal constraints between events, and a set $\mathcal{A}$ of *activities*, imposing constraints on the plant state evolution. A coordination constraint $c = \langle e_1, e_2, \Delta T^{min}_{e_1 \to e_2}, \Delta T^{max}_{e_1 \to e_2} \rangle$ constrains the distance from event e_1 to event e_2 to be in $[\Delta T^{min}_{e_1 \to e_2}, \Delta T^{max}_{e_1 \to e_2}] \subset [0, +\infty]$. An *activity* $a = \langle e_1, e_2, c_S \rangle$ has an associated *start event* e_1 and an *end event* e_2. Given an assignment $T : \mathcal{E} \mapsto \mathbb{R}$ of times to all events in P (a *schedule*), c_S is a *state constraint* that can take on one of the following forms, where D_S, D_E, $D_\forall$ and $D_\exists$ are domains of S described by linear constraints on the state variables:

1. *Start in state region* D_S: $\mathbf{s}(T(e_1)) \in D_S$;
2. *End in state region* D_E: $\mathbf{s}(T(e_2)) \in D_E$;
3. *Remain in state region* $D_\forall$: $\forall t \in [T(e_1), T(e_2)], \mathbf{s}(t) \in D_\forall$;
4. *Go by state region* $D_\exists$: $\exists t \in [T(e_1), T(e_2)], \mathbf{s}(t) \in D_\exists$.

We illustrate a state plan diagrammatically by an acyclic directed graph in which events are represented by nodes, coordination constraints by arcs, labeled by their corresponding time bounds, and activities by arcs labeled with associated state constraints. The state plan for the the multiple-UAV fire-fighting mission example is shown in Fig. 1b.

Definition of the CMEx Problem

Schedule T for state plan P is *temporally consistent* if it satisfies all $c \in \mathcal{C}$. Given an activity $a = \langle e_1, e_2, c_S \rangle$ and a schedule T, a state sequence $\mathbf{S} = \langle \mathbf{s}_0 \dots \mathbf{s}_t \rangle$ *satisfies activity* a if it satisfies c_S. $\mathbf{S}$ then *satisfies state plan* P if there exists a temporally consistent schedule such that $\mathbf{S}$ satisfies

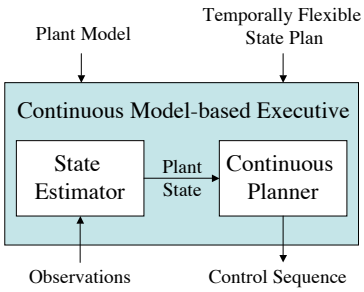


Figure 2: Continuous model-based executive architecture

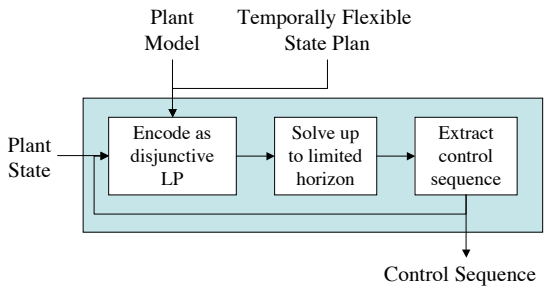


Figure 3: Receding horizon continuous planner

every activity in $\mathcal{A}$. Similarly, given a plant model $\mathcal{M}$ and initial state $\mathbf{s}_0$, a control sequence $\mathbf{U} = \langle \mathbf{u}_0 \dots \mathbf{u}_t \rangle$ *satisfies* P if it generates a state sequence that satisfies P. $\mathbf{U}$ is *optimal* if it satisfies P while minimizing an *objective function* $F(\mathbf{U}, \mathbf{S}, T)$. A common objective is to minimize the scheduled time $T(e_E)$ for the end event e_E of P.

Given an initial state $\mathbf{s}_0$, a plant model $\mathcal{M}$ and state plan P, the CMEx problem consists of generating, for each $t > 0$, a control action $\mathbf{u}_t$ from a control sequence $\langle \mathbf{u}_0 \dots \mathbf{u}_{t-1} \rangle$ and its corresponding state sequence $\langle \mathbf{s}_0 \dots \mathbf{s}_t \rangle$, such that $\langle \mathbf{u}_0 \dots \mathbf{u}_t \rangle$ is optimal. A corresponding *continuous model-based executive* consists of a *state estimator* and a *continuous planner* (Fig. 2). The continuous planner takes in a state plan, and generates optimal control sequences, based on the plant model, and state sequences provided by the state estimator. The estimator reasons on sensor observations and on the plant model in order to continuously track the state of the plant. Previous work on hybrid estimation (Hofbaur & Williams 2004) provides a framework for this state estimator; in this paper, we focus on presenting an algorithm for the continuous planner.

Overall Approach

Previous model-based executives, such as Titan, focus on reactively controlling discrete-event systems (Williams 2003). This approach is not applicable to temporal plan execution of systems with continuous dynamics; our continuous model-based executive uses a different approach that consists of planning into the future, in order to perform optimal, safe execution of temporal plans. However, solving the whole CMEx problem over an infinite horizon would present two major challenges. First, the problem is intractable in the case of long-duration missions. Second, it would require perfect knowledge of the state plan and the environment beforehand; this assumption does not always hold in real-life applications such as our fire-fighting scenario, in which the position of the fire might precisely be known only once the UAVs are close enough to the fire to localize it. Furthermore, the executive must be able to compensate possible drift due to approximations or errors in the plant model.

Receding Horizon CMEx

Model Predictive Control (*MPC*), also called *Receding Horizon Control*, is a method introduced in (Propoi 1963; Richalet *et al.* 1976) that tackles these two challenges in the context of low-level control of systems with continuous dynamics. MPC solves the control problem up to a limited *planning horizon*, and re-solves it when it reaches a shorter *execution horizon*. This method makes the problem tractable by restricting it to a small planning window; it also allows for on-line, robust adaptation to disturbances.

In this paper, we extend MPC to continuous model-based execution of temporal plans by introducing a *receding horizon continuous model-based executive*. We formally define receding horizon CMEx as follows. Given a state plan P, a plant model $\mathcal{M}$, and an initial state $\mathbf{s}(t_0)$, *single-stage, limited horizon CMEx* consists of generating an optimal control sequence $\langle \mathbf{u}_{t_0} \dots \mathbf{u}_{t_0+N_t} \rangle$ for P, where N_t is the planning horizon. The *receding horizon CMEx* problem consists of iteratively solving single-stage, limited horizon CMEx for successive initial states $\mathbf{s}(t_0 + i \cdot n_t)$ with $i = 0, 1, \dots$, where $n_t \leq N_t$ is the execution horizon. The architecture for our model-based executive is presented in Fig. 3.

Disjunctive Linear Programming Formulation

As introduced in Fig. 3, we solve each single-stage limited horizon CMEx problem by encoding it as a *disjunctive linear program* (*DLP*) (Balas 1979). A DLP is an optimization problem with respect to a linear cost function over decision variables, subject to constraints that can be written as logical combinations of linear inequalities (Eq. (1)). In this paper, we solve DLPs by reformulating them as Mixed-Integer Linear Programs. Other current work addresses directly solving DLPs (Krishnan 2004).

$$\begin{array}{c} \min f(\mathbf{x}) \\ subject\ to: \bigwedge_i \bigvee_j g_{i,j}(\mathbf{x}) \leq 0 \end{array} \quad (1)$$

Any arbitrary propositional logic formula whose propositions are linear inequalities is reducible to a DLP. Hence, in this paper, we expose formulae in propositional form.

Single-stage Limited Horizon CMEx as a DLP

We now present how we encode single-stage limited horizon CMEx as a DLP. Our innovation is the encoding of the state plan as a goal specification for the plant.

State Plan Encodings

In the following paragraphs, we present the encodings for the state plan. A range of objective functions are possible, the most common being to minimize completion time. To encode this, for every event $e \in \mathcal{E}$, we add to the DLP cost function, the time $T(e)$ at which e is scheduled.

Temporal constraints between events: Eq. (2) encodes a temporal constraint between two events e_S and e_E. For example, in Fig. 1b, events e_1 and e_5 must be distant from each other by a least 0 and at most 20 time units.

$$\Delta T^{\min}_{e_S \to e_E} \leq T(e_E) - T(e_S) \leq \Delta T^{\max}_{e_S \to e_E} \tag{2}$$

State activity constraints: Activities are of the following types: *start in*, *end in*, *remain in* and *go by*. *Start in* and *go by* are derivable easily from the primitives *remain in* and *end in*. We present the encodings for these two primitives below. In each case, we assume that the domains D_E and $D_\forall$ are unions of polyhedra (Eq. (6)), so that $\mathbf{s}_t \in D_E$ and $\mathbf{s}_t \in D_\forall$ can be expressed as DLP constraints similar to (Eq. (7)).

***Remain in* activity:** Eq. (3) presents the encoding for the *remain in state* $D_\forall$ activity between events e_S and e_E. This imposes $\mathbf{s} \in D_\forall$ for all time steps between $T(e_S)$ and $T(e_E)$. Our example imposes the constraint "Remain in state [v_1 in fire region]", which means that v_1 must be in the fire region between e_2 and e_3 while it is dropping water. $T_0 \in \mathbb{R}$ denotes the initial time instant of index $t = 0$.

$$\bigwedge_{t=0\ldots N_t} \left\{ \begin{array}{rc} & T(e_S) \leq T_0 + t \cdot \Delta T \\ \wedge & T(e_E) \geq T_0 + t \cdot \Delta T \end{array} \right\} \Rightarrow \mathbf{s}_t \in D_\forall \tag{3}$$

***End in* activity:** Consider a *end in* activity imposing $\mathbf{s} \in D_\forall$ at the time $T(e_E)$ when event e_E is scheduled. An example in our fire-fighting scenario is the "End in state [v_2 in fire region]" constraint imposing v_2 to be in the fire region at event e_4. The general encoding is presented in Eq. (4), which translates to the fact that, either there exists a time instant of index t in the planning window that is ΔT-close to $T(e_E)$ and for which $\mathbf{s}_t \in D_\forall$, or event e_E must be scheduled outside of the current planning window .

$$\begin{array}{c} \bigvee_{t=0\ldots N_t} \\ \vee \\ \vee \end{array} \begin{array}{c} \left\{ \begin{array}{rc} & T(e_E) \geq T_0 + (t - \frac{1}{2})\Delta T \\ \wedge & T(e_E) \leq T_0 + (t + \frac{1}{2})\Delta T \\ \wedge & \mathbf{s}_t \in D_\forall \end{array} \right\} \\ T(e_E) \leq T_0 - \frac{\Delta T}{2} \\ T(e_E) \geq T_0 + (N_t + \frac{1}{2})\Delta T \end{array} \tag{4}$$

Guidance heuristic for *End in* activities: During execution of an "End in state region D_E" activity, the end event may be scheduled beyond the current horizon. In this case, the model-based executive constructs a *heuristic*, in order to guide the trajectory towards D_E. For this purpose, an estimate of the "distance" to D_E from the end of the current partial state trajectory is added to the DLP cost function, so that the partial trajectory ends as "close" to D_E as possible. In the case of the multiple-UAV fire-fighting scenario, this "distance" is an estimate of the time needed to go from the end of the current trajectory to the goal region D_E.

The heuristic is formally defined as a function $h_{D_E} : \mathcal{S} \mapsto \mathbb{R}$, where $\mathcal{S} = \{S_i \subset S\}$ is a finite partition of S such that $D_E = S_{i_{D_E}} \in \mathcal{S}$. Given $S_i \in \mathcal{S}$, $h_{D_E}(S_i)$ is an estimate of the cost to go from S_i to D_E. In the fire-fighting example, $\mathcal{S}$ is a grid map in which each grid cell S_i corresponds to a hypercube centered on a state vector $\mathbf{s}_i$, and $h_{D_E}(S_i)$ is an estimate of the time necessary to go from state $\mathbf{s}_i$ to the goal state $\mathbf{s}_{i_{D_E}}$. Similar to (Bellingham, Richards, & How 2002), we compute h_{D_E} by constructing a visibility graph based on the unsafe regions of the $\langle x, y \rangle$ state space, and by computing, for every i, the cost to go from $\langle x_i, y_i \rangle \in S_i$ to the goal state $\langle x_{i_{D_E}}, y_{i_{D_E}} \rangle \in D_E$.

Eq. (5) presents the constraint, for a given "End in state region D_E" activity a, starting at event e_S and ending at event e_E. This encodes the fact that, if a is scheduled to start within the execution horizon but end beyond, then the executive must choose a region $S_i \in \mathcal{S}$ so that the partial state trajectory ends in S_i, and the value h of the heuristic at S_i is minimized (by adding h to the DLP cost function).

$$\begin{array}{l} \left\{ \begin{array}{rc} & T(e_S) < T_0 + n_t \cdot \Delta T \\ \wedge & T(e_E) \geq T_0 + n_t \cdot \Delta T \end{array} \right\} \\ \Rightarrow \bigvee_{S_i \in \mathcal{S}} \left\{ \begin{array}{rc} & h = h_{D_E}(S_i) \\ \wedge & \mathbf{s}_{n_t} \in S_i \end{array} \right\} \end{array} \tag{5}$$

Eq. (5) can be simplified by reducing $\mathcal{S}$ to a subset $\tilde{\mathcal{S}} \subset \mathcal{S}$ that excludes all the S_i unreachable within the horizon. For instance, in the multiple-UAV example, the maximum velocity constraints allow us to ignore the S_i that are not reachable by the UAVs within the execution horizon. We present in a later section how $\mathcal{M}$ allows us to determine, in the general case, when a region of the state space is unreachable.

Plant Model Encodings

Recall that a plant model $\mathcal{M}$ consists of a state space S and a context Ω imposing linear constraints on the variables, and a set of state equations $\mathcal{SE}$. We represent unsafe regions in S and Ω by unions of polyhedra, where a polyhedron $\mathcal{P}_S$ of S is defined in Eq. (6). Polyhedra of Ω are defined similarly.

$$\mathcal{P}_S = \left\{ \mathbf{s} \in \mathbb{R}^n \mid \mathbf{a}_i^T \mathbf{s} \leq b_i \, , i = 1 \ldots n_{\mathcal{P}_S} \right\} \tag{6}$$

The corresponding encoding (Eq. (7)) constrains $\mathbf{s}_t$ to be outside of $\mathcal{P}_S$ for all t. In the UAV example, this corresponds to the constraint encoding obstacle collision avoidance.

$$\bigwedge_{t=1\ldots N_t} \bigvee_{i=1\ldots n_{\mathcal{P}_S}} \mathbf{a}_i^T \mathbf{s}_t \geq b_i \tag{7}$$

The state equations in $\mathcal{SE}$ are given in DLP form in Eq. (8), with the time increment ΔT assumed small with respect to the plant dynamics. In our fixed-wing UAV example, we use the zero-order hold time discretization model from (Kuwata 2003).

$$\mathbf{s}_{t+1} = \mathbf{A}\mathbf{s}_t + \mathbf{B}\mathbf{u}_t \tag{8}$$

In this section, we presented how we designed optimal control sequences that satisfy the state plan, by formulating the problem as a DLP using an MPC framework. We now present how we simplify the DLP to solve it in real time.

Constraint Pruning Policies

Recall that our executive solves the CMEx problem by encoding it as a DLP and iteratively solving it over small planning windows. The ability of the executive to look into the future is limited by the number of variables and constraints in the DLP. In the next section, we introduce novel *pruning policies* that dramatically reduce the number of constraints.

Plant Model Constraint Pruning

Recall that $\mathcal{M}$ defines unsafe regions in S using polyhedra (Eq. (6)). The DLP constraint for a polyhedron $\mathcal{P}_S$ (Eq. (7)) can be pruned if $\mathcal{P}_S$ is unreachable from the current plant state $\mathbf{s}_0$, within the horizon N_t. That is, if the region R of all states reachable from $\mathbf{s}_0$ within N_t is disjunct from $\mathcal{P}_S$. R is formally defined in Eq. (9) and (10), with $R_0 = \{\mathbf{s}_0\}$.

$$R_{t+1} = \left\{ \mathbf{s}_{t+1} \middle| \begin{array}{c} \forall t = 0 \dots N_t - 1, \\ \mathbf{s}_{t+1} = \mathbf{A}\mathbf{s}_t + \mathbf{B}\mathbf{u}_t, \\ \mathbf{s}_t \in R_t, \mathbf{u}_t \in \Omega \end{array} \right\} \quad (9)$$

$$R = \bigcup_{t=0 \dots N_t} R_t \quad (10)$$

Techniques have been developed in order to compute R (Tiwari 2003). In the UAV example, for a given vehicle, we use a simpler, sound but incomplete method, in which we approximate R by a circle centered on the vehicle and of radius $N_t \cdot \Delta T \cdot v^{\max}$, where $v^{\max}$ is the vehicle's maximum velocity.

State Plan Constraint Pruning

State plan constraints can be either temporal constraints between events, *remain in* constraints, *end in* constraints, or heuristic guidance constraints for *end in* activities. For each type, we now show that the problem of finding a policy is equivalent to that of foreseeing if an event could possibly be scheduled within the current horizon. This is solved by computing bounds $\langle T_e^{\min}, T_e^{\max} \rangle$ on $T(e)$, for every $e \in \mathcal{E}$. Given the execution times of past events, these bounds are computed from the bounds $\langle \Delta T^{\min}_{\langle e,e' \rangle}, \Delta T^{\max}_{\langle e,e' \rangle} \rangle$ on the distance between any pair of events $\langle e, e' \rangle$, obtained using the method in (Dechter, Meiri, & Pearl 1991). This involves running an all-pairs shortest path algorithm on the distance graph corresponding to P, which can be done offline.

Temporal constraint pruning: A temporal constraint between a pair of events $\langle e_S, e_E \rangle$ can be pruned if the time bounds on either event guarantee that the event will be scheduled outside of the current planning window (Alg. 1).

However, pruning some of the temporal constraints specified in the state plan can have two bad consequences. First, implicit temporal constraints between two events that can be scheduled within the current planning window might no longer be enforced. Implicit temporal constraints are constraints that do not appear explicitly in the state plan, but rather result from several explicit temporal constraints. Second, the schedule might violate temporal constraints between events that remain to be scheduled, and events that have already been executed.

Alg. 1 Pruning policy for the temporal constraint between events e_S and e_E

1: **if** $T_{e_S}^{\max} < T_0$ **then**
2: prune $\{e_S$ has already been executed$\}$
3: **else if** $T_{e_S}^{\min} > T_0 + N_t \cdot \Delta T$ **then**
4: prune$\{e_S$ is out of reach within the current horizon$\}$
5: **else if** $T_{e_E}^{\max} < T_0$ **then**
6: prune $\{e_E$ has already been executed$\}$
7: **else if** $T_{e_E}^{\min} > T_0 + N_t \cdot \Delta T$ **then**
8: prune$\{e_E$ is out of reach within the current horizon$\}$
9: **end if**

Alg. 2 Pruning policy for the absolute temporal constraint on an event e

1: **if** $T_e^{\max} < T_0$ **then**
2: prune $\{e$ has already been executed$\}$
3: **else if** $T_e^{\min} > T_0 + N_t \cdot \Delta T$ **then**
4: prune$\{e$ is out of reach within the current horizon$\}$
5: POSTPONE(e)
6: **end if**

To tackle the first issue aforementioned, rather than encoding only the temporal constraints that are mentioned in the state plan, we encode the temporal constraints between any pair of events $\langle e, e' \rangle$, using the temporal bounds $\langle \Delta T^{\min}_{\langle e,e' \rangle}, \Delta T^{\max}_{\langle e,e' \rangle} \rangle$ computed by the method in (Dechter, Meiri, & Pearl 1991). This way, no implicit temporal constraint is ignored, because all temporal constraints between events are explicitly encoded.

To address the second issue, we also encode the absolute temporal constraints on every event e: $T_e^{\min} \leq T(e) \leq T_e^{\max}$. The pruning policy for those constraints is presented in Alg. 2. The constraint can be pruned if e is guaranteed to be scheduled in the past (i.e. it has already been executed, line 1). It can also be pruned if e is guaranteed to be scheduled beyond the current planning horizon (line 3). In that case, e must be explicitly postponed (Alg. 3, lines 1 & 2) to make sure it will not be scheduled before T_0 at the next iteration (which would then correspond to scheduling e in the past before it has been executed). The change in $T_e^{\min}$ is then propagated to the other events (Alg. 3, line 3).

Alg. 3 POSTPONE(e) routine to postpone an event e

1: $T_e^{\min} \leftarrow T_0 + N_t \cdot \Delta T$
2: add $T(e) \geq T_0 + N_t \cdot \Delta T$ to the DLP
3: **for all** events e' **do** $\{$propagate to other events$\}$
4: $T_{e'}^{\min} \leftarrow \max(T_{e'}^{\min}, T_e^{\min} + \Delta T^{\min}_{\langle e,e' \rangle})$
5: **end for**

Alg. 4 Pruning policy for a "Remain in state region $D_\forall$" activity starting at event e_S and ending at event e_E

1: **if** $T^{\max}_{e_E} < T_0$ **then**
2: prune {activity is completed}
3: **else if** $T^{\max}_{e_S} < T_0$ **then**
4: do not prune{activity is being executed}
5: **else if** $T^{\min}_{e_S} > T_0 + N_t \cdot \Delta T$ **then**
6: prune {activity will start beyond N_t}
7: **else if** $T^{\max}_{e_S} < T_0 + N_t \cdot \Delta T$ **then**
8: do not prune {activity will start within N_t}
9: **else if** $R \cap D_\forall = \emptyset$ **then**
10: prune; POSTPONE(e_S)
11: **end if**

Alg. 5 Pruning policy for a "End in state region D_E" activity ending at event e_E

1: **if** $T^{\max}_{e_E} < T_0$ **then**
2: prune {e_E has already occurred}
3: **else if** $T^{\max}_{e_E} \leq T_0 + N_t \cdot \Delta T$ **then**
4: do not prune {e_E will be scheduled within N_t}
5: **else if** $T^{\min}_{e_E} > T_0 + N_t \cdot \Delta T$ **then**
6: prune {e_E will be scheduled beyond N_t}
7: **else if** $R \cap D_E = \emptyset$ **then**
8: prune; POSTPONE(e_E)
9: **end if**

***Remain in* constraint pruning (Alg. 4):** Consider the constraint c_S on a "Remain in state region $D_\forall$" activity a, between events e_S and e_E (Eq. (3)). If e_E is guaranteed to be scheduled in the past (i.e. it has already occurred, line 1), then a has been completed and c_S can be pruned. Otherwise, if e_S has already occurred (line 3), then a is being executed and c_S must not be pruned. Otherwise, if a is guaranteed to start beyond the planning horizon (line 5), then c_S can be pruned. Conversely, if a is guaranteed to start within the planning horizon (line 7), then c_S must not be pruned.

Otherwise, the time bounds on $T(e_S)$ and $T(e_E)$ provide no guarantee, but we can still use $\mathcal{M}$ to try to prune the constraint: if $\mathcal{M}$ guarantees that $D_\forall$ is unreachable within the planning horizon, then c_S can be pruned (line 9; refer to Eq. (9) & (10) for the definition of R). Similarly to Alg. 2, e_S must then be explicitly postponed.

***End in* constraint pruning (Alg. 5):** Consider a constraint c_S on an "End in state region D_E" activity ending at event e_E (Eq. (4)). If e_E is guaranteed to be scheduled in the past (i.e., it has already occurred, line 1), then c_S can be pruned. Otherwise, if the value of $T^{\max}_{e_E}$ guarantees that e_E will be scheduled within the planning horizon (line 3), then c_S must not be pruned. Conversely, it can be pruned if $T^{\min}_{e_E}$ guarantees that e_E will be scheduled beyond the planning horizon (line 5). Finally, c_S can also be pruned if the plant model guarantees that D_E is unreachable within the planning horizon from the current plant state (line 7). Similarly to Alg. 2, e_E must then be explicitly postponed.

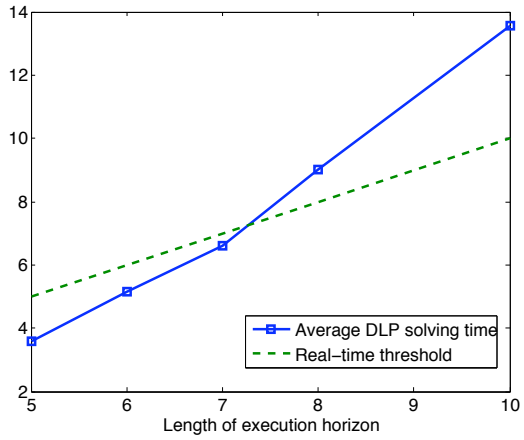


Figure 4: Performance of the model-based executive.

Guidance constraint pruning: The heuristic guidance constraint for an *end in* activity a between events e_S and e_E (Eq. (5)) can be pruned if a is guaranteed either to end within the execution horizon ($T^{\max}_{e_E} < T_0 + n_t \cdot \Delta T$) or to start beyond ($T^{\min}_{e_S} > T_0 + n_t \cdot \Delta T$).

In the next section, we show that our model-based executive is able to design optimal control sequences in real time.

Results and Discussion

The model-based executive has been implemented in C++, using Ilog CPLEX to solve the DLPs. It has been demonstrated on a range of fire-fighting scenarios on a hardware-in-the-loop testbed, comprised of CloudCap autopilots controlling a set of fixed-wing UAVs. This offers a precise assessment of real-time performance on UAVs. Fig. 1a was obtained from a snapshot of the operator interface, and illustrates the trajectories corresponding to our example.

Fig. 4 presents an analysis of the performance of the executive on a more complex example, comprised of two vehicles, two obstacles, and 26 activities, for a total execution time of about 1300s. These results were obtained on a 1.7GHz computer with 512MB RAM, by averaging over the whole plan execution, and over 5 different runs with random initial conditions. At each iteration, the computation was cut short if and when it passed 200s.

The x axis corresponds to the length of the execution horizon, $n_t \cdot \Delta T$, in seconds. For these results, we maintained a planning buffer of $N_t \cdot \Delta T - n_t \cdot \Delta T = 10s$ (where $N_t \cdot \Delta T$ is the length of the planning horizon). The full line represents the average time in seconds required by CPLEX to solve the DLP at each iteration, while the dotted line is the line $y = x$, corresponding to the real-time threshold.

Fig. 4 shows that, below the value $x \simeq 7.3s$, the model-based executive is able to compute optimal control sequences in real time (the average DLP solving time is below the length of the execution horizon). For longer planning horizons corresponding to values of x above $7.3s$, CPLEX is

unable to find optimal solutions to the DLPs before the executive is required to replan (average solving time greater than the execution horizon). Note that in that case, since CPLEX runs as an anytime algorithm, we can still interrupt it and use the best solution found so far to generate sub-optimal control sequences.

Note also that the number of variables in the DLP is linear in the length of the planning horizon; therefore, the complexity of the DLP is worst-case exponential in the length of the horizon. In Fig. 4, however, the relationship appears to be linear. This can be explained by the fact that the DLP is very sparse, since no constraint in the corresponding MILP involves more than three or four variables.

Future work includes carrying out more experiments in order to benchmark the performance of the pruning policies presented in this paper.

Conclusion

In this paper, we have presented a continuous model-based executive that is able to robustly execute temporal plans for agile, non-holonomic systems with continuous dynamics. In order to deal with the under-actuated nature of the plant and to provide robustness, the model-based executive reasons on temporally flexible state plans, specifying the intended plant state evolution. The use of pruning policies enables the executive to design optimal control sequences in real time, which was demonstrated on a hardware-in-the-loop testbed in the context of multiple-UAV fire-fighting scenarios. Our approach is broadly applicable to other dynamic systems, such as chemical plants or Mars life support systems.

References

Ambros-Ingerson, J. A., and Steel, S. 1988. Integrating planning, execution and monitoring. In *Proc. AAAI*.

Balas, E. 1979. Disjunctive programming. *Annals of Discrete Mathematics* 5:3–51.

Bellingham, J.; Richards, A.; and How, J. 2002. Receding horizon control of autonomous aerial vehicles. In *ACC*.

Chien, S.; Knight, R.; Stechert, A.; Sherwood, R.; and Rabideau, G. 2000. Using iterative repair to improve responsiveness of planning and scheduling. In *Proc. AIPS*.

Dechter, R.; Meiri, I.; and Pearl, J. 1991. Temporal constraint networks. *Artificial Intelligence Journal*.

Hofbaur, M. W., and Williams, B. C. 2004. Hybrid estimation of complex systems. *IEEE SMC - Part B: Cybernetics*.

Krishnan, R. 2004. Solving hybrid decision-control problems through conflict-directed branch and bound. Master's thesis, MIT.

Kuwata, Y. 2003. Real-time trajectory design for unmanned aerial vehicles using receding horizon control. Master's thesis, MIT.

Morris, P.; Muscettola, N.; and Tsamardinos, I. 1998. Reformulating temporal plans for efficient execution. In *Proc. KRR*, 444–452.

Propoi, A. 1963. Use of linear programming methods for synthesizing sampled-data automatic systems. *Automation and Remote Control* 24(7):837–844.

Richalet, J.; Rault, A.; Testud, J.; and Papon, J. 1976. Algorithmic control of industrial processes. In *IFAC Sym. Id. & Syst. Param. Est.*, 1119–1167.

Tiwari, A. 2003. Approximate reachability for linear systems. In *Proceedings of HSCC*, 514–525.

Tsamardinos, I.; Pollack, M. E.; and Ramakrishnan, S. 2003. Assessing the probability of legal execution of plans with temporal uncertainty. In *Proc. ICAPS*.

Vidal, T., and Ghallab, M. 1996. Dealing with uncertain durations in temporal constraint networks dedicated to planning. In *Proc. ECAI*, 48–52.

Wilkins, D. E., and Myer, K. L. 1995. A common knowledge representation for plan generation and reactive execution. *Journal of Logic and Computation* 5(6):731–761.

Williams, B. C. 2003. Model-based programming of intelligent embedded systems and robotic space explorers. In *Proc. IEEE: Modeling and Design of Embedded Software*.

Execution Monitoring and Replanning with Incremental and Collaborative Scheduling

David E. Wilkins[1], Stephen F. Smith[2], Laurence A. Kramer[2], Thomas J. Lee[1] and Timothy W. Rauenbusch[1]

[1]Artificial Intelligence Center
SRI International
Menlo Park, CA 94025
wilkins,tomlee,rauenbusch@ai.sri.com

[2]The Robotics Institute
Carnegie Mellon University
Pittsburgh, PA 15213
sfs,lkramer@cs.cmu.edu

Abstract

We describe the Flight Manager Assistant (FMA), a prototype system, designed to support real-time management of airlift operations at the USAF Air Mobility Command (AMC). In current practice, AMC flight managers are assigned to manage individual air missions. They tend to be overburdened with associated data monitoring and constraint checking, and generally react to detected problems in a local, myopic fashion. Consequently, decisions taken for one mission can often have deleterious effects on others. FMA combines two key capabilities for overcoming these problems: (1) intelligent monitoring of incoming information (for example, weather, airport operations, aircraft status) and recognizing those situations that require corrective action, and (2) dynamic rescheduling of missions in response to detected problems, both to understand the global implications of changed circumstances and to determine appropriate rescheduling actions. FMA builds on two of our existing technologies: an execution-monitoring framework previously applied to small-unit operations and control of robots, and a dynamic scheduling tool that is transitioning into operational use in AMC's Tanker/Airlift Control Center. FMA's dynamic mediation module provides for collaborative mission management by different planning and execution offices by structuring communication for decision making.

Introduction and Problem Statement

Management of flight operations at the United States Air Force Air Mobility Command (AMC) is a challenging problem. AMC typically flies several thousand missions worldwide on a weekly basis (more in a crisis situation), involving several hundreds of aircraft and comparable numbers of aircrews. The execution of any given mission requires attention to a broad range of constraints relating to the mission's requirements (e.g., delivery dates, cargo type and weight), resource availability (e.g., aircraft, aircrews, airports, diplomatic clearances), and usage constraints (e.g., crew duty day restrictions and scheduled return dates, aircraft speed, range, and capacity, airspace restrictions). Although missions are planned and globally scheduled to satisfy such constraints, the dynamics of execution regularly forces changes. Aircraft break down, airports become unavailable due to weather, missions become delayed due to diplomatic clearance problems, and so on, and all such events can warrant reassessment of previous allocation decisions. In such execution-driven rescheduling contexts, it is important to weigh potential recovery options against their prospective impact on future operations, and to take actions that continue to make the most effective global use of AMC assets.

In current practice, management of flight operations at AMC is a stovepiped process, where planning and execution are treated as sequential steps and information flows in one direction (from planning to execution). New mission requirements flow into AMC's planning offices on a continuous basis, and as they do aircraft and aircrews are incrementally allocated to support new missions in accordance with associated priorities and as resource availability allows. When a mission gets to within 24 hours of execution, it is "pushed" from the planning side of AMC to the execution office, and becomes the responsibility of an individual flight manager. AMC flight managers take responsibility for checking to ensure that all mission constraints remain satisfied before and during execution, and as problems are detected, they diagnose and revise mission plans to facilitate mission continuation and/or recovery. Unfortunately, AMC flight managers are not well supported in this execution management task. Some alerting tools do exist for signaling certain kinds of problems, but there is generally no ability to differentiate routine checks from exceptional events (i.e., everything shows up red), and no ability to detect more complex, compound conditions. Flight managers are typically overburdened by the data monitoring and constraint checking activities that are required to ensure the continuing viability of executing missions. Furthermore, when problematic situations are detected, flight managers have no visibility of the larger AMC operating picture, and must take recovery actions without regard to potential interactions with other missions. As a result, execution management often proceeds in fire-fighting mode, where putting out one fire ignites the next one.

For the past several years, we have been engaged in the development of technologies that we believe can provide a basis for more effective flight management. At Carnegie Mellon University (CMU) we have been developing the

AMC Allocator, a dynamic scheduling tool for day-to-day management of airlift and tanker schedules [Kramer & Smith 2002, Smith et al. 2004]. The AMC Allocator provides a range of capabilities for incrementally revising schedules to accommodate new or changed requirements, with continued emphasis on efficient resource utilization. It is currently transitioning into use as a "planning" tool in the Tanker/Airlift Control center at AMC. At SRI, we have been developing the Small Unit Operations Execution Assistant (SUO-EA), which monitors large volumes of situational data and gets urgent, plan-aware alerts to the right users [Wilkins et al. 2003]. SUO-EA has been successfully demonstrated in both the DARPA SUO program and ONR UCAV program. Also at SRI, we have developed technologies for incremental negotiation and coalition formation technology within the DARPA Autonomous Negotiating Teams program and the ONR UCAV program [Ortiz et al. 2003]. Finally, SRI's Open Agent Architecture (OAA) [Cheyer & Martin 2001] provides a robust integration infrastructure that has been used in dozens of programs and applications.

In this paper, we describe the Flight Manager Assistant (FMA), a system that integrates the above set of technology components to provide a flexible, mixed-initiative tool for real-time flight management. Through a coupling of execution monitoring capabilities with a global dynamic scheduler, the FMA is designed to promote a more integrated, and hence more informed, basis for detecting and responding to exceptional execution events. The FMA actively monitors data information sources for expectations it derives from the current schedule, recognizes deviations immediately, and applies policies for responding to deviations. Responses to significant deviations may alert the user to take control. Other options might include automated responses (when permitted by policy), or invoking the scheduler to explore alternative rescheduling options. By integrating status update information with the current schedule, the FMA indicates the important consequences of detected events on current and future operations. Through generation and comparison of alternative schedule repair options (either through interaction with the user or automatically), the FMA supports determination of globally coherent recovery actions while also promoting schedule changes that minimize disruption to other missions whenever possible. A given schedule repair process may also initiate and assist a collaboration between the user responsible for execution and the users who planned the missions. Finally, the FMA can provide automated support for implementing the human-selected response. The FMA continuously reacts to new information while interspersing its proactive pursuit of response procedures.

The broad goal of the FMA project has been to develop technology that enables increased organizational responsiveness and effectiveness in managing the dynamics of mission operations. In our view, there are two key factors to realizing this goal:

- *Increased automation.* Ubiquitous computers, data sources, and reliable, high-bandwidth communication networks are providing too much information for humans to monitor. In our vision, flight managers will rely on an automated execution aid to monitor the large (and ever increasing) volume of incoming information. By understanding the plan and situation, such an execution aid will consider the outputs of multiple monitoring techniques and tools, and then judge when the user should be alerted. Good judgment avoids overalerting. There may be many exceptions noted in the current plan by various AMC monitoring tools – the FMA recognizes which are most important, focuses the human on those, and assists with developing responses.
- *Closing the loop between planning and execution.* The ability to effectively respond to important alerts requires access to the global state of current and planned future operations, and to the rationale that underlies current mission plans/schedules. In our vision, flight managers will utilize dynamic scheduling tools to understand the consequences of detected events, to generate alternative reactions and evaluate the impact of each, and to provide a basis for negotiating mission requirements—the FMA provides these sorts of capabilities and enables a flight manager to apply a more global perspective in determining how to respond. The FMA also alerts originating planners to problems with their missions and provides support for them to contribute information relevant to execution decisions and achieve globally beneficial changes to individual mission plans.

The current FMA prototype is composed of two principal components: a Flight Manager Executive (built from SRI's SUO-EA system) and a Dynamic Scheduler (derived from CMU's AMC Allocator system). We have demonstrated this prototype on a series of execution management vignettes, using actual (full scale) AMC schedules pulled from AMC's Consolidated Air Mobility Planning System (CAMPS), and representative (but scripted) execution data streams. A third Dynamic Mediation component (based on SRI's incremental negotiation techniques) has undergone preliminary proof-of-concept testing.

In the sections below we describe these components in more detail, and give an indication of the application's status and potential for transition.

FMA Architecture

The FMA architecture features actors. There is an actor for each participant in the decision-making process. The FMA is configurable for arbitrary sets of decision makers. A

typical configuration includes at least one actor for the Execution Office and for each planning office (e.g., SAAM, Channel). Figure 1 depicts the various actors in a common configuration of our Flight Manager Assistant. We designed the software architecture for the various SRI and CMU components, and decided to use OAA to communicate between the various software agents in our architecture. Our system, the Flight Manager Assistant, is composed of four software modules:

- GUI
- Executive (Exception Handler)
- Dynamic Scheduler (DS)
- Dynamic Mediator (DM)

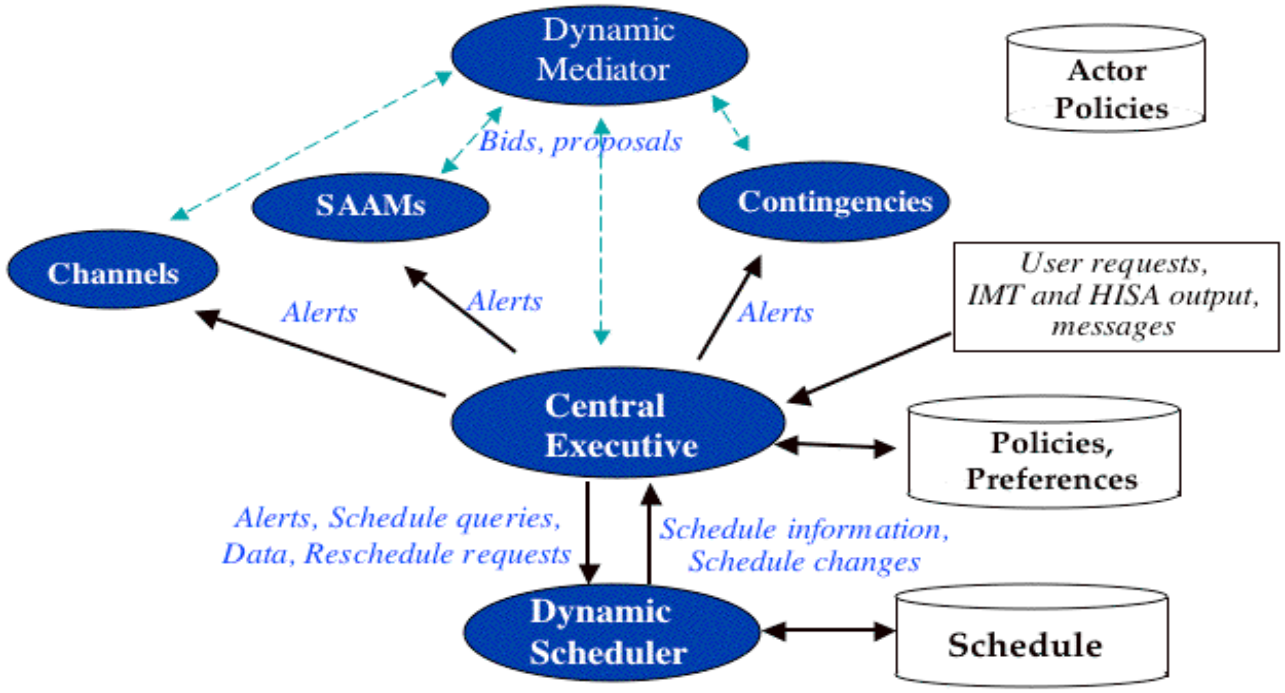


Figure 1: FMA Architecture. Arrows represent message and information flow; every agent communicates with the Actor Policies (arrows omitted). FMA monitors the output of HISA and IMT (AMC software tools which report MOG exceptions and execution-time exceptions respectively).

The DS is an FMA actor. Each other actor is an instantiation of the Executive, with its own GUI and value-of-information (VOI) functions that determine the alerts received and their priority.

The inputs the FMA monitors come from various AMC tools and messages from other actors and external agents. For example, one tool detects and reports maximum on ground (MOG) conflicts at airbases.

Executive. The key problem for the Executive is that algorithms that alert on constraint violations and threats in a straightforward manner inundate the user in dynamic domains. Unwanted alerts are a problem in many domains, from medicine to transportation to battle command. An execution aid that gives alerts every few seconds will quickly be discarded by the user in stressful situations (if not immediately). To be useful, an execution aid must produce high-value, user-appropriate alerts. Alerts and their presentation may also have to be adjusted to the situation, including the user's cognitive state (or the computational state of a software agent). For example, in high-stress situations, tolerances could be increased or certain types of alerts might be ignored or postponed.

Our approach is grounded in the concept of determining the value of an alert. First, the system must estimate the value of new information to the user. We use the term value of information (VOI) to refer to the pragmatic import the information has relative to its receiver. We assume that the practical value of information derives from its usefulness in making informed decisions. However, alerting the user to all valuable information could have a negative impact in certain situations, such as when the alert distracts the user from more important tasks, or when too many alerts overwhelm the user. We therefore introduce the concept of value of an alert (VOA), which is the pragmatic import (for making informed decisions) of taking an action to focus the user's attention on a piece of information. VOA takes VOI into account but weighs it against the costs and benefits of

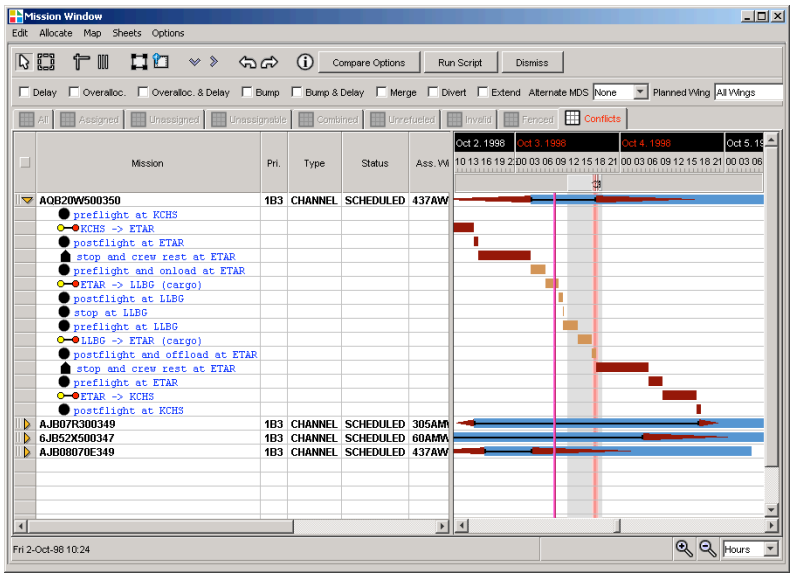


Figure 2: Screenshot of the Dynamic Scheduler

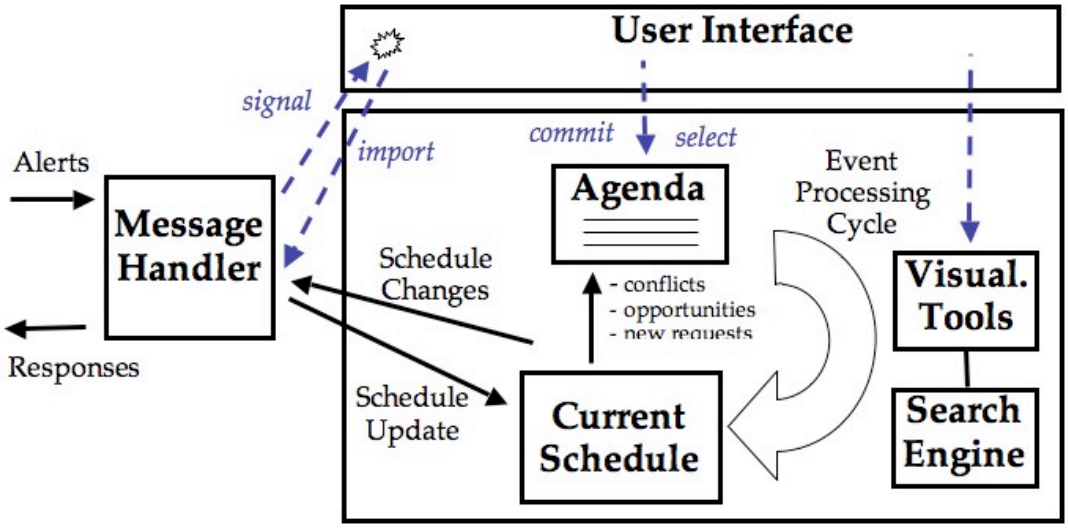


Figure 3: Internal architecture of the FMA Dynamic Scheduler

interrupting the user. If the user is busy doing something significantly more important, then issuing an alert might not be valuable, even when VOI is high.

Our monitoring framework integrates many domain-specific and task-specific monitoring techniques and then uses the concept of value of an alert to avoid operator overload. We have used this framework to implement Execution Assistants (EAs) in three different dynamic, data-rich, real-world domains to assist a human in monitoring team behavior. One domain (Army small unit operations) has hundreds of mobile, geographically distributed agents, a combination of humans, robots, and vehicles. The second domain (teams of unmanned ground and air vehicles) has a handful of cooperating robots. Both domains involve unpredictable adversaries in the vicinity. The application to integrated flight management at AMC represents our third application. Our approach customizes monitoring behavior for each specific task, plan, and situation, as well as for user preferences.

Dynamic Scheduler. The dynamic scheduler (DS) provides capabilities for assessing the broader impact of events that have caused alerts and for determining

appropriate mitigating changes to the current airlift schedule. As indicated earlier, the DS extends the technology and software first implemented in the AMC Allocator [Kramer & Smith 2002, Smith et al. 2004], a system for day-to-day management of airlift and tanker schedules that is now embedded as an operational module in the AMC CAMPS mission planning system. At its core, the AMC Allocator utilizes incremental, constraint-based scheduling techniques that allow selective re-optimization of allocation decisions to accommodate new higher-priority missions while minimizing disruption to previous assignments.

As resource assignments are made to a given mission, any necessary auxiliary tasks (for example, positioning or depositioning flights or crew rest periods) are generated and inserted into the mission plan. In the simplest case, all missions are planned and scheduled as round trips. Various missions will be sequenced when necessary to satisfy overall resource capacity constraints (and in some cases rejected as unsupportable). It is also possible to direct the system to consider mission merging possibilities, which provides another means for optimizing resource usage. For example, the system might suggest using an aircraft from one mission to support a second mission instead of returning directly back to home station.

Mission scheduling and resource allocation capabilities can be invoked in automated or semiautomated modes. In the latter case, the system generates and compares different options that might be taken. Planners interact with the AMC Allocator through graphical displays, which incorporate mission-oriented, resource-oriented, and map-based views of the current set of commitments.

To provide a dynamic scheduler (DS) for use in an execution management context, the AMC Allocator technology has been extended to accept and respond to updated "state of the world" information. The AMC Allocator's GUI was augmented to include an Agenda Panel for displaying, managing, and examining the effects of alerts received from the FMA Executive. Graphical tools were also developed for visualizing the impact of an alert on the existing schedule. The alerts are communicated via OAA to a new message handling module in the DS, which is responsible for computing the effects of an alert on the existing schedule and passing the alerts to the DS UI. This internal architecture is depicted in Figure 2.

While the DS retains the core constraint-based, incremental scheduling architecture of the AMC Allocator, it has been significantly reengineered and extended to incorporate the constraints and resource models that must be taken into account in an execution-management context (for example, airport MOG constraints that dictate how many aircraft can be accommodated simultaneously). Mission itineraries are modeled with much greater fidelity than in the AMC Allocator, introducing new activities such as take-offs, block-ins, preflights, and postflights. In addition, the DS incorporates a more flexible temporal constraint network model than the AMC Allocator. This new flexibility allows for dynamic extension of activities such as crew rests, which in the AMC Allocator were assigned a fixed duration.

Like the AMC Allocator, the DS supports mixed-initiative scheduling, allowing the end user a range of interaction options, from primarily manual with constraint checking, to user selection of system-recommended options for schedule deconfliction, to fully automated rescheduling actions based on predefined user preferences. The DS incorporates all previously developed options for relaxing constraints in circumstances of constraint conflict, such as overallocating aircraft or aircrews, delaying missions, bumping lower-priority missions, or merging multiple missions into a single mission to reclaim capacity. To resolve problems that involve in-process missions, the DS may also add activity delay and itinerary diversion options.

Dynamic Mediator. DM enables the flight manager to make an effective decision by gathering information from other actors quickly. When the flight manager must alter the schedule in response to an unexpected event, time is an important factor because a delayed decision may require the schedule to be altered even more. For example, when faced with a reduction in MOG capacity, the flight manager needs to make a decision that allocates the remaining capacity to the missions that most require it.

The DM module makes two main assumptions:

(1) No single entity possesses all the information relevant to the decision.
(2) The time allowed for making the decision is limited or a delayed decision is costly.

The originating planners have information relevant to making alterations to the mission schedule that has not been entered into FMA in advance because it is information that is not needed for normal scheduling. For example, for deciding which missions most require the remaining MOG capacity, the cargo contents and the purpose of the mission are often relevant.

Extracting information relevant to decision making is costly because planners must be contacted to extract information. DM automates parts of the process of incrementally extracting only that information that is relevant to the flight manager's decision. The DM lowers the cost of collecting information and computing the correct decision.

Prior to the FMA, the communication was attempted in only the most important decision situations because interpersonal communication was too costly. As a result, the flight manager often makes an educated guess as to the importance of the relevant missions and therefore may make an inappropriate decision based on that guess. The DM module makes communication practical by (1) managing the communication between the flight manager and the planners to focus on relevant information, and (2) storing, organizing, and analyzing the information for the purpose of making a decision. The DM module enables the flight manager to make better decisions during execution, while not precluding the use of personal contact for the most important decisions.

The DM module automates collection of relevant information from planners using queries and replies, implements a search for those queries and replies that minimize the expected communication costs, and enables correct decision making with limited information.

Application Status

We defined a demonstration scenario consisting of several storyboard-level vignettes that illustrate the capabilities of the FMA. The FMA was demonstrated on the vignettes using scripted data feeds that were generated to be as similar to actual data feeds as possible. For instance, one such script uses all 1100 MOG exceptions from the output of an AMC monitoring system. Based on review by subject-matter experts, all the demonstrated vignettes show useful capabilities beyond what is currently provided by existing AMC flight management software.

A brief summary of each vignette follows:
- A MOG conflict is detected by the FMA Executive and resolved by the execution office and planners with assistance of the DS.
- A single event causes multiple, cascading problems. An airplane breaks on the runway of Airport 1, causing both a wing capacity overallocation problem and a cargo stalled problem. The FMA Executive detects the problems and DS-aided responses must handle multiple problems.
- Multiple events (bad weather and an instrument landing system (ILS) failure) when considered together cause a problem. The FMA detects the problem and suggests responses.
- The FMA monitors system behavior and gives alerts or responds to the situation. For example, the FMA might alert when AMC tools that report MOG exceptions and execution-time exceptions are not present or have lost input feeds, or when FMA actors are not present.
- The FMA performs automated responses to a minor problem, controlled by user-established and selected policy.

To give an idea of how the FMA operates, we briefly describe the execution flow of the second vignette above.

Input Event Sequence:
1. The Executive receives a report that the ILS for port P will be offline for a time window [t1, t2] for repairs.
2. The Executive receives a weather exception at P that overlaps with [t1, t2].

- The Executive infers that the airport will be closed for some period because of simultaneous bad weather and no ILS capability. Either event by itself is no problem but together they cause a problem.
- The Executive communicates port closure information to the scheduler.
- The Executive queries the Scheduler for affected missions and alerts the Execution user and affected planning offices, customizing the alert to each actor.
- The Scheduler automatically computes the immediate impact and suggests rescheduling actions:
    - Options include bumping, delaying, overallocating and rerouting.
- The Scheduler computes the “ripple effect” on the downstream schedule.
- The Execution user, possibly collaborating with planning offices using the Dynamic Mediator, selects a schedule fix, after possibly modifying it during interactions with the Scheduler.
- The Execution user and appropriate planning offices are notified of all relevant changes to missions.

The Executive is designed to coexist with and complement the existing flight management software tools currently deployed at AMC. Some existing tools at AMC detect deviations and problems, but they are based on simple rules. Thus, they detect too many false alarms that overwhelm the user with alerts and therefore the user cannot focus on the most important deviations. The FMA improves upon these tools by its VOA computation, which will filter out low-value alerts, and show high-value alerts to those users for whom they have high value. Furthermore, the FMA detects problems that are not detected by existing tools (for example, the closure vignette described above).

Transition tasks. The Executive generally takes inputs in forms that are available in existing AMC tools and databases. The Dynamic Scheduler is already in use at AMC as part of CAMPS. The primary tasks that would be required to transition this technology are as follows:
- The Executive must integrate and interface with any data sources to be monitored.
- The FMA system operates in real time, but must be made more robust with respect to tracking and reasoning about current time.

• Design and implementation of an interactive alert/collaboration GUI or integration with existing GUIs must be accomplished.
• Policies must be encoded to implement AMC procedures; it may be desirable to monitor additional data sources.

Evaluation and Summary

Subject-matter experts determined that the alerts generated and schedule repairs completed using the FMA were correct and valuable in each of the vignettes. The Executive (1) monitors all exceptions from multiple tools, (2) estimates the value of each possible alert, and (3) issues high-value alerts that focus user attention on key problems. Using actor-specific VOA, it effectively filtered and prioritized the alerts generated by existing AMC tools. For example, we ran the Executive and SAAM actors on 1085 actual MOG alerts. The Executive filters all but 242 of the 1085 alerts, of which only one is highest priority, and only eight require immediate attention. The Executive sends the SAAM actor 145 alerts, all of which are lower priority.

Such filtering greatly reduces the amount of information humans must monitor, allowing the humans to concentrate on more important tasks than monitoring large amounts of incoming information. Timely alerts result in faster and better responses to unexpected events. Using the DS to assist with modifications results in more missions being accomplished, more efficient resource usage, fewer constraint violations, and fewer downstream problems. Because the FMA analyzes all inputs against the entire schedule, large, complex schedules can be accurately monitored, and no relevant information is ignored or missed. Finally, our distributed actor architecture ensures that the planners (and other actors) get planner-specific alerts. Thus, planners are kept apprised of the status of their missions and can provide feedback during execution.

The Dynamic Scheduler (DS) provides a range of capabilities for responding effectively and rapidly to exceptional events that have been detected. Upon receipt of an alert from the Executive, the status information contained in the alert is superimposed over the current existing schedule, and a list of resulting issues (e.g., schedule conflicts) is posted on an agenda panel. As the user selects a given conflict to address, the system invokes graphical displays that indicate the impact of the event. The DS can be directed by the user to generate sets of possible actions for resolving a given schedule conflict (e.g., delay, divert, or coalesce a problematic mission). Alternatively, the DS can be invoked automatically by the Executive (if policy permits) to resolve and/or improve the current schedule. As decisions are made as to which recovery course of action to take, this information is communicated back to the Executive for implementation.

Importantly, policies control system responses; for example, some responses can be made more automated and others more interactive. The coupling of intelligent execution monitoring to dynamic scheduling capabilities introduces several further benefits. Users gain a better understanding of the implications of detected events and prospective responses on other current and planned activities; such implications include projected resource shortfalls, potential mission delays or disruptions, and opportunities for schedule improvement. This coupling also provides rapid generation of alternative recovery actions and more globally rational flight management.

Acknowledgments. This research was supported by the Air Force Research Laboratory — Rome, under Contract F30602-02-C-0152. We thank Steve Hofmann for his domain expertise and advice.

References

Adam Cheyer and David Martin, "The Open Agent Architecture", *Journal of Autonomous Agents and Multi-Agent Systems*, Kluwer Academic Publishers, Volume 4 (1-2), pages 143-148, 2001.

Laurence A. Kramer and Stephen F. Smith, "Optimizing for Change: Mixed-Initiative Resource Allocation with the AMC Barrel Allocator", in *Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space*, Houston, TX, October 2002.

Charles L. Ortiz, Jr., Timothy W. Rauenbusch and Eric Hsu (2003). "Dynamic Resource-Bounded Negotiation in Non-Additive Domains." In Victor Lesser, Charles L. Ortiz and Milind Tambe (Eds.) *Distributed Sensor Networks: A Multiagent Perspective*. In Series: Multiagent Systems, Artificial Societies, and Simulated Organizations, Volume 9. Kluwer Academic Publishers.

Timothy W. Rauenbusch (2004). *Measuring Information Transmission for Team Decision Making*. Ph.D. Thesis, Harvard University.

Stephen F. Smith, Marcel Becker and Laurence A. Kramer, "Continuous Management of Airlift and Tanker Resources: A Constraint-Based Approach", *Mathematical and Computer Modeling—Special Issue on Defense Transportation: Algorithms, Models and Applications for the 21st Century*, 39 (6-8), 2004.

David E. Wilkins, Thomas J. Lee and Pauline Berry, "Interactive Execution Monitoring of Agent Teams", *Journal of Artificial Intelligence Research*, volume 18, pages 217-261, March 2003.
http://www.jair.org/abstracts/wilkins03a.html.html

Self-interested Planning Agents using Plan Repair

Roman van der Krogt and **Mathijs de Weerdt***
Transport Research Center Delft
Delft University of Technology
Delft, The Netherlands
{r.p.j.vanderkrogt|m.m.deweerdt}@ewi.tudelft.nl

Abstract

We present a novel approach to multiagent planning for self-interested agents. The main idea behind our approach is that multiagent planning systems should be built upon (single-agent) *plan repair* systems. In our system agents can exchange goals and subgoals through an auction, using their own heuristics or utility functions to determine when to auction and what to bid. Some experimental results for a logistics domain demonstrate empirically that this system supports the coordination of self-interested agents.

Introduction

Most of the interesting applications of planning involve more than one agent to plan for. Often these agents are self-interested and require some privacy concerning their plans and the dependencies of actions in their plans on other agents' actions. We propose a system in which *self-interested* agents can i) *construct their plans* themselves, ii) *coordinate* their actions during planning, and do so while iii) maintaining their *privacy*. With this system we take the challenge of negotiated distributed planning that "methods must be developed for adapting the various [existing] approaches in a way that is consistent with the resource-constrained nature of planning agents: planning should be a continuous, incremental process at both the individual and group level." (DesJardins *et al.* 2000).

Our idea is to combine a dynamic planning method for each agent with an auction for delegating (sub)tasks. However, to coordinate subtasks we should deal with inter-agent dependencies (Malone & Crowston 1994) to prevent deadlocks. Currently, multiagent planning methods manage inter-agent dependencies at a central place (Wilkins & Myers 1998), or by constructing and communicating a (partial) global plan (Decker & Li 2000; von Martial 1992). Obviously, in many applications, agents are not prepared to share this kind of information.

In our system, we have a number of agents that first concurrently plan for a single goal, after which they take part in an auction (if there is any) to exchange goals and subgoals. Then, they apply a plan repair technique to add another goal to their plan, and take part in an auction again. They continue to alternatingly perform these steps of adapting a plan using plan repair and taking part in an auction until a complete and valid plan is computed. When an agent gets a task assigned on which others depend, we use a heuristic that lets the agent schedule it early in its plan to prevent cyclic dependencies. Furthermore, we give the agents some high-level information about the services others can provide to reason about which subgoals they should auction.

As an example of a situation in which this type of planning is required, consider the following logistics problem. In this domain, a number of independent planning agents have to transport goods between different locations in different cities. Each of the agents is capable of only a select number of actions: for each of the cities, there is an agent that is capable of transporting goods *within* that city, using trucks. For transport *between* cities, only one agent can transport goods by air from one airport to another. Thus, for a typical transportation order, three agents have to work together: one to bring the goods from their current location to the airport in that city, one to transport the goods to another airport, and a third and final agent is required for the transport from that airport to the destination within that same city. As these agents are different companies, they are self-interested and competitive. However, they are willing to help each other, provided adequate compensation is offered.

In this paper we show how such companies can construct their plans individually, and meanwhile coordinate (some of) their actions while maintaining their privacy. In the next section we define an abstract version of this problem more precisely, and we show how a propositional plan repair method can be combined with a simple auction to deal with this problem. Also we present solutions to subproblems such as the prevention of deadlock, and dealing with agents that do nothing but accepting orders to sell them again. The given logistics problem is used to show the suitability of these ideas. Finally, in the discussion we summarize our findings, compare them with related work, and give away our ideas for further study.

A Multiagent Planner using Plan Repair

In this paper we propose a method that dynamically creates, coordinates, and repairs plans for agents that do not want to share crucial information. We base this work on the work of

*This research is supported by the Technology Foundation STW, applied science division of NWO, and the technology program of the Ministry of Economic Affairs.

propositional planning, see e.g. (Kambhampati 1997). We focus on problems that can be modeled as a *set of distinct propositional planning problems* $\Pi_a = \langle O_a, I_a, G_a \rangle$, one for each agent a. In such a problem the set O_a is the set of actions that the agent a can perform, I_a is the part of the initial state the agent can observe, and G_a are the goals to be achieved by the agent. The initial state is described by propositions, an action by its preconditions and effects, and goals, preconditions, and effects are all defined by conjunctions of literals. The problems of all agents are mutually distinct, meaning that we require that there are no agents able to perform actions using the same resources (i.e., described by the same propositions) and each agent has complete knowledge about its own problem. At first this may seem too restrictive, but in many domains agents (companies) that are not cooperative can indeed not use each other's resources. For some resources of general use where conflicts may occur (such as cross roads) we may introduce an additional agent to coordinate the use of such a resource.

Note that in a propositional planning problem there is usually no realistic model of the costs and duration of actions, nor of deadlines. Therefore, the length of the plan (i.e., the number of actions) is used as an indication of the costs.

To render the problem more manageable, we assume that all actions in the domain can be undone (are reversible), and that there are no goals that are inherently unattainable. This assumption ensures that in principle a solution can always be found. We also assume that agents do not break contracts, unless they really cannot hold to them, in which case they inform the other party immediately.

The above-mentioned assumptions help us to focus on the more interesting and more difficult problem of designing a system

- that only communicates offers and bids, but little else, and
- in which agents can auction (sub)tasks to other agents, while preventing
  - cyclic dependencies,
  - lazy agents, and deal with
  - decommitment of subtasks by subcontractors.

In the following section we lay out the design of such a system and we explain how to deal with problems that may occur when building it. The crux of our idea is quite simple: coordinate (single agent) plan repair systems through a task auction. To implement this idea, we suppose that we have a dynamic planner for such problems at our disposal, such as the Partial Order Plan Repair system (POPR) (van der Krogt & de Weerdt 2005), which is based on VHPOP (Younes & Simmons 2003). Although we use the same planning system for each of the agents in our discussion, we do not rely on the specifics of this planner for the coordination of the agents. This ensures that we truly simulate a situation in which each agent is free to choose its plan repair system, although these should be extended with a common communication module.[1] We assume that such a system includes a heuristic function $\mathcal{U}(P, \Pi, g)$ that, given a problem Π and a plan P, estimates the costs of adapting P to achieve another goal g. Usually such a system is only able to solve *single* instances of a propositional planning problem, not a combination of them. How to combine a number of these systems to form a multiagent planning system is the topic of this paper.

[1] Such a plan repair system can be derived from the planning system that the agent is currently employing by using the techniques of (van der Krogt & de Weerdt 2005).

Planning The first important decision made to achieve the properties described above is to process the goals one-by-one by a plan repair system, instead of in a single batch (as is usual in planning). This has a number of advantages: firstly, failure to add a goal to the plan immediately tells us which goal we should put up for auction, while when planning for a batch of goals fails, it is not immediately obvious which of the goals cannot be achieved. Secondly, we get regular moments at which we can easily make changes in the problem. Moreover, at these moments we have a valid plan that partially achieves our goals to base our decisions on. This means that we can make a more informed decision than if we would interrupt a regular planner at certain points. There is one disadvantage, however: we cannot as easily exploit positive interactions that may exist between goals. In the section on our experimental work, we shall come back to this issue.

We now describe the basic steps of this goal-by-goal planning approach. The process starts by taking the original planning problem Π, and creating a goal queue Q from it (containing all the goals that are to be solved in order to solve Π) as well as the problem Π_{PR}, initially identical to Π, but without goals. We use this problem Π_{PR} to keep track of the problem that we are trying to solve in the current iteration. Later it may contain additional goals that this agent has accepted from others, and it does not need to include all goals from Π (as some might not have been planned for yet, or are currently planned for by other agents). To plan a single goal g from Q the system performs the following steps:

1. It queries the planning heuristic $\mathcal{U}$ of the plan repair system to estimate the cost of adding g to the plan.
2. The heuristic may report that it cannot incorporate the goal, or that the costs of incorporating are so large that it is preferable to ask other agents for help. If this is the case, the agent passes this goal to the blackboard for auction. Otherwise, it removes it g from Q, adds it as a goal to Π_{PR}, and updates its plan for this new planning problem using the plan repair system.

These steps are interleaved with processing auctions (if any), as discussed hereafter. Once the goal queue is empty, each goal of the agent is either planned for in its own plan, or has been given to another agent (via the blackboard). From this point on, the agent stays active to respond to auctions until all other agents are finished as well.

Having described the basic planning loop, we turn to the multiagent specifics. First we discuss the auctioning of goals. Then, we describe a way with which we can, during the planning phase, decompose a goal into subgoals, some of which the agent might not be able to achieve itself. These

subgoals then can also be auctioned.

Auctions As said, an agent planning a goal first consults its planning heuristic to discover whether it is advisable to plan for this goal itself. If it is not, or if it turns out after plan repair that the goal is unattainable, the agent will put the goal up for auction. For ease of implementation, this auction is currently run by a blackboard, but it can be distributed over the agents as well, of course. The blackboard keeps a list of auctions, and processes them one-by-one. This prevents additional difficulties that agents face when dealing with multiple simultaneous auctions (such as the "eager bidder" problem (Schillo, Kray, & Fischer 2002)). For each auction, the blackboard sends out request for bids. Note that we currently process the auctions in order of arrival. In the future, we might include a priority which determines the order of the auctions.

When an agent receives a request to bid on a goal, a heuristic is applied to discover whether this agent can incorporate the new goal in the current plan. If so, this also tells us what the estimated cost is of adapting the plan. This value is then sent as our bid for this goal. In the current system, we have chosen to allow the agents a single, sealed bid.

The blackboard waits for all bids, and selects the cheapest bid. The winner is awarded the goal, and receives payment equal to the second-lowest bid (Vickrey 1961).[2] Upon being awarded a goal g, the agent adds g to the front of its goal queue. This ensures that this goal is processed next, and that the agent can actually attain g by repairing its plan. If we allow other goals to be processed first, g might no longer be achievable. This would require decommitment of the agent, a situation that we would rather prevent. Only in the unlucky event that during plan repair it turns out that the heuristic was completely wrong, decommitment is performed as discussed on the next page.

Services Exchanging goals is necessary but not sufficient for a complete multiagent planning system, for it is often the case that a certain goal cannot be achieved by a single agent, but only through cooperation. For example, moving a package from one city to another in our example logistics domain requires three agents to work together. Hence, we also need to be able to decompose goals into subgoals that may have to be carried out by other agents. To continue our example, moving a package from one city to another decomposes into three subgoals: delivering the good to the airport in the source city, bringing the good to the airport in the destination city, and finally the delivery to the destination. Decomposition is not trivial, but fortunately, we can do some decomposition during the planning phase.

To perform decomposition during planning, agents need some knowledge on the actions (or groups of actions) that other agents can perform. We encode such knowledge as *services*. A *service* is a task that can be achieved by one or more other agents. It is not required to know *how* a task is achieved, nor is it required to know *who* can exactly achieve portions of a task. For example, in the distributed logistics domain, a trucking agent in a city might know that other agents can achieve the task to bring a certain package from other cities to the airport in his own city.

We model services as regular actions. To distinguish such actions from regular actions we refer to them as *external actions*. Like regular actions, external actions can be integrated in the plan during the planning phase to indicate that help from other agents is required. At the end of a planning iteration (in which an additional goal from the goal queue is planned for), the effects of new external actions are sent to the blackboard for auction. In this way, propositions can be "exchanged" between agents.

The complete planning loop Having described the features of the algorithm in isolation, we now end this section with the complete algorithm as we have used in our experiments. The algorithm is presented below, and starts with setting up some data structures, such as the goal queue Q, and the initial planning problem Π_{PR}. Then, in step 4, it tries to add a goal from the queue to the current plan P. At first, in step 4.2, we compute the heuristic value $\mathcal{U}(P, \Pi_{PR}, g)$ of establishing g with P. If this is estimated to cost more than the agent is willing to spend (with an unsatisfiable goal returning ∞), we send the goal to the blackboard for auction. Otherwise, we update the planning problem Π_{PR}, and compute the new plan. If this plan contains any external actions, the subgoals they satisfy are sent to the blackboard for auction. Having processed a goal from the queue (if any), we check whether a goal g' is currently being auctioned. If so, we compute our costs for it (using the heuristic $\mathcal{U}$ of the plan repair system), and send this as a bid to the blackboard. If our bid is winning, we add the goal g' to the front of our goal queue.

PLANNINGLOOP (Π)
Input: *A problem* $\Pi = \langle O, I, G \rangle$

begin
1. *Setup the goal queue* Q *containing all goals from* Π
2. *Create the initial problem description* $\Pi_{PR} = \langle O, I, \emptyset \rangle$
3. *Create the initial (empty) plan* P
4. **if** Q *is not empty* **then**
   - 4.1. **pop** *goal* g *from* Q
   - 4.2. *Estimate cost for this goal:* $c = \mathcal{U}(P, \Pi_{PR}, g)$
   - 4.3. **if** *the goal is too expensive* **then**
     send g *to the blackboard for auction*
   - 4.4. **else**
     - 4.4.1. *Update problem:* $\Pi_{PR} = \Pi_{PR} \cup \{g\}$
     - 4.4.2. *Update plan:* $P = PR(P, \Pi_{PR})$
     - 4.4.3. **if** P *contains external actions* **then**
       request *results (subgoals) of these actions via an auction (blackboard)*
5. **if** *an auction is ongoing for a goal* g' **then**
   - 5.1. **send** *bid (which is* $\mathcal{U}(P, \Pi_{PR}, g')$*)*
   - 5.2. **if** *goal is awarded* **then**
     push g' *onto the front of* Q
6. *goto step 4*

end

[2] Note that with a repeated auction the main advantage of a Vickrey auction (that it is a dominant strategy to bid ones private value) is lost for agents that reason about future auctions. Other types of auctions are a topic for future study.

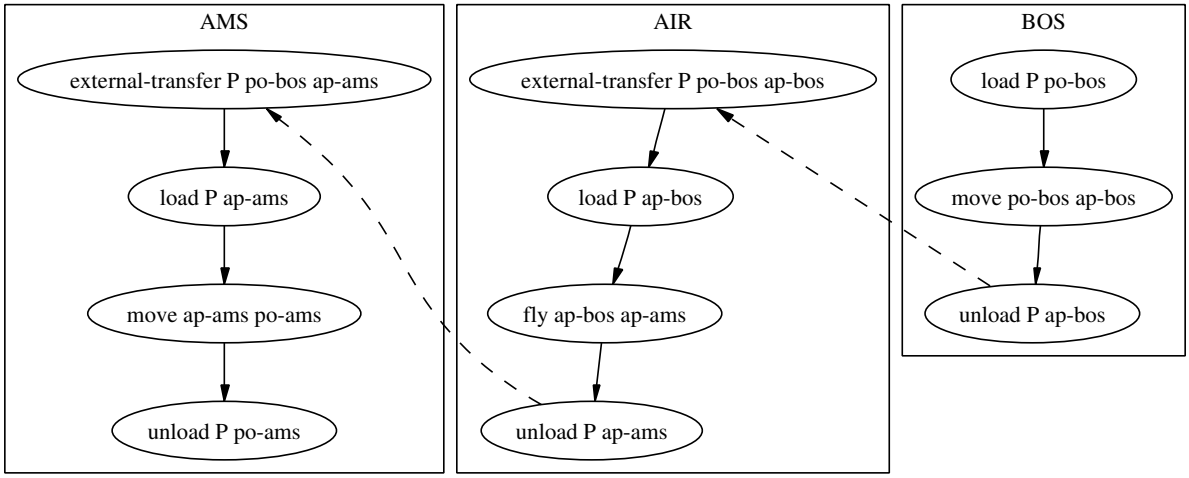


Figure 1: The multiagent plan for transporting package P from $po{-}bos$ to $po{-}ams$ using three agents: BOS, AIR, and AMS.

Note that besides the details mentioned in the algorithm a bit more bookkeeping is necessary. For example, we have to detect when everybody has finished planning for all their goals. Currently this is being recorded at the blackboard, where every agent can declare itself ready (and can remove that declaration when it accepts a new goal from the blackboard). Also, in step 4.4.3 where the subgoals from external actions are auctioned, we should only send those subgoals that were not present in the plan before. Finally, we observe that, currently, agents do not receive feedback on their goals that have been sent for auction. That is, if none of the other agents bids on a (sub) goal, the auctioneer will continue to periodically try to auction this goal, instead of reporting this to the original sender who should then try and find another solution. For the domains that we have used in our experiments, this is no problem, as there is always someone who can achieve the goal (the goals do not have time limits). For more complicated domains this does not hold, of course. We will come back to this issue in our discussion on future work.

Example. Suppose the following logistics problem. There are two cities, Amsterdam and Boston, each with an airport (denoted by $ap{-}ams$ and $ap{-}bos$, respectively). A package P is to be transported from the post office in Boston ($po{-}bos$) to the post office in Amsterdam ($po{-}ams$). In each city, there is a transportation company that uses trucks to transport goods within the city. Furthermore, there is an airline company that can transport goods between airports. In the domain of each agent one external action is present, called `external-transfer`, that describes that other agents are capable of transportation as well.

Initially, the goal (from $po{-}bos$ to $po{-}ams$) is given to the trucking agent in Amsterdam (we refer to this agent as AMS, the trucking agent in Boston is referred to as BOS, and the airline agent is denoted by AIR). AMS queries its heuristic and finds out that it can reach this goal. Using plan repair on the empty plan, it comes up with the following plan:

1. `external-transfer` P from $po{-}bos$ to $ap{-}ams$
2. `load` P at $ap{-}ams$
3. `move` from $ap{-}ams$ to $po{-}ams$
4. `unload` P at $po{-}ams$

The effect of the external action is a proposition $at(P, ap{-}ams)$, which AMS sends to the blackboard. In the following auction, BOS bids ∞ (it cannot reach this goal) and AIR bids 4. Hence, the goal is awarded to AIR, which creates the following plan for it:

1. `external-transfer` P from $po{-}bos$ to $ap{-}bos$
2. `load` P at $ap{-}bos$
3. `fly` from $ap{-}bos$ to $ap{-}ams$
4. `unload` P at $ap{-}ams$

This results in the goal $ap(P, ap{-}bos)$ being auctioned, which is subsequently won by and planned for by BOS. This completes the multiagent plan for delivering the package from Boston to Amsterdam using all three agents as shown in Figure 1.

Coordination problems

Whereas the former section gave an overview of the basics of our multiagent planning system, in practice one bumps into some additional difficulties that need to be solved. In this section we pay attention to three important issues that we encountered in building the system.

"Lazy" agents The first issue that we encountered in our initial experiments was that sometimes an agent accepted a goal from the blackboard, and then planned to use a service to have other agents satisfy the exact same goal. We called these agents "lazy agents", for they did not want to do some work themselves. In some experiments, this is a minor inefficiency, but in other problems two of such lazy agents were present who were continuously bouncing the same goal back-and-forth. As a solution we considered goals *tabu* for production by external actions. That is, when adapting the plan to include a goal g, no external actions may be planned

that produce g. We adapted both the planning algorithm and the heuristic to honor these tabus.

Decommitment of bidders As indicated in the previous section, agents place their bids based on a heuristic estimate of the costs of changing their plans. Using a heuristic has two important repercussions: firstly, an agent may need to modify its plan in a far greater (and possibly more expensive) way than anticipated, and secondly, an agent may find that it cannot achieve the goal at all. When an agent discovers that it cannot actually satisfy the goal it has bid on, it indicates this in a message to the blackboard. The blackboard then re-auctions the goal, disregarding the bids of agents that have bid on the goal and rejected it before. Under the assumptions we made, there is always at least one agent capable of achieving the goal. Since bids of agents that have rejected a goal are disregarded, the goal will eventually be awarded to the agent that can satisfy the goal. For now, a decommitting agent pays (as a penalty) the cost difference between its own bid and the next one.[3]

Managing dependencies A distributed planning system, such as presented in the previous section, should ensure the validity of the proposed plans. For this, it is required that not only the individual plans are valid, but also that the combination of plans is valid. In particular, we should verify three conditions:

1. Actions in different plans may not interfere. As explained in the previous section, strongly autonomous agents, such as competitive companies, usually have distinct areas of control, or an additional agent can be introduced to ensure mutual exclusion of shared resources.
2. If an agent depends on another agent to provide a subgoal, another agent should actually provide this subgoal, and
3. the combined plans may not contain *cyclic dependencies*. That is, it may not be that an action a is (indirectly) dependent upon an action (of another agent) that is dependent on an effect of a.

The second condition is ensured by our assumption that all agents are sincere: when an agent promises to provide a subgoal, it will either do so, or inform the blackboard that it cannot. The last condition is the most difficult to guarantee, because in principle agents need to know the details of the other agents' plans to ensure this property. In existing solutions to prevent cyclic dependencies either a central facility is keeping track of dependencies (Wilkins & Myers 1998), or agents communicate to form a so-called partial global plan (Decker & Li 2000).

In our goal-by-goal approach, however, we can use a so-called backward planning heuristic. When an agent plans a task for someone else, it can prevent cycles from occurring without any additional communications by placing all actions (possibly including external actions) required for this task *before* all other actions in its plan. This heuristic depends for a great deal on the fact that only one goal is auctioned by the blackboard in a single iteration. Thus, only one agent can create a new inter-agent dependency at a time. If we ensure that this new dependency is not dependent upon previously existing dependencies, we prevent cycles from occurring. Any additional external actions inserted will be auctioned only after this part of the plan has been completed. Note that the need for this heuristic disappears when using a domain in which time is explicitly represented and a planner that can reason with time, since time attributes can be used to prevent cyclic dependencies.

Example. Consider the logistics problem of the previous example. Suppose that a second package P' would have to be transferred in the opposite direction, i.e., from the post office in Amsterdam (*po−ams*) to the post office in Boston (*po−bos*). BOS already had a plan for transporting P from *po−bos* to *ap−bos*, but now it creates the following plan for this situation:

1. `external-transfer` P' from *po−ams* to *ap−bos*
2. `load` P' at *ap−bos*
3. `move` from *ap−bos* to *po−bos*
4. `unload` P' at *po−bos*
5. `load` P at *po−bos*
6. `move` from *po−bos* to *ap−bos*
7. `unload` P at *ap−bos*

After the auctions for transporting P have been dealt with, AIR and BOS have computed the plans from the previous example. Then, the auction of $at(P', ap{-}bos)$ takes place, which is won by AIR. Due to the backward planning heuristic, AIR inserts the actions for this subgoal before its other actions:

1. `fly` from *ap−bos* to *ap−ams*
2. `external-transfer` P' from *po−ams* to *ap−ams*
3. `load` P' at *ap−ams*
4. `fly` from *ap−ams* to *ap−bos*
5. `unload` P' at *ap−bos*
6. `external-transfer` P from *po−bos* to *ap−bos*
7. `load` P at *ap−bos*
8. `fly` from *ap−bos* to *ap−ams*
9. `unload` P at *ap−ams*

Thus, the added actions (related to P') do not have to wait for existing actions to finish. Hence, AIR cannot create a cyclic dependency. Had it tried to reuse part of its existing plan (like AMS did) by first waiting for the external action related to P (step 6), it would have created a cyclic dependency, because BOS firsts waits for the external action related to P' to finish (step 1).

Experimental Results

For our experiments we applied the method described in the previous section to the POPR plan repair system (van der Krogt & de Weerdt 2005), which is an adaptation of the VHPOP planner by (Younes & Simmons 2003). We used a series of benchmark problems from the AIPS competition (Bacchus *et al.* 2000) in the logistics domain that was used as an example before. We took a total of 11 problems, varying from 2 to 5 cities, and from 4 to 15 goals. The number of cities grows as the number of goals do: for a problem

[3] We are considering leveled-commitment contracting (Sandholm 2002) to enable strategic decommitting.

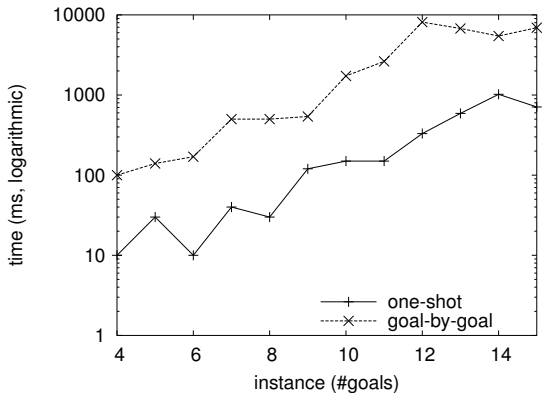


Figure 2: Run times of one-shot planning and goal-by-goal planning by a single agent.

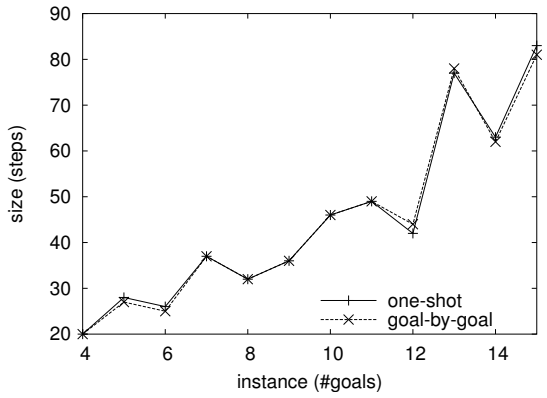


Figure 3: Plan lengths of one-shot planning and goal-by-goal planning by a single agent.

with n goals, $\left\lceil \frac{n}{3} \right\rceil$ cities are used. Each goal consists of the transport of a single package from one location to another. Transportation can be performed by truck (within a city), or by plane (between airports in different cities). Sometimes, the transportation orders are within a city, but for most orders, the destination city is different from the starting city. Because there are no deadlines in this domain, goals can eventually always be achieved.

Goal-by-Goal Planning

The first question that we posed is the following. In our current approach, we choose to plan goal-by-goal. That is, instead of considering all goals at once, in a single planning problem, we first plan for one goal, then add another, etc. The question is how this affects the quality of our solutions, and of course the speed with which we reach these solutions. To investigate this, we compared the plans produced by VHPOP (which plans in a single batch) and the plans produced by POPR, when given a series of plan repair problems in which the goals were added one by one.

Figures 2 and 3 show the runtime and plan quality respectively when goal-by-goal planning is compared with solving a single planning problem involving all goals. Although, in principle, a planning problem can be solved more efficiently by dividing it into subgoals (Korf 1987), we can see from Figure 2 that goal-by-goal planning takes quite some more time than solving a single planning instance. This is due to the fact that for each goal, a new planning problem is created, which invalidates a lot of the structures that were created before. For example, part of the heuristic that POPR inherits from VHPOP relies on a planning graph, which is currently created completely anew for each planning problem, whereas it can be reused when we solve multiple goals within a single problem. In principle, this could be mitigated by realizing that we are not solving *any* plan repair problem, but a specific one in which only a single goal is added. This would allow the unrefinement heuristic (which can be used to solve general plan repair problems) to reuse some of the existing structures that are not invalidated.[4] For our current system, we have not done so, however.

Concerning the quality of the plans, we can see from Figure 3 that one-shot planning for a single problem or goal-by-goal planning makes hardly any difference. This is to be expected, as VHPOP (also) uses a LIFO queue for its goal agenda and hence tries to completely satisfy one goal (including all subgoals that lead to this goal) before working on the next one.

Multiagent Planning

The more important part of our experiments obviously has to do with multiagent planning. In particular, we want to verify that our approach is feasible. The planning problems used in the previous section were translated into their multiagent counterparts by introducing a number of agents as follows: for each city, we introduced an agent that is capable of transport within that city (using trucks). One additional agent was given control over the airplanes, and is hence capable of inter-city transports. We used two types of domains: one for the inner-city agents and one for the inter-city agent. The former consisted of the usual `load`, `unload` and `move` actions, with which cargo can be loaded, transported and unloaded. In addition, we added an `external-transport` action that represents the knowledge of the other agents' capabilities. It specifies that any package in any location can be moved to the local airport by some means. The domain of the inter-city agent also consisted of four actions: `load`, `unload` and `fly` to transport goods from one airport to another, and an `external-transport` action specifying that other agents are capable of transporting goods between two locations within the same city.

These logistics problems were used to verify that our approach is feasible. The run time for three different cases can be seen in Figure 4. The first case is labelled *one-shot*. This shows the run time of the unmodified (central) VHPOP planner on the benchmark problem. In this case, we have a

[4]For example, the reason to regenerate the planning graph is that the initial state or the set of available actions might have been changed. Since this is not the case in our specific plan repair problems, the planning graph can be reused.

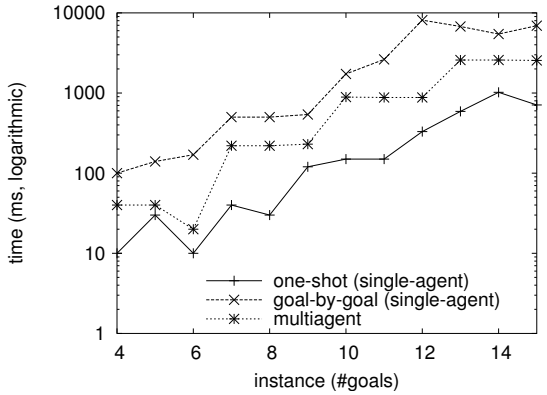


Figure 4: Run times of multiagent planning compared to single-agent planning. The time reported for the multiagent experiments is the time it took the slowest of the agents to compute its plan (the "make span").

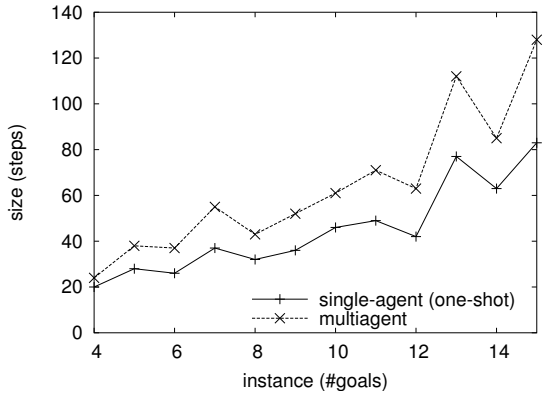


Figure 5: Plan lengths of single-agent one-shot planning, and the cumulative size of the multiagent plans.

single agent that plans for all trucks and airplanes. (Clearly, this is a hypothetical situation for a domain involving self-interested companies.) The second case, labelled *goal-by-goal* shows the amount of time it takes a single planner to create a plan for this problem when it uses a goal-by-goal approach. The third case is labelled *multiagent*. In this case, we used one planner to plan for the transport of goods in each of the cities (thus, for n cities, we used n planners) and a single planner for the planning of inter-city transportation orders (thus, a total of $n + 1$ planners for problems with n cities). As we can see, for these problems multiagent planning is considerably faster than planning centrally using a goal-by-goal approach, due to the fact that we can have different agents plan in parallel.[5] Notice that the differences are significant (as one might guess from the figure), as can be seen from the results of paired t-tests we performed:

		t	p
one-shot	goal-by-goal	-3.0756	< 0.02
one-shot	multiagent	-3.106	< 0.01
goal-by-goal	multiagent	2.7874	< 0.02

Besides run-time performance, plan quality is also important. Figure 5 shows the size of the resulting plans. As expected, the backward planning heuristic that we employ has a negative effect on the size of the plans, compared with a centralized solution. This is because it forces an ordering on the agents' plans that is stricter than necessary. As a result, the plans that we obtain are significantly bigger (a paired t-test results in t=-5.0344 and $p < 0.01$). This is the price one has to pay for not exchanging detailed information on the structure of the plans. An important question for future work is whether we can relax the ordering that is imposed by the heuristic a little, allowing us to reuse a part of the existing plan.

Discussion

In this paper we gave experimental evidence that self-interested agents can plan and coordinate their plans while only exchanging a very small amount of information. Our method should work with any plan repair algorithm, allowing agents to choose their own dynamic planner. We described how to use such an existing plan repair algorithm in a goal-by-goal setting and a simple auction, we showed how to prevent cyclic inter-agent dependencies, and how to deal with lazy agents and decommitment by a bidder that overshooted itself.

We studied the difference in both plan size and planning time between multiagent planning and single-agent planning. It turns out that our distributed approach produces longer plans than central solutions. This can be mainly attributed to our cycle-prevention heuristic, which is often too restrictive. However, it allows us to create valid multiagent plans without exchanging details about the plans, which is very important for self-interested agents.

The distribution of the planning problem in a multiagent planning system leads to an improvement of planning performance compared to a single-agent solving a planning problem goal-by-goal. We expect that for more realistic and more complicated domains the difference may be even larger, since agents can do a lot of work in parallel. Summarizing, from the experiments we conclude that it is indeed possible to use multiple single-agent plan repair systems to let self-interested agents plan for their goals individually, and request (or provide) help when necessary.

Related Work

This system for coordinating self-interested agents using propositional plan repair is unique in that we do not assume that the agents are *collaborating*. Agents may even be each

[5]For these easy problems a planning cycle takes about 5-10ms, while one communication takes about 40ms, because Linux schedules processes in slots of at least 10ms and communication uses at least 4 different processes. In realistic (i.e. complicated) domains the planning component is the dominating factor in the total run time. Therefore we focused on the time required for planning. The total time including communication is only slightly better than the single-agent goal-by-goal results for these simple problems.

other's competitors. Previous work on multiagent planning, although often more advanced in modeling problems realistically (by involving time constraints, minimizing costs, and efficient use of resources) assumes that the agents are collaborative. For example, in the Cougaar system (Kleinmann, Lazarus, & Tomlinson 2003) cooperative agents are coordinated by exchanging more and more details of their hierarchical plans until conflicts can be resolved (similar to (von Martial 1992)).

The Generalized Partial Global Planning (GPGP) method (Decker & Lesser 1992; Decker & Li 2000) describes a framework for distributedly constructing a (partial) global plan to be able to discover all kinds of potential conflicts. In GPGP agents exchange parts of their plans, so that each agent can build a partial global plan, containing the knowledge that this agent has of the other agents' plans. Using this partial global plan, the agent can detect possible positive and negative effects, and deal with them. To use GPGP, however, the agents need to trust each other with some of the details of their plans. Self-interested agents are not prepared to do this. A similar line of reasoning holds for most of the cooperative (often hierarchical) and mixed-initiative multiagent planning systems. Of these, the idea of planner-independent collaborative planning by Kim and Gratch (Kim & Gratch 2004) is particularly interesting in view of our idea for planner independence. They use such planners to solve small problems that can support the decision process of the user. In their situation there is no need for plan repair or cooperation.

Thirdly, in (Brenner 2003) a method using partially ordered temporal plans is proposed to solve multiagent planning problems in such a way that agents can ask others about the state of the world, who will (truthfully) answer as soon as possible. This work relaxes our assumption that agents have complete knowledge about the relevant part of the world, but in all of the above mentioned systems the agents are not self-interested.

Finally, we would like to compare our method to a multiagent planning approach based on the COMAS system (Cox, Elahi, & Cleereman 2003). In their approach each agent has one or more *unique* capabilities. Each agent can directly request such a 'specialist' when it needs its capability (based on knowledge about other agents' capabilities). The requesting agent is then sent a complete subplan that it can include in its plan. Besides the exchange of a lot more information than in our method (both beforehand and during planning), their system also takes a rather simple approach to preventing cyclic dependencies: they assume that actions that can possibly lead to cyclic dependencies (e.g. the load/unload pair of actions in logistics) can only be executed (and hence planned for) by a single agent.

Plan merging systems (Tsamardinos, Pollack, & Horty 2000; de Weerdt *et al.* 2003) can also be used by self-interested agents in order to coordinate their plans. In these systems, each agent builds its own plan, without exchanging information with the other agents. When all plans have been computed, limited information is exchanged to detect possible interactions. Clearly, these approaches are static and cannot be used to request help from other agents during the planning process.

Another line of research concerns the *reasoning* behind the creation of multiagent plans. Examples of this type of research are the work on *joint intentions* by Cohen and Levesque (1991) and the SharedPlans approach of Grosz and Kraus (1999). Although both these approaches focus on collaborative behaviour, some aspects are important to our work as well. Firstly, when one of our agents carries out an action to bring about a subgoal of another agent this can be seen as a particular type of joint intention. Secondly, we are considering a formalisation of our approach similar to the theory of elaborating multiagent plans as presented in the SharedPlans framework

Next to this work on coordinating multiagent plans, there is also a substantial body of work on *task allocation* for self-interested agents. For example using market mechanisms (Walsh & Wellman 1999), or using extensions of the contract-net protocol (Collins *et al.* 1998; Smith 1980). Ideas from this work may be used to improve the simple auction of our approach, for example to enable parallel or combinatorial auctions. Task (re)allocation, however, cannot completely be disconnected from planning. In our work we focus not so much on task allocation, but on coordinating the agents' *planning* and *plan repair* behavior (without the construction of a global set of constraints).

Future Work

Since our initial experiments showed promising results, we intend to continue this line of research towards a fully equipped multiagent planning system. Besides looking at improvements to our heuristic, one of the first things to do is to relax some of our assumptions to be able to tackle more advanced problems. First of all we would like to have a method to estimate the costs of external actions. Typically "external" actions are more expensive than your own actions. If all actions have costs, we can try to optimize costs instead of plan length. In most domains this may give more realistic solutions. For example, there may be two airports in a city, each serviced by a different airline company. Our current system cannot distinguish between the two options. When the costs of such external actions are known, the most efficient option can be chosen. Another important topic for future study is using a different type of auction and (de)committing mechanism (e.g. (Hoen & Poutré 2003; Sandholm 2002)) that matches the specific requirements of efficiently allocating sets of subtasks to self-interested planning agents.

Another important issue for further study is to give feedback to the agent on their auctioned goals. As we indicated in a previous section, the agents currently submit their auctions to the auctioneer, and assume they will be successfully auctioned. However, it may very well be that no other agent bids on a certain goal, in which case the agent submitting the auction should reconsider its plan, since its subgoals cannot be achieved. Also, in many applications, agents may need to deal with a very dynamic situation where actions may turn out to be disabled or planned goals may become useless. We would like to find an efficient coordination mechanism that can use the plan repair systems of the agents to remove parts

of their plans that become irrelevant.

Furthermore, the algorithm for each agent is currently sequential: it processes a goal, then an auction, then a goal again, and so on. In the future we would like to have two independent subprocesses per agent taking care of each of these tasks. The same holds for the blackboard: it auctions goals one at a time, whereas we might want to have multiple simultaneous auctions, or smart heuristics for ordering the goals before auctioning. Finally, we would like to investigate whether exchanging just a tiny bit more information about the dependencies of actions (or for example making contracts that include time constraints) can lead to a more efficient plan and to more individuality by relaxing the heuristic of 'planning actions for others first in your plan'.

References

Bacchus, F.; Kautz, H.; Smith, D. E.; Long, D.; Geffner, H.; and Koehler, J. 2000. The Fifth International Conference on Artificial Intelligence Planning and Scheduling Systems Planning Competition. http://www.cs.toronto.edu/aips2000/.

Brenner, M. 2003. Multiagent planning with partially ordered temporal plans. In *Proceedings of the Doctorial Consortium of the International Conferenence on AI Planning and Scheduling*.

Cohen, P., and Levesque, H. 1991. Teamwork. *Nous* 25(4):487–512.

Collins, J.; Tsvetovatyy, M.; Gini, M.; and Mobasher, B. 1998. MAGNET: A multi-agent contracting system for plan execution. In *Proceeding of the Workshop on Artificial Intelligence and Manufacturing (SIGMAN-98)*.

Cox, M. T.; Elahi, M. M.; and Cleereman, K. 2003. A distributed planning approach using multiagent goal transformations. In *Fourteenth Midwest Artificial Intelligence and Cognitive Sciences Conference*, 18–23.

de Weerdt, M. M.; Bos, A.; Tonino, J.; and Witteveen, C. 2003. A resource logic for multi-agent plan merging. *Annals of Mathematics and Artificial Intelligence, special issue on Computational Logic in Multi-Agent Systems* 37(1–2):93–130.

Decker, K. S., and Lesser, V. R. 1992. Generalizing the partial global planning algorithm. *International Journal of Intelligent and Cooperative Information Systems* 1(2):319–346.

Decker, K. S., and Li, J. 2000. Coordinating mutually exclusive resources using gpgp. *Autonomous Agents and Multi-Agent Systems* 3(2):113–157.

DesJardins, M. E.; Durfee, E. H.; Ortiz, C. L.; and Wolverton, M. J. 2000. A survey of research in distributed, continual planning. *AI Magazine* 20(4):13–22.

Grosz, B. J., and Kraus, S. 1999. The evolution of SharedPlans. In Rao, A., and Wooldridge, M. J., eds., *Foundations and Theories of Rational Agency*. Dordrecht, The Netherlands: Kluwer Academic Publishers. 227–262.

Hoen, P.J., t., and Poutré, J.A., L. 2003. A decommitment strategy in a competitive multi-agent transportation setting. In *Proceedings of the AAMAS-03 Workshop on Agent Mediated Electronic Commerce V: Designing Mechanisms and Systems*, volume 3048 of *Lecture Notes on Artificial Intelligence*.

Kambhampati, S. 1997. Refinement planning as a unifying framework for plan synthesis. *AI Magazine* 18(2):67–97.

Kim, H.-S., and Gratch, J. 2004. A planner-independent collaborative planning assistant. In *Proceedings of the Third International Conference on Autonomous Agents and Multi-Agent Systems*, 764–771.

Kleinmann, K.; Lazarus, R.; and Tomlinson, R. 2003. An infrastructure for adaptive control of multi-agent systems. In *IEEE Int. Conf. on Integration of Knowledge Intensive Multi-Agent Systems*, 230–236.

Korf, R. 1987. Planning as search: A quantitative approach. *Artificial Intelligence* 33(1):65–88.

Malone, T. W., and Crowston, K. 1994. The interdisciplinary study of coordination. *ACM Computing Surveys* 21(1):87–119.

Sandholm, T. W. 2002. Algorithm for optimal winner determination in combinatorial auctions. *Artificial Intelligence* 135(1–2):1–54.

Schillo, M.; Kray, C.; and Fischer, K. 2002. The eager bidder problem: A fundamental problem of DAI and selected solutions. In *Proceedings of the First International Conference on Autonomous Agents and Multi-Agent Systems*, 599–606. ACM Press.

Smith, R. G. 1980. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Transactions on Computers* C-29(12):1104–1113.

Tsamardinos, I.; Pollack, M. E.; and Horty, J. F. 2000. Merging plans with quantitative temporal constraints, temporally extended actions, and conditional branches. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems (AIPS-00)*, 264–272. Menlo Park, CA: AAAI Press.

van der Krogt, R., and de Weerdt, M. 2005. Plan repair as an extension of planning. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS-05)*.

Vickrey, W. 1961. Computer speculation, auctions, and competitive sealed tenders. *Journal of Finance* 16:8–37.

von Martial, F. 1992. *Coordinating Plans of Autonomous Agents*, volume 610 of *Lecture Notes on Artificial Intelligence*. Berlin: Springer Verlag.

Walsh, W. E., and Wellman, M. P. 1999. A market protocol for decentralized task allocation and scheduling with hierarchical dependencies. In *Proceedings of the Third International Conference on Multi-Agent Systems (ICMAS-98)*, 325–332. An extended version of this paper is also available.

Wilkins, D., and Myers, K. 1998. A multiagent planning architecture. In *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems (AIPS-98)*, 154–162. Menlo Park, CA: AAAI Press. Also available as a technical report.

Younes, H. L. S., and Simmons, R. G. 2003. VHPOP: Versatile heuristic partial order planner. *Journal of AI Research* 20:405–430.

Exploiting Interaction Structure in Networked Distributed POMDPs

R. Nair
Knowledge Systems Group
Honeywell Labs
Minneapolis MN 55418
ranjit.nair@honeywell.com

P. Varakantham and **M. Tambe**
Computer Science Dept.
University of Southern California
Los Angeles CA 90089
{varakant,tambe}@usc.edu

M. Yokoo
Dept. of Intelligent Systems
Kyushu University
Kyushu, Japan
yokoo@is.kyushu-u.ac.jp

Abstract

In many real-world multiagent applications such as distributed sensor nets, a network of agents is formed based on each agent's limited interactions with a small number of neighbors. While distributed POMDPs capture the real-world uncertainty in multiagent domains, they fail to exploit such locality of interaction. Distributed constraint optimization (DCOP) captures the locality of interaction but fails to capture planning under uncertainty. This paper present a new model synthesized from distributed POMDPs and DCOPs, called Networked Distributed POMDPs (ND-POMDPs). Exploiting network structure enables us to present two novel algorithms for ND-POMDPs: a distributed policy generation algorithm that performs local search and a systematic policy search that is guaranteed to reach the global optimal.

Introduction

Distributed Partially Observable Markov Decision Problems (Distributed POMDPs) are emerging as an important approach for multiagent teamwork. With distributed POMDPs, a central policy generator plans an optimal joint policy that maximizes the agents' expected joint reward under both action and observation uncertainty. Distributed POMDPs enable modeling more realistically the problems of a team's coordinated action under uncertainty (Nair *et al.* 2003; Montemerlo *et al.* 2004; Bernstein, Zilberstein, & Immerman 2000).

Unfortunately, as shown by Bernstein *et al.* (2000), the problem of finding the optimal joint policy for a general unrestricted distributed POMDP is NEXP-Complete. Researchers have hence attempted two different approaches to address this complexity. First, they have focused on algorithms that sacrifice global optimality and instead focus on local optimality (Nair *et al.* 2003; Peshkin *et al.* 2000). Second, they have focused on domains that require restricted types of interactions between two agents, e.g., transition independence or reward independence (Becker *et al.* 2003). While these approaches have led to useful advances, the complexity of the distributed POMDP problem has limited most experiments to a central policy generator planning for just two agents.

This paper introduces a third complementary approach called Networked Distributed POMDPs (ND-POMDPs). ND-POMDPs is a hybrid model that synergistically combines the local agent interactions of distributed constraint optimization (DCOP) (Modi *et al.* 2003; Yokoo & Hirayama 1996) with the planning under uncertainty in POMDPs. DCOPs have successfully exploited limited agent interactions in multiagent systems, with over a decade of algorithm development. Distributed POMDPs benefit by building upon such algorithms that enable distributed planning, and provide algorithmic guarantees. DCOPs benefit due to the significant enrichment to enable (distributed) planning under uncertainty — a key DCOP deficiency in practical applications such as sensor nets (Lesser, Ortiz, & Tambe 2003).

Indeed, the DCOP-POMDP synergy in ND-POMDPs leads to two novel algorithms. First, LID-JESP algorithm combines the existing JESP algorithm of Nair *et al.* (2003) and the *DBA* (Yokoo & Hirayama 1996) DCOP algorithm. LID-JESP thus combines the dynamic programming of JESP with two innovations: (i) distributed instead of JESP's centralized policy generation; (ii) LID-JESP guarantees termination in a local optimal, but also provides monotonicity for anytime performance. LID-JESP illustrates by example the beneficial synergies of DCOP and POMDPs in exploiting agent interaction graphs. Second, we present a more systematic policy search that is guaranteed to reach the global optimal on tree-structured agent-interaction graphs; and illustrate that by exploiting properties from constraint literature, it can guarantee optimality in general. Finally, we empirically compare the performance of the two algorithms with two benchmarks that do not exploit network structure. As these experiments show, we are able to solve larger problems by exploiting network structure of the interaction. Significantly, an increase in the number of agents keeping the maximum number of neighbors fixed leads to a linear increase in run time using the LID-JESP algorithm; while it leads to an exponential increase in run time for algorithms that don't consider network structure. Thus, approaches like LID-JESP appear well-suited for domains like sensor grids with a large number of agents, where each interacts with a small number of neighbors.

Domains, Motivation and Model

Our research is motivated by domains such as distributed sensor nets(Lesser, Ortiz, & Tambe 2003), distributed UAV

teams, and distributed satellites, where multiple agents must coordinate to accomplish a joint goal, but agents have a strong locality in their interactions. For example, with distributed sensor nets, multiple sensor agents must coordinate to track individual targets moving through an area. In particular, we consider in this paper a problem motivated by the real-world challenge in (Lesser, Ortiz, & Tambe 2003). Here, each sensor node can scan in one of four directions — North, South, East or West (see Figure 1), and to track a target, two sensors with overlapping scanning areas must coordinate by scanning the same area simultaneously. We assume that there are two independent targets and that each target's movement is uncertain and unaffected by the actions of the sensor agents. Additionally, each sensor receives observations only from the area it is scanning and this observation can have both false positives as well as false negatives. Further, each agent pays a cost for scanning whether the target is present or not. This cost is not incurred if the sensor chooses not to scan in any direction. As seen in this domain, each sensor interacts with only a limited number of neighboring sensor agents. For instance, sensors 1 and 3 do not share any scanning area, and have no effect on each other, except potentially via sensor 2. The sensors' observations and transitions are independent of each other's actions. Existing distributed POMDP algorithms, although rich enough to capture the uncertainties in this domain, are unlikely to work well for such a domain because they are not geared to take advantage of the locality of interaction. As a result they will have to consider all possible action choices of even non-interacting agents, in trying to solve the distributed POMDP. Distributed constraint satisfaction and distributed constraint optimization (DCOP) have been applied to sensor nets but these approaches cannot capture the uncertainty in the domain. Hence we introduce the networked distributed POMDP (ND-POMDP) model, a hybrid of POMDP and DCOP, that can handle the uncertainties in the domain as well as take advantage of locality of interaction.

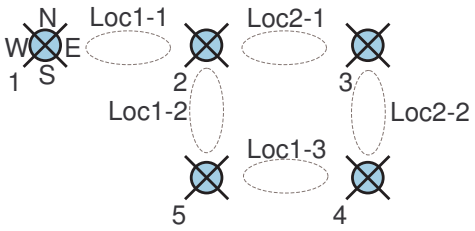


Figure 1: Sensor net scenario: If present, target1 is in Loc1-1, Loc1-2 or Loc1-3, and target2 is in Loc2-1 or Loc2-2.

ND-POMDPs

We define an ND-POMDP for a group Ag of n agents as a tuple $\langle S, A, P, \Omega, O, R\rangle$, where $S = \times_{1\leq i\leq n} S_i \times S_u$ is the set of world states. S_i refers to the set of local states of agent i and S_u is the set of unaffectable states. Unaffectable state refers to that part of the world state that cannot be affected by the actions of any of the agents and can refer to environmental factors like weather that no agent can control. $A = \times_{1\leq i\leq n} A_i$ is the set of joint actions, where $A_1, \ldots, A_n$, are the sets of action for agents 1 to n.

We assume a *transition independent* distributed POMDP model, where the transition function is defined as $P(s, a, s') = P_u(s_u, s'_u) \cdot \prod_{1\leq i\leq n} P_i(s_i, s_u, a_i, s'_i)$, where $s = \langle s_1, \ldots, s_n, s_u\rangle$, $s' = \langle s'_1, \ldots, s'_n, s'_u\rangle$ and $a = \langle a_1, \ldots, a_n\rangle$. The local transition function for agent i, is defined as $P_i(s_i, s_u, a_i, s'_i) = Pr(s'_i|s_i, s_u, a_i)$ and the unaffectable transition function is defined as $P_u(s_u, s'_u) = Pr(s'_u|s_u)$.

$\Omega = \times_{1\leq i\leq n} \Omega_i$ is the set of joint observations where Ω_i is the set of observations for agents i. In this paper, we assume that an agent's observations are independent of other agents' actions. Thus, we define the joint observation function as $O(s, a, \omega) = \prod_{1\leq i\leq n} O_i(s_i, s_u, a_i, \omega_i)$, where $s = \langle s_1, \ldots, s_n, s_u\rangle$, $a = \langle a_1, \ldots, a_n\rangle$ and $\omega = \langle \omega_1, \ldots, \omega_n\rangle$. The observation function for agent i is defined as $O(s_i, s_u, a_i, \omega_i) = Pr(\omega_i|s_1, s_u, a_i)$.

R refers to the reward function and is defined as $R(s, a) = \sum_l R_l(s_{l1}, \ldots, s_{lk}, s_u, a_{l1}, \ldots, a_{lk})$, where each l could refer to any sub-group of agents and $k = |l|$. Based on the reward function, we can construct an *interaction graph* where a link exists between a sub-group of agents, l, for every component R_l in the reward function. However, for simplicity we will assume that each R_l is for at most two agents.

Thus, we can define the *interaction graph* as $G = (Ag, E)$, where the vertices are the set of agents Ag and $E = \{(i, j)|R_{e_{ij}}$ is a component of $R\}$ refers to a set of undirected edges between agent i and j. Note that, in addition to binary rewards, we also allow local rewards. Thus the reward function is defined as: $R(s, a) = \sum_i R_i(s_i, s_u, a_i) + \sum_{e_{ij}\in E} R_{ij}(s_i, s_j, s_u, a_i, a_j)$, where we assume that $i < j$.

Based on the interaction graph, we define *neighborhood* of i as $N_i = \{j|\exists e_{ij} \in E\}$. We refer to the local states of the neighbors of i as $S_{N_i} = \times_{j\ s.t.\ e_{ij}\in E} S_j$. Similarly we also define A_{N_i}, Ω_{N_i}, P_{N_i} and O_{N_i}.

The goal in the ND-POMDP model is to come up with a joint policy $\pi = \langle \pi_i, \ldots, \pi_n\rangle$ that maximizes the expected reward of the team over a finite horizon T starting from an initial probability distribution b over states. π_i refers to the individual policy of agent i and is a mapping from the set of observation histories of i to the set of actions A_i. π_{N_i} refers to the joint policy of the agents in N_i.

Locality of Interaction

Given a factored reward function and the assumptions of transitional and observational independence, the resulting value function can be factored (Guestrin, Venkataraman, & Koller 2002) as well into value functions for each of the agents, V_i and each of the edges in the interaction graph, V_{ij}.

We define a *local neighborhood utility* as follows:

$$U^{N_i}(\pi) = V_i^{\pi_i}(s_i, s_u, \vec{\omega}_i) + \sum_{e_{ij}\in E} V_{ij}^{\langle \pi_i, \pi_j\rangle}(s_i, s_j, s_u, \vec{\omega}_i, \vec{\omega}_j) \quad (1)$$

which is the value returned by Algorithm 2.

Equation 1 sums over $j \in N_i$ only in the neighborhood of i, and hence any change of policies of agents not in the

neighborhood of i does not affect $U^{N_i}(\pi)$. Thus any such policy assignment, π' that has different policies for only non-neighborhood agents, has equal value as $U^{N_i}(\pi)$. Thus, while trying to find best policy for agent i given its neighbor's policy, we do not need to consider non-neighbor's policies. This is the property of *locality of interaction* that is used in Sections and .

Similarity to DCOP

The ND-POMDP can be thought of as a DCOP where each agent is a node. The variable at each node is the local policy of that agent and the domain of values is the set of possible individual policies. The reward component R_i can be thought of as a local constraint while the rewards R_{ij}, corresponding to edges in the *interaction graph*, are the binary constraints in a constraint graph. In the following section, we push this analogy further by taking inspiration from the DBA algorithm (Yokoo & Hirayama 1996), an algorithm for distributed constraint satisfaction, to develop an algorithm for solving ND-POMDPs.

Locally optimal policy generation

The locally optimal policy generation algorithm called LID-JESP (Locally interacting distributed joint equilibrium search for policies) is based on the DBA algorithm (Yokoo & Hirayama 1996) and JESP (Nair *et al.* 2003). In this algorithm (see Algorithm 1), each agent tries to improve its policy with respect to its neighbors' policies in a distributed manner similar to DBA. Initially each agent i starts with a random policy and exchanges its policies with its neighbors (lines 2-3). It then computes its local neighborhood utility (see Equation 1) from its initial belief state b with respect to its current policy and its neighbors' policy. Agent i then tries to improve upon its current policy by calling function GET-VALUE (see Algorithm 3), which returns the value of agent i's best response to its neighbors' policies. This algorithm is described in detail in Section . Agent i then computes the gain that it can make to its local neighborhood utility, and exchanges its gain with its neighbors (lines 7-10). If i's gain is greater than that of one any of its neighbor's[1], i changes its policy and sends its new policy to all its neighbors. This process of trying to improve the local policy is continued until termination. Termination detection is based on using a termination counter to count the number of cycles where $gain_i = 0$. If its gain is greater than zero the termination counter is reset. Agent i then exchanges its termination counter with its neighbors and sets its counter to the minimum of its counter and its neighbors' counters. Agent i will terminate if its termination counter becomes equal to the diameter of the interaction graph.

Finding Best Response

The algorithm for computing the best response is a dynamic-programming approach similar to that used in JESP. Here, we define an *episode* of agent i at time t as $e_i^t = \langle s_u^t, s_i^t, s_{N_i}^t, \vec{\omega}_{N_i}^t \rangle$. Given that the neighbors' policies are fixed, treating episode as the state, results in a single agent POMDP, where the transition function and observation function can be defined as follows:

$$P'(e_i^t, a_i^t, e_i^{t+1}) = P_u(s_u^t, s_u^{t+1}) \cdot P_i(s_i^t, s_u^t, a_i, s_i^{t+1}) \cdot P_{N_i}(s_{N_i}^t, s_u^t, a_{N_i}, s_{N_i}^{t+1}) \cdot O_{N_i}(s_{N_i}^{t+1}, s_u^{t+1}, a_{N_i}, \omega_{N_i})$$

$$O'(e_i^{t+1}, a_i^t, \omega_i^{t+1}) = O_i(s_i^{t+1}, s_u^{t+1}, a_i, \omega_i)$$

A multiagent belief state for an agent i given the distribution over the initial state, $b(s)$ is defined as:

$$B(e_i^t) = Pr(s_u^t, s_i^t, s_{N_i}^t, \vec{\omega}_{N_i}^t | \vec{\omega}_i^t, \vec{a}_i^{t-1}, b) \quad (2)$$

We can now compute the best response using the following equation (see Algorithm 3):

$$V^t(B_i^t, b) = \max_{a_i \in A_i} V^{a_i,t}(B_i^t, b) \quad (3)$$

The function, $V^{a_i,t}$, can be computed using Algorithm 4 as follows:

$$V^{a_i,t}(B_i^t, b) = \sum_{e_i^t} B_i^t(e_i^t) \cdot \left(R\left(s^t, \langle a_i, \pi_{N_i}(\vec{\omega}_{N_i}) \rangle\right) + \sum_{\omega_1^{t+1} \in \Omega_1} \Pr(\omega_i^{t+1} | B_i^t, a_i) \cdot V_i^{t+1}\left(B_i^{t+1}\right)\right) \quad (4)$$

B_i^{t+1} is the belief state updated after performing action a_i and observing ω_i^{t+1} and is computed using Algorithm 5.

Algorithm 1 LID-JESP(*Agent i*)

1: $d \leftarrow$ diameter of graph, $terminationCtr_i \leftarrow 0$
2: $\pi_i \leftarrow$ randomly selected policy, $prevVal \leftarrow 0$
3: Exchange π_i with N_i
4: **while** $terminationCtr_i < d$ **do**
5: **for all** s_i, s_{N_i}, s_u **do**
6: $prevVal \stackrel{+}{\leftarrow} Pr(s_i, s_{N_i}, s_u|b) \cdot$ EVALUATE($Agent\ i, s_i, s_u, s_{N_i}, \pi_i, \pi_{N_i}, \langle\rangle, \langle\rangle, 0, T$)
7: $gain_i \leftarrow$ GETVALUE($Agent\ i, b, \pi_{N_i}, 0, T$) $- prevVal$
8: **if** $gain_i > 0$ **then** $terminationCtr_i \leftarrow 0$
9: **else** $terminationCtr_i \stackrel{+}{\leftarrow} 1$
10: Exchange $gain_i$,$terminationCtr_i$ with N_i
11: $terminationCtr_i \leftarrow \mathbf{min}_{j \in N_i \cup \{i\}} terminationCtr_j$
12: $maxGain \leftarrow \mathbf{max}_{j \in N_i \cup \{i\}} gain_j$
13: $winner \leftarrow \mathbf{argmax}_{j \in N_i \cup \{i\}} gain_j$
14: **if** $maxGain > 0$ **and** $i = winner$ **then**
15: initialize π_i
16: FINDPOLICY($Agent\ i, b, \langle\rangle, \pi_{N_i}, 0, T$)
17: Communicate π_i with N_i
18: **else if** $maxGain > 0$ **then**
19: Receive π_{winner} from $winner$ and update π_{N_i}
20: **return** π_i

Theoretical Results

Proposition 1 *If $terminationCtr_i =$ diameter of graph, then agents are in a local optimum.*

[1]The function $\mathbf{argmax}_j$ disambiguates between multiple j corresponding to the same max value by returning the lowest j.

Algorithm 2 EVALUATE($Agent\, i, s_i^t, s_u^t, s_{N_i}^t, \pi_i, \pi_{N_i}, \vec{\omega}_i, \vec{\omega}_{N_i}, t, T$)

1: $a_i \leftarrow \pi_i(\vec{\omega}_i)$, $a_{N_i} \leftarrow \pi_{N_i}(\vec{\omega}_{N_i})$
2: $val \leftarrow R_i(s_i^t, a_i) + \sum_{e_{ij}} R_{ij}\left(s_i^t, s_j^t, s_u^t, a_i, a_j\right)$
3: **if** $t < T-1$ **then**
4: **for all** $s_i^{t+1}, s_{N_i}^{t+1}, s_u^{t+1}$ **do**
5: **for all** ω_i, ω_{N_i} **do**
6: $val \overset{+}{\leftarrow} P_u(s_u^t, s_u^{t+1}) \cdot P_i(s_i^t, s_u^t, a_i, s_i^{t+1}) \cdot P_{N_i}(s_{N_i}^t, s_u^t, a_{N_i}, s_{N_i}^{t+1}) \cdot O_i(s_i^{t+1}, s_u^{t+1}, a_i, \omega_i) \cdot O_{N_i}(s_{N_i}^{t+1}, s_u^{t+1}, a_{N_i}, \omega_{N_i}) \cdot$ EVALUATE($Agent\ i, s_i^{t+1}, s_u^{t+1}, s_{N_i}^{t+1}, \pi_i, \pi_{N_i}, \langle\vec{\omega}_i, \omega_i\rangle, \langle\vec{\omega}_{N_i}, \omega_{N_i}\rangle, t+1, T$)
7: **return** val

Algorithm 3 GETVALUE($Agent\, i, B^t, \pi_{N_i}, t, T$)

1: **if** $t \geq T$ **then return** 0
2: **if** $V_t(B^t)$ is already recorded **then return** $V_t(B^t)$
3: $best \leftarrow -\infty$
4: **for all** $a_i \in A_i$ **do**
5: $value \leftarrow$ GETVALUEACTION($Agent\, i, B^t, a_i, \pi_{N_i}, t, T$)
6: record $V_t^{a_i}(B^t)$ as value
7: **if** $value > best$ **then** $best \leftarrow value$
8: record $V_t(B^t)$ as $best$
9: **return** $best$

Algorithm 4 GETVALUEACTION($Agent\, i, B^t, a, \pi_{N_i}, t, T$)

1: $value \leftarrow 0$
2: **for all** $e^t = \left\langle s_u^t, s_i^t, s_{N_i}^t, \vec{\omega}_{N_i}\right\rangle$ s.t. $B^t(e^t) > 0$ **do**
3: $a_{N_i} \leftarrow \pi_{N_i}(\vec{\omega}_{N_i})$, $reward \leftarrow R_i(s_i^t, a_i) + \sum_{e_{ij}} R_{ij}\left(s_i^t, s_j^t, s_u^t, a_i, a_j\right)$
4: $value \overset{+}{\leftarrow} B^t(e^t) \cdot reward$
5: **if** $t < T-1$ **then**
6: **for all** $\omega_i \in \Omega_i$ **do**
7: $B^{t+1} \leftarrow$ UPDATE($Agent\, i, B^t, a, \omega_i, \pi_{N_i}$)
8: $prob \leftarrow 0$
9: **for all** $s_u^t, s_i^t, s_{N_i}^t$ **do**
10: **for all** $e^{t+1} = \left\langle s_u^{t+1}, s_i^{t+1}, s_{N_i}^{t+1}, \langle\vec{\omega}_{N_i}, \omega_{N_i}\rangle\right\rangle$ s.t. $B^{t+1}(e^{t+1}) > 0$ **do**
11: $a_{N_i} \leftarrow \pi_{N_i}(\vec{\omega}_{N_i})$
12: $prob \overset{+}{\leftarrow} B^t(e^t) \cdot P_u(s_u^t, s_u^{t+1}) \cdot P_i(s_i^t, s_u^t, a_i, s_i^{t+1}) \cdot P_{N_i}(s_{N_i}^t, s_u^t, a_{N_i}, s_{N_i}^{t+1}) \cdot O_i(s_i^{t+1}, s_u^{t+1}, a_i, \omega_i) \cdot O_{N_i}(s_{N_i}^{t+1}, s_u^{t+1}, a_{N_i}, \omega_{N_i})$
13: $value \overset{+}{\leftarrow} prob \cdot$ GETVALUE($Agent\ i, B^{t+1}, \pi_{N_i}, t+1, T$)
14: **return** $value$

Algorithm 5 UPDATE($Agent\, i, B^t, a_i, \omega_i, \pi_{N_i}$)

1: **for all** e^{t+1} **do**
2: $B^{t+1}(e^{t+1}) \leftarrow 0$, $a_{N_i} \leftarrow \pi_{N_i}(\vec{\omega}_{N_i})$
3: **for all** $s^t \in S$ **do**
4: $B^{t+1}(e^{t+1}) \overset{+}{\leftarrow} B^t(e^t) \cdot P_u(s_u^t, s_u^{t+1}) \cdot P_i(s_i^t, s_u^t, a_i, s_i^{t+1}) \cdot P_{N_i}(s_{N_i}^t, s_u^t, a_{N_i}, s_{N_i}^{t+1}) \cdot O_i(s_i^{t+1}, s_u^{t+1}, a_i, \omega_i) \cdot O_{N_i}(s_{N_i}^{t+1}, s_u^{t+1}, a_{N_i}, \omega_{N_i})$
5: normalize B^{t+1}
6: **return** B^{t+1}

Algorithm 6 FINDPOLICY($Agent\, i, B^t, \vec{\omega}_i, \pi_{N_i}, t, T$)

1: $a^* \leftarrow \arg\max_{a_i} V_{a_i}^t(B^t)$, $\pi_i(\vec{\omega}_i) \leftarrow a^*$
2: **if** $t < T-1$ **then**
3: **for all** $\omega_i \in \Omega_i$ **do**
4: $B^{t+1} \leftarrow$ UPDATE($Agent\, i, B^t, a^*, \omega_i, \pi_{N_i}$)
5: FINDPOLICY($Agent\, i, B^{t+1}, \langle\vec{\omega}_i, \omega_i\rangle, \pi_{N_i}, t+1, T$)
6: **return**

Proof: Assume that in cycle c, $terminationCtr_i =$ diameter but agents are not in a local optimum.

In cycle $c -$ diameter, there must be at least one agent j who can improve, i.e., $gain_j \neq 0$ (otherwise, agents are in a local optimum in cycle $c -$ diameter and no agent can improve later).

Let the distance between agents i and j be d_{ij}. Then, in cycle $c -$ diameter $+ d_{ij}$ (which is less than or equal to c), $terminationCtr_i$ becomes 0. However, $terminationCtr_i$ increases at most by one for each cycle. Thus, in cycle c, $terminationCtr_i \leq$ diameter $- d_{ij}$.

If $d_{ij} \geq 1$, in cycle c, $terminationCtr_i <$ diameter. Also, if $d_{ij} = 0$, i.e., in cycle $c -$ diameter, $gain_i \neq 0$, then in cycle $c -$ diameter $+ 1$, $terminationCtr_i = 0$, thus, in cylce c, $terminationCtr_i <$ diameter. In either case, the assumption that $terminationCtr_i =$ diameter cannot hold. □

Proposition 2 *If agents reach a local optimum, $terminationCtr_i$ becomes d within d cycles, where $d =$ diameter.*

Proof: Once agents reach a local optimum, for each agent i $gain_i = 0$ henceforth. Thus, $terminationCtr_i$ is never reset to 0 and is incremented by 1 in every cycle. Thus, after $d =$ diameter cycles, $terminationCtr_i$ becomes diameter. □

Based on the above proofs, we can conclude the LID-JESP is guaranteed to terminate if and only if the agents reach a local optimum.

Proposition 3 *When applying LID-JESP, the global utility is strictly increasing with each cycle until local optimum is attained.*

Proof sketch By construction of LID-JESP, neighboring agents cannot modify their policies in the same cycle. Agent i chooses to change its policy if it can improve upon its local neighborhood utility U^{N_i}. From Equation 1, increasing U^{N_i} results in an increase in global utility. Based on locality of interaction, if a non-neighbor j changes its policy, it will not affect U^{N_i} but will increase U^{N_j}. Thus with each cycle global utility is strictly increasing until local optimum is reached. □.

Global Optimal Algorithm (GOA)

The global optimal algorithm exploits network structure in finding the optimal policy for a distributed POMDP. It requires a tree structure (interaction graph constructed based

on the reward structure) among the agents. However this does not impair its application to other kinds of interaction graphs, because any interaction graph with cycles can be converted to a tree using cycle cutset algorithms.

All child agents find their best policy (policy with highest expected reward) for a fixed parent policy and return the expected reward associated with their best policy. The value of a parent policy is the sum of the best policy rewards obtained from the children. A parent node goes through all its possible policies, and changes its best policy if it gets higher value than its current best policy. This process is repeated at each level in the tree, until the root exhausts all its policies. The above method helps GOA take advantage of the structure and prune unnecessary joint policy evaluations.

Algorithm 7 provides the pseudo code for the Global Optimal algorithm at each agent. Lines 1-4 set the best policy(π_i^*) at an agent and its children when terminate is received. Lines 10-12 calculate the value of policy, π_i for a given policy, π_j of its parent (in the interaction graph). Lines 13-20 record the best value for a policy when all children provide best responses. Lines 21-23 store the best value and policy obtained thus far for a given a parent policy. Lines 23-25 are for the root agent to signal the termination of the algorithm.

Algorithm 7 GO-JOINTPOLICY($Agent\, i, \pi_j, terminate$)

1: **if** $terminate =$ yes **then**
2: $\pi_i^* \leftarrow bestResponse\{\pi_j\}$
3: **for all** $Agent\ k \in children_i$ **do**
4: GO-JOINTPOLICY(k, π_i^*,yes)
5: $\Pi_i \leftarrow$ ENUMERATE($Agent\ i, A_i, O_i, T$)
6: $bestVal \leftarrow$ -∞
7: $j \leftarrow$ parent(i)
8: **for all** $\pi_i \in \Pi_i$ **do**
9: $jointVal \leftarrow 0$
10: **if** $i \neq$ root **then**
11: **for all** s_i, s_j, s_u **do**
12: $jointVal \stackrel{+}{\leftarrow} Pr(s_i, s_j, s_u|b) \cdot$ EVALUATE($Agent\ i, s_i, s_u, s_j, \pi_i, \pi_j, \langle\rangle, \langle\rangle, 0, T$)
13: **if** $policyValMap\{\pi_i\} \neq$ null **then**
14: $jointVal \stackrel{+}{\leftarrow} policyValMap\{\pi_i\}$
15: **else**
16: $childVal \leftarrow 0$
17: **for all** $Agent\ k \in children_i$ **do**
18: $childVal \stackrel{+}{\leftarrow}$ GO-JOINTPOLICY(k, π_i,no)
19: $policyValMap\{\pi_i\} \leftarrow childVal$
20: $jointVal \stackrel{+}{\leftarrow} childVal$
21: **if** $jointVal > bestVal$ **then**
22: $bestVal \leftarrow jointVal$
23: $\pi_i^* \leftarrow \pi_i$
24: **if** $i = root$ **then**
25: **for all** $Agent\ k \in children_i$ **do**
26: GO-JOINTPOLICY(k, π_i^*,yes)
27: $bestResponse\{\pi_j\} = \pi_i^*$
28: **return** $bestVal$

Experimental Results

For our experiments, we use the sensor domain from Section (see Figure 1). We consider three different sensor configurations of increasing complexity. The first configuration is a chain with 3 agents (sensors 1-3). Here target1 is either absent or located at Loc1-1. Similarly, target2 is either absent or at Loc2-1. Each agent can perform either turnOff, scanEast or scanWest. Each agent receives an observation targetPresent or targetAbsent depending on the unaffectable state and its last action. The second configuration is a 4 agent chain (sensors 1-4). In this configuration target2 can be in either absent or in Loc2-1 or Loc2-2, giving rise to 6 unaffectable states. The number of individual actions and observations are unchanged. The 3rd configuration is the 5 agent F configuration and is identical to Figure 1. In this configuration target1 can also be located at Loc1-2 and Loc1-3 giving rise to 12 unaffectable states. We include an additional action for each agent called scanVert that allows the agent to scan North and South.

For each of these scenarios, we ran the LID-JESP algorithm from Section . For our first benchmark (JESP), we used Nair *et al.*'s JESP algorithm (2003). This algorithm uses a centralized processor to find a locally optimal joint policy and does not consider the network structure of the interaction. For our second benchmark (LID-JESP-no-nw), we ran the LID-JESP with a fully connected *interaction graph*. For the 3 and 4 agent chains, we also ran the GOA algorithm (from Section) and the GOA algorithm with a completely connected interaction graph (GOA-no-nw).

Figures 2- 4 compare the performance of the various algorithms for the 3 agent chain, 4 agent chain and 5 agent F configurations respectively. Each of the graphs on the left of these figures shows the run time[2] in seconds on a logscale on the Y-axis for increasing finite horizon T on the X-axis, while the graphs on the right show the value of policy found on the Y-axis and increasing finite horizon T on the X-axis. The run times and values for LID-JESP, JESP and LID-JESP-no-nw are each the average obtained from 5 runs, each with different randomly chosen starting policies . However, for a particular run, the various algorithms use the same starting policies. As can be seen in the graphs on the right of Figures 2- 4, the values obtained for LID-JESP, JESP and LID-JESP-no-nw are quite similar, although LID-JESP and LID-JESP-no-nw often converged on a higher local optima than JESP. On average the value obtained using LID-JESP will be less than that obtained by GOA. Random restarts can be used to try and converge at a higher local optima. Please note that GOA and GOA-no-nw are both exact algorithms and will hence return the same value. In comparing the run times, please note that GOA does significantly better than GOA-no-nw, which could not be run for $T > 2$ and $T > 1$ for the 3 and 4 agent chains, respectively, within 10,000 seconds. GOA, itself, could not be run for $T > 3$ and $T > 2$ within 10,000 seconds for the 3 and 4 agent chains respectively. All three locally optimal algorithms show a significant improvement over GOA in terms of run time. However, it should be noted that LID-JESP significantly outperforms LID-JESP-no-nw and JESP by exploiting *locality of interaction*.

[2]Machine specs for all experiments: Intel Xeon 2.8 GHz processor, 2GB RAM, Linux Redhat 8.1

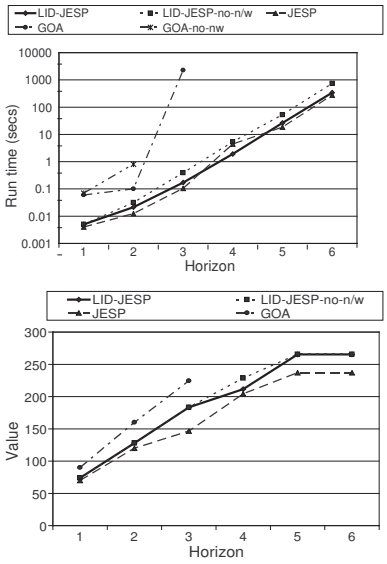


Figure 2: 3 agent chain.(a) Run time(secs),(b) Value

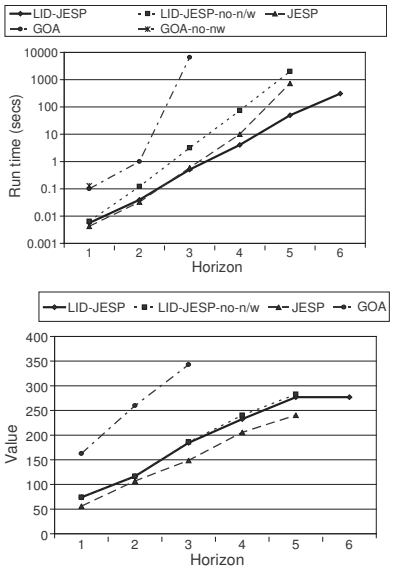


Figure 3: 4 agent chain. (a) Run time (secs), (b) Value

Comparing Figures 2(a) and 3(a) shows that an increase in the number of agents, keeping the maximum number of neighbors fixed, leads to a linear increase in run time using the LID-JESP algorithm; while it leads to an exponential increase in run time for algorithms that don't consider network structure. It is important to note that all the experiments were run on a single processor machine. We would expect LID-JESP to out-perform centralized algorithms like JESP even more on multi-processor machines owing to its distributedness.

Summary and Related Work

In a large class of applications, such as distributed sensor nets, distributed UAVs and satellites, a large network of agents is formed from each agent's limited interactions with a small number of neighboring agents. We exploit such network structure to present a new distributed POMDP model called ND-POMDP. We present two distributed algorithms for ND-POMDPs that exploit network structure: a dynamic-programming algorithm that performs local search and a more systematic policy search that is guaranteed to reach the global optimal. Experimental results illustrate the significant efficiency gains of the two algorithms when compared with previous algorithms that are unable to exploit such structure.

Among related work, we have earlier discussed the relationship of our work to key DCOP and distributed POMDP algorithms, i.e., that we synthesize new algorithms by exploiting their synergies. We now discuss some other recent algorithms for locally and globally optimal policy generation for distributed POMDPs. For instance, Hansen *et al.* (2004) present an exact algorithm for partially observable stochastic games (POSGs) based on dynamic programming and iterated elimination of dominant policies. Emery-Montemerlo (2004) approximate POSGs as a series of one-step Bayesian games using heuristics to find the future discounted value for actions. We have earlier discussed Nair *et al.* (2003)'s JESP algorithm that uses dynamic programming to reach a local optimal. In addition, Becker *et al.*'s work (2003) on transition-independent distributed MDPs is related to our assumptions about transition and observability independence in ND-POMDPs. These are all centralized policy generation algorithms that could benefit from the key ideas in this paper — that of exploiting local interaction structure among agents to (i) enable distributed policy generation; (ii) limit policy generation complexity by considering only interactions with "neighboring" agents. Guestrin *et al.* (2002), present "coordination graphs" which have similarities to constraint graphs. The key difference in their approach is that the "coordination graph" is obtained from the value function which is computed in a centralized manner. The agents then use a distributed procedure for online action selection based on the coordination graph. In our approach, the value function is computed in a distributed manner. Dolgov and Durfee (2004) also study the effect of network structure on multiagent MDPs. However, their algorithm assumed that each agent tried to optimize its individual utility instead of the team's utility.

Acknowledgments

This material is based upon work supported by the DARPA/IPTO COORDINATORs program and the Air Force Research Laboratoryunder Contract No. FA8750–05–C–0030.The views and conclusions contained in this document are those of the authors, and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

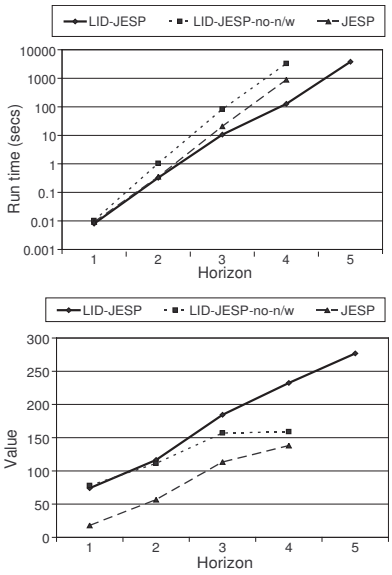


Figure 4: 5 agent F-config. (a) Run time (secs), (b) Value

References

Becker, R.; Zilberstein, S.; Lesser, V.; and Goldman, C. V. 2003. Transition-independent decentralized Markov decision processes. In *AAMAS*.

Bernstein, D. S.; Zilberstein, S.; and Immerman, N. 2000. The complexity of decentralized control of MDPs. In *UAI*.

Dolgov, D., and Durfee, E. 2004. Graphical models in local, asymmetric multi-agent markov decision processes. In *AAMAS*.

Guestrin, C.; Venkataraman, S.; and Koller, D. 2002. Context specific multiagent coordination and planning with factored MDPs. In *AAAI*.

Hansen, E.; Bernstein, D.; and Zilberstein, S. 2004. Dynamic Programming for Partially Observable Stochastic Games. In *AAAI*.

Lesser, V.; Ortiz, C.; and Tambe, M. 2003. *Distributed sensor nets: A multiagent perspective*. Kluwer academic publishers.

Modi, P. J.; Shen, W.; Tambe, M.; and Yokoo, M. 2003. An asynchronous complete method for distributed constraint optimization. In *AAMAS*.

Montemerlo, R. E.; Gordon, G.; Schneider, J.; and Thrun, S. 2004. Approximate solutions for partially observable stochastic games with common payoffs. In *AAMAS*.

Nair, R.; Pynadath, D.; Yokoo, M.; Tambe, M.; and Marsella, S. 2003. Taming decentralized POMDPs: Towards efficient policy computation for multiagent settings. In *IJCAI*.

Peshkin, L.; Meuleau, N.; Kim, K.-E.; and Kaelbling, L. 2000. Learning to cooperate via policy search. In *UAI*.

Yokoo, M., and Hirayama, K. 1996. Distributed breakout algorithm for solving distributed constraint satisfaction problems. In *ICMAS*. MIT Press.

Bounded Policy Iteration for Decentralized POMDPs

Daniel S. Bernstein
Dept. of Computer Science
University of Massachusetts
Amherst, MA 01003
bern@cs.umass.edu

Eric A. Hansen
Dept. of CS and Engineering
Mississippi State University
Mississippi State, MS 39762
hansen@cse.msstate.edu

Shlomo Zilberstein
Dept. of Computer Science
University of Massachusetts
Amherst, MA 01003
shlomo@cs.umass.edu

Abstract

We present a bounded policy iteration algorithm for infinite-horizon decentralized POMDPs. Policies are represented as joint stochastic finite-state controllers, which consist of a local controller for each agent. We also let a joint controller include a correlation device that allows the agents to correlate their behavior without exchanging information during execution, and show that this leads to improved performance. The algorithm uses a fixed amount of memory, and each iteration is guaranteed to produce a controller with value at least as high as the previous one for all possible initial state distributions. For the case of a single agent, the algorithm reduces to Poupart and Boutilier's bounded policy iteration for POMDPs.

1 Introduction

The Markov decision process (MDP) framework has proven to be useful for solving problems of sequential decision making under uncertainty. For some problems, an agent must base its decision on partial information about the system state. In this case, it is often better to use the more general partially observable Markov decision process (POMDP) framework. Though POMDPs are difficult to solve in the worst case, much progress has been made in the development of practical dynamic programming algorithms [Smallwood and Sondik, 1973; Cassandra *et al.*, 1997; Hansen, 1998; Poupart and Boutilier, 2003; Feng and Zilberstein, 2004].

Even more general are problems in which a team of decision makers, each with its own local observations, must act together. Domains in which these types of problems arise include networking, multi-robot coordination, e-commerce, and space exploration systems. To model such problems, we can use the decentralized partially observable Markov decision process (DEC-POMDP) framework. Though this model has been recognized for decades (see, e.g., [Witsenhausen, 1971]), there has been little work on efficient algorithms for it.

Recently, an exact dynamic programming algorithm was proposed for DEC-POMDPs [Hansen *et al.*, 2004]. Though the algorithm was presented in the context of finite-horizon problems, there are various ways to extend it to the infinite-horizon case. However, in both cases, it suffers from the fact that the memory requirements grow quickly with each iteration, and in practice it has only been used to solve very small problems. It is likely that any optimal algorithm would suffer this problem, as finite-horizon DEC-POMDPs have been shown to be NEXP-complete, even for just two agents [Bernstein *et al.*, 2002].

In this paper, we present a memory-bounded dynamic programming algorithm for infinite-horizon DEC-POMDPs. The algorithm uses a stochastic finite-state controller to represent the joint policy for the agents. A straightforward approach is to use a set of independent local controllers, one for each agent. We provide an example to illustrate that higher value can be obtained through the use of shared randomness. As such, we define a joint controller to be a set of local controllers along with a *correlation device*. The correlation device is a finite-state machine that sends a signal to all of the agents on each time step. Its behavior can be determined prior to execution time, and thus it does not require that the agents exchange information after receiving local observations.

Our algorithm generalizes *bounded policy iteration* for POMDPs [Poupart and Boutilier, 2003] to the multi-agent case. On each iteration, a node is chosen from one of the local controllers or the correlation device, and its parameters are updated through the solution of a linear program. The generalization has the same theoretical guarantees as in the POMDP case. Namely, an iteration is guaranteed to produce a new controller with value at least as high for every possible initial state distribution.

In our experiments, we applied our algorithm to idealized networking and robot navigation problems. Both problems are too large for exact dynamic programming, but could be handled by our approximation algorithm. We found that the addition of a correlation device gives rise to better solutions. In addition, larger controllers most often lead to better solutions.

A number of approximation algorithms have been developed previously for DEC-POMDPs [Peshkin *et al.*, 2000; Nair *et al.*, 2003; Emery-Montemerlo *et al.*, 2004]. However, the previous algorithms do not guarantee both bounded memory usage and monotonic value improvement for all initial state distributions. Furthermore, the use of correlated stochastic policies in the DEC-POMDP context is novel. The

importance of correlation has been recognized in the game theory community [Aumann, 1974], but there has been little work on algorithms for finding correlated policies.

2 Background

In this section, we present our formal framework for multi-agent decision making. A *decentralized partially-observable Markov decision process (DEC-POMDP)* is a tuple $\langle I, S, \{A_i\}, \{O_i\}, P, R\rangle$, where

- I is a finite set of agents indexed $1, \ldots, n$
- S is a finite set of states
- A_i is a finite set of actions available to agent i and $\vec{A} = \times_{i \in I} A_i$ is the set of joint actions, where $\vec{a} = \langle a_1, \ldots, a_n\rangle$ denotes a joint action
- O_i is a finite set of observations for agent i and $\vec{O} = \times_{i \in I} O_i$ is the set of joint observations, where $\vec{o} = \langle o_1, \ldots, o_n\rangle$ denotes a joint observation
- P is a set of Markovian state transition and observation probabilities, where $P(s', \vec{o}|s, \vec{a})$ denotes the probability that taking joint action $\vec{a}$ in state s results in a transition to state s' and joint observation $\vec{o}$
- $R : S \times \vec{A} \rightarrow \Re$ is a reward function

In this paper, we consider the case in which the process unfolds over an infinite sequence of stages. At each stage, all agents simultaneously select an action, and each receives the global reward and a local observation. The objective of the agents is to maximize the expected discounted sum of rewards received. We denote the discount factor γ and require that $0 \leq \gamma < 1$.

3 Finite-State Controllers

Our algorithm uses stochastic finite-state controllers to represent policies. In this section, we first define a type of controller in which the agents act independently. We then provide an example demonstrating the utility of correlation, and show how to extend the definition of a joint controller to allow for correlation among agents.

3.1 Local Finite-State Controllers

In a DEC-POMDP, each agent must select an action based on its history of local observations. Finite-state controllers provide a way to represent local policies using a finite amount of memory. The state of the controller is based on the observation sequence, and the agent's actions are based on the state of its controller. We allow for stochastic transitions and stochastic action selection, as this can help to make up for limited memory. This type of controller has been used previously in the single-agent context [Platzman, 1980; Meuleau *et al.*, 1999; Poupart and Boutilier, 2003].

Formally, we define a *local finite-state controller* for agent i to be a tuple $\langle Q_i, \psi_i, \eta_i\rangle$, where Q_i is a finite set of controller nodes, $\psi_i : Q_i \rightarrow \Delta A_i$ is an action selection function, and $\eta_i : Q_i \times A_i \times O_i \rightarrow \Delta Q_i$ is a transition function. The functions ψ_i and η_i parameterize the conditional distribution $P(a_i, q_i'|q_i, o_i)$.

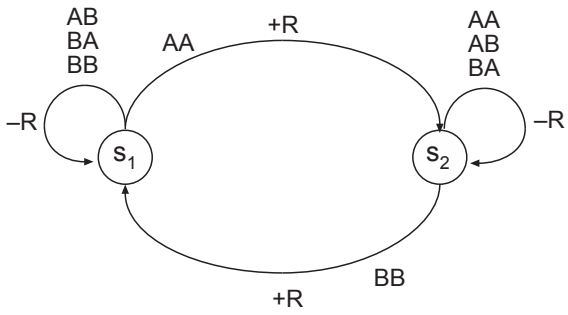


Figure 1: This figure shows a DEC-POMDP for which the optimal memoryless joint policy requires correlation.

Taken together, the agents' controllers determine the conditional distribution $P(\vec{a}, \vec{q}'|\vec{q}, \vec{o})$. This is denoted an *independent joint controller*. In the following subsection, we show that independence can be limiting.

3.2 The Utility of Correlation

The joint controllers described above do not allow the agents to correlate their behavior via a shared source of randomness. We will use a simple example to illustrate the utility of correlation in partially observable domains where agents have limited memory. This example generalizes the one given in [Singh *et al.*, 1994] to illustrate the utility of stochastic policies in single-agent partially observable settings.

Consider the DEC-POMDP shown in Figure 1. This problem has two states, two agents, and two actions per agent (A and B). The agents each have only one observation, and thus cannot distinguish between the two states. For this example, we will consider only memoryless policies.

Suppose that the agents can independently randomize their behavior using distributions $P(a_1)$ and $P(a_2)$, and consider the policy in which each agent chooses either A or B according to a uniform distribution. This yields an expected reward of $-\frac{R}{2}$ per time step, which results in an expected long-term reward of $\frac{-R}{2(1-\gamma)}$. It is straightforward to show that no independent policy yields higher reward than this one for all states.

Next, let us consider the larger class of policies in which the agents may act in a correlated fashion. In other words, we consider all joint distributions $P(a_1, a_2)$. Consider the policy that assigns probability $\frac{1}{2}$ to the pair AA and probability $\frac{1}{2}$ to the pair BB. This yields an average reward of 0 at each time step and thus an expected long-term reward of 0. The difference between the rewards obtained by the independent and correlated policies can be made arbitrarily large by increasing R.

3.3 Correlated Joint Controllers

In the previous subsection, we established that correlation can be useful in the face of limited memory. In this subsection, we extend our definition of a joint controller to allow for correlation among the agents. To do this, we introduce an additional finite-state machine, called a correlation device, that provides

Variables: $\epsilon, x(c, a_i), x(c, a_i, o_i, q_i')$
Objective: Maximize ϵ
Improvement constraints:

$$\forall s, q_{-i}, c \quad V(s, \vec{q}, c) + \epsilon \leq \sum_{\vec{a}} P(a_{-i}|c, q_{-i})[x(c, a_i)R(s, \vec{a}) + \gamma \sum_{s', \vec{o}, \vec{q}', c'} x(c, a_i, o_i, q_i')P(q_{-i}'|c, q_{-i}, a_{-i}, o_{-i})P(s', \vec{o}|s, \vec{a})P(c'|c)V(s', \vec{q}', c')]$$

Probability constraints:

$$\forall c \quad \sum_{a_i} x(c, a_i) = 1, \quad \forall c, a_i, o_i \quad \sum_{q_i'} x(c, a_i, o_i, q_i') = x(c, a_i)$$

$$\forall c, a_i \quad x(c, a_i) \geq 0, \quad \forall c, a_i, o_i, q_i' \quad x(c, a_i, o_i, q_i') \geq 0$$

Table 1: The linear program used to find new parameters for agent i's node q_i. The variable $x(c, a_i)$ represents $P(a_i|q_i, c)$, and the variable $x(c, a_i, o_i, q_i')$ represents $P(a_i, q_i'|c, q_i, o_i)$.

extra signals to the agents at each time step. The device operates independently of the DEC-POMDP process, and thus does not provide the agents with information about the other agents' observations. In fact, the random numbers necessary for its operation could be determined prior to execution time.

Formally, a *correlation device* is a tuple $\langle C, \psi \rangle$, where C is a set of states and $\psi : C \to \Delta C$ is a state transition function. At each step, the device undergoes a transition, and each agent observes its state.

We must modify the definition of a local controller to take the state of the correlation device as input. Now, a local controller for agent i is a conditional distribution of the form $P(a_i, q_i'|c, q_i, o_i)$. The correlation device together with the local controllers form a joint conditional distribution $P(c', \vec{a}, \vec{q}'|c, \vec{q}, \vec{o})$. We will refer to this as a *correlated joint controller*. Note that a correlated joint controller with $|C| = 1$ is effectively an independent joint controller.

The value function for a correlated joint controller can be computed by solving the following system of linear equations, one for each $s \in S$, $\vec{q} \in \vec{Q}$, and $c \in C$:

$$V(s, \vec{q}, c) = \sum_{\vec{a}} P(\vec{a}|c, \vec{q})[R(s, \vec{a}) + \gamma \sum_{s', \vec{o}, \vec{q}', c'} P(s', \vec{o}|s, \vec{a})P(\vec{q}'|c, \vec{q}, \vec{a}, \vec{o}) \cdot P(c'|c)V(s', \vec{q}', c')].$$

We sometimes refer to the value of the controller for an initial state distribution. For a distribution δ, this is defined as

$$V(\delta) = \max_{\vec{q}, c} \sum_{s} \delta(s)V(s, \vec{q}, c).$$

It is assumed that, given an initial state distribution, the controller is started in the joint node which maximizes value from that distribution.

4 Bounded Policy Iteration

We now describe our bounded policy iteration algorithm for improving correlated joint controllers. To improve a correlated joint controller, we can either change the correlation device or one of the local controllers. Both improvements can be done via a *bounded backup*, which involves solving a linear program. Following an improvement, the controller can be reevaluated through the solution of a set of linear equations. Below, we describe how a bounded backup works, and prove that it always produces a new controller with value at least as high for all initial state distributions.

4.1 Improving a Local Controller

We first describe how to improve a local controller. To do this, we choose an agent i, along with a node q_i. Then, we search for new parameters for the conditional distribution $P(a_i, q_i'|c, q_i, o_i)$.

The search for new parameters works as follows. We assume that the original controller will be used from the second step on, and try to replace the parameters for q_i with better ones for just the first step. In other words, we look for the best parameters satisfying the following inequality:

$$V(s, \vec{q}, c) \leq \sum_{\vec{a}} P(\vec{a}|c, \vec{q})[R(s, a) + \gamma \sum_{s', \vec{o}, \vec{q}', c'} P(\vec{q}'|c, \vec{q}, \vec{a}, \vec{o})P(s', \vec{o}|s, \vec{a}) \cdot P(c'|c)V(s', \vec{q}', c)]$$

for all $s \in S$, $q_{-i} \in Q_{-i}$, and $c \in C$. Note that the inequality is always satisfied by the original parameters. However, it is often possible to get an improvement.

Finding new parameters can be done using linear programming, as shown in Table 1. We note that this linear program is the same as that of Poupart and Boutilier [2003] for POMDPs, with the nodes of the other local controllers and correlation device considered part of the hidden state. Its size is polynomial in the sizes of the DEC-POMDP and the joint controller, but exponential in the number of agents.

4.2 Improving the Correlation Device

The procedure for improving the correlation device is very similar to the procedure for improving a local controller. We

Variables: $\epsilon, x(c')$
Objective: Maximize ϵ
Improvement constraints:

$$\forall s, \vec{q} \quad V(s, \vec{q}, c) + \epsilon \quad \leq \quad \sum_{\vec{a}} P(\vec{a}|c, \vec{q})[R(s, \vec{a}) + \gamma \sum_{s', \vec{o}, \vec{q}', c'} P(\vec{q}'|c, \vec{q}, \vec{a}, \vec{o}) P(s', \vec{o}|s, \vec{a}) x(c') V(s', \vec{q}', c')]$$

Probability constraints:

$$\forall c' \quad \sum_{c'} x(c') = 1, \quad \forall c' \quad x(c') \geq 0$$

Table 2: The linear program used to find new parameters for the correlation device node c. The variable $x(c')$ represents $P(c'|c)$.

first choose a device node c, and consider changing its parameters for just the first step. We look for the best parameters satisfying the following inequality:

$$V(s, \vec{q}, c) \quad \leq \quad \sum_{\vec{a}} P(\vec{a}|c, \vec{q})[R(s, a) + \gamma \sum_{s', \vec{o}, \vec{q}', c} P(\vec{q}'|c, \vec{q}, \vec{a}, \vec{o}) P(s', \vec{o}|s, \vec{a}) \cdot P(c'|c) V(s', \vec{q}', c')]$$

for all $s \in S$ and $\vec{q} \in \vec{Q}$.

As in the previous case, the search for parameters can be formulated as a linear program. This is shown in Table 2. This linear program is also polynomial in the sizes of the DEC-POMDP and joint controller, but exponential in the number of agents.

4.3 Monotonic Improvement

We have the following theorem, which says that performing either of the two updates cannot lead to a decrease in value for any initial state distribution.

Theorem 1 *Performing a bounded backup on a local controller or the correlation device produces a correlated joint controller with value at least as high for every initial state distribution.*

Proof. Consider the case in which some node q_i of agent i's local controller is changed. Let V_o be the value function for the original controller, and let V_n be the value function for the new controller. Recall that the new parameters for $P(a_i, q_i'|c, q_i, o_i)$ must satisfy the following inequality for all $s \in S$, $q_{-i} \in Q_{-i}$, and $c \in C$:

$$V_o(s, \vec{q}, c) \quad \leq \quad \sum_{\vec{a}} P(\vec{a}|c, \vec{q})[R(s, a) + \gamma \sum_{s', \vec{o}, \vec{q}', c'} P(\vec{q}'|c, \vec{q}, \vec{a}, \vec{o}) P(s', \vec{o}|s, \vec{a}) \cdot P(c'|c) V_o(s', \vec{q}', c)]$$

Notice that the formula on the right is the Bellman operator for the new controller, applied to the old value function. Denoting this operator T_n, the system of inequalities implies that $T_n V_o \geq V_o$. By monotonicity, we have that for all $k \geq 0$, $T_n^{k+1}(V_o) \geq T_n^k(V_o)$. Since $V_n = \lim_{k \to \infty} T_n^k(V_o)$, we have that $V_n \geq V_o$. Thus, the value of the new controller is higher than that of the original controller for all possible initial state distributions.

The argument for changing nodes of the correlation device is almost identical to the one given above. □

4.4 Local Optima

Although bounded backups give nondecreasing values for all initial state distributions, convergence to optimality is not guaranteed. There are a couple of factors contributing to this. First is the fact that only one local controller, or the correlation device, is improved at once. Thus, it is possible for the algorithm to get stuck in a suboptimal Nash equilibrium in which each of the controllers and the correlation device is optimal with the others held fixed. It is an open problem whether there is a linear program for updating more than one controller at a time.

Of course, a bounded backup does not find the *optimal* parameters for one controller with the others held fixed. Thus, a sequence of such updates may converge to a local optimum without even reaching a Nash equilibrium. For POMDPs, Poupart and Boutilier [2003] provide a characterization of these local optima, and a heuristic for escaping from them. This could be applied in our case, but it would not address the suboptimal Nash equilibrium problem.

5 Experiments

We implemented bounded policy iteration and tested it on two different problems, an idealized networking scenario and a problem of navigating on a grid. Below, we describe our experimental methodology, the specifics of the problems, and our results.

5.1 Experimental Setup

Although our algorithm guarantees nondecreasing value for all initial state distributions, we chose a specific distribution to focus on for each problem. Experiments with different distributions yielded qualitatively similar results.

We define a *trial run* of the algorithm as follows. At the start of a trial run, a size is chosen for each of the local controllers and the correlation device. The action selection and transition functions are initialized to be deterministic, with

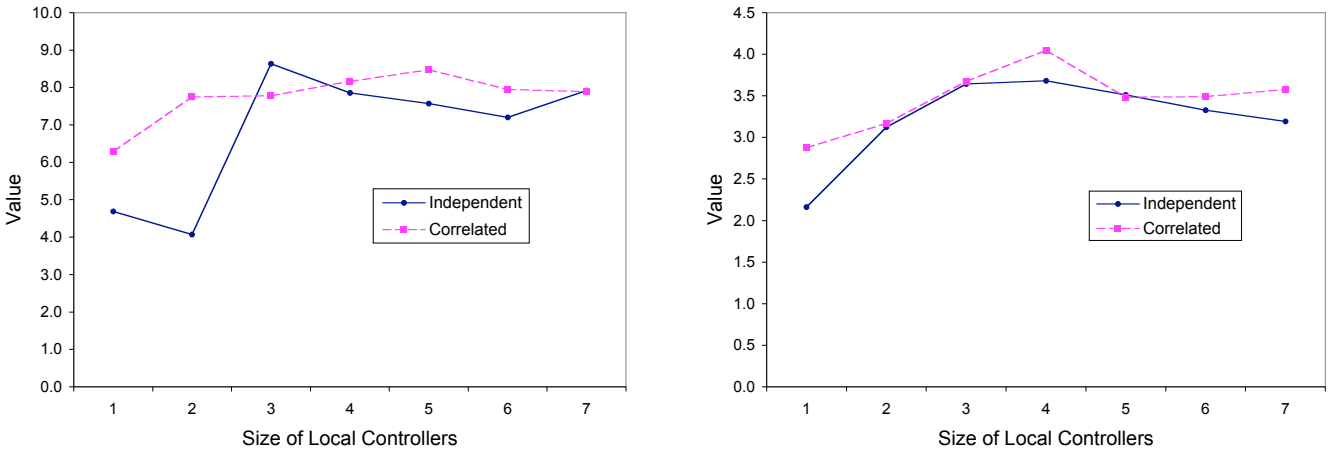


Figure 2: Average value per trial run plotted against the size of the local controllers, for (a) the multi-access broadcast channel problem, and (b) the robot navigation problem. The solid line represents independent controllers (a correlation device with one node), and the dotted line represents a joint controller including a two-node correlation device.

the outcomes drawn according to a uniform distribution. A *step* consists of choosing a node uniformly at random from the correlation device or one of the local controllers, and performing a bounded backup on that node. After 50 steps, the run is considered over. In practice, we found that values usually stabilized within 15 steps.

We varied the sizes of the local controllers from 1 to 7 (the agents' controllers were always the same sizes as each other), and we varied the size of the correlation device from 1 to 2. Thus, the number of joint nodes ranged from 1 to 98. Memory limitations prevented us from using larger controllers. For each combination of sizes, we performed 20 trial runs. We recorded the highest value obtained across all runs, as well as the average value over all runs.

5.2 Multi-Access Broadcast Channel

Our first domain is an idealized model of control of a multi-access broadcast channel [Ooi and Wornell, 1996]. In this problem, nodes need to broadcast messages to each other over a channel. Only one node may broadcast at a time, otherwise a collision occurs. The nodes share the common goal of maximizing the throughput of the channel.

At the start of each time step, each node decides whether or not to send a message. The nodes receive a reward of 1 when a message is successfully broadcast and a reward of 0 otherwise. At the end of the time step, each node observes its own buffer, and whether the previous step contained a collision, a successful broadcast, or nothing attempted.

The message buffer for each agent has space for only one message. If a node is unable to broadcast a message, the message remains in the buffer for the next time step. If a node i is able to send its message, the probability that its buffer will fill up on the next step is p_i. Our problem has two nodes, with $p_1 = 0.9$ and $p_2 = 0.1$. There are 4 states, 2 actions per agent, and 5 observations per agent. The discount factor is 0.9. The start state distribution is deterministic, with the buffer for agent 1 containing a message and the buffer for agent 2 being empty.

5.3 Meeting on a Grid

In this problem, we have two robots navigating on a two-by-two grid with no obstacles. Each robot can only sense whether there are walls to its left or right, and the goal is for the robots to spend as much time as possible on the same square. The actions are to move up, down, left, or right, or to stay on the same square. When a robot attempts to move to an open square, it only goes in the intended direction with probability 0.6, otherwise it either goes in another direction or stays in the same square. Any move into a wall results in staying in the same square. The robots do not interfere with each other and cannot sense each other.

This problem has 16 states, since each robot can be in any of 4 squares at any time. Each robot has 4 observations, since it has a bit for sensing a wall to its left or right. The total number of actions for each agent is 5. The reward is 1 when the agents share a square, and 0 otherwise, and the discount factor is 0.9. The initial state distribution is deterministic, placing both robots in the upper left corner of the grid.

5.4 Results

For each combination of controller sizes, we looked at the best solutions found across all trial runs. The values for these solutions were the same for all controller sizes except for the few smallest.

It was more instructive to compare average values over all trial runs. Figure 2 shows graphs of average values plotted against controller size. We found that, for the most part, the average value increases when we increase the size of the correlation device from one node to two nodes (essentially moving from independent to correlated).

For small controllers, the average value tends to increase with controller size. However, as the controllers get larger,

there is no clear trend. This behavior is somewhat intuitive, given the way the algorithm works. For new node parameters to be acceptable, they must not decrease the value for any combination of states, nodes for the other controllers, and nodes for the correlation device. This becomes more difficult as controllers get larger, and thus it is easier to get stuck in a local optimum.

Improving multiple controllers at once would help to alleviate the aforementioned problem. As mentioned earlier, we do not currently have a way to do this using linear programming, and it thus remains an interesting topic for future work.

6 Conclusion and Future Work

We have presented a bounded policy iteration algorithm for DEC-POMDPs. Besides the fact that it uses finite memory, the algorithm has a number of other appealing theoretical guarantees. First, by using correlated joint controllers, we can achieve higher value than with independent joint controllers of the same size. Second, assuming a constant number of agents, each iteration of the algorithm completes in polynomial time. Finally, monotonic value improvement is guaranteed for all states on each iteration.

Our empirical results are encouraging. By bounding the size of the controller, we are able to achieve a tradeoff between computational complexity and the quality of the approximation. Up to a point, increasing the sizes of the local controllers leads to higher values on average. After this point, average values tend to level off or decrease. Increasing the size of the correlation device leads to higher value, which is consistent with our theoretical results.

For future work, there are many more experiments that can be done with bounded policy iteration. For instance, in moving to a larger controller, we could use the previous controller as a starting point, rather than starting over with a random controller. Poupart and Boutilier's [2003] escape technique could be useful here. Also, rather than choosing nodes uniformly at random for updating, we could develop a principled way to order the nodes.

We are also looking into ways of extending the algorithm to handle problems with large numbers of agents. In many problems, each agent interacts with only a small subset of the other agents. This additional structure can be exploited to reduce the size of the problem representation, and it should be possible to extend our algorithm to take advantage of these local interactions.

Finally, it would be interesting to extend bounded policy iteration to the noncooperative setting, where each agent has a separate reward function. One approach is to require that a change in parameters does not lead to a decrease in value for *any* agent. Another approach is to consider just the value function for the agent whose node is being updated. This should move the joint controller towards a Nash equilibrium.

7 Acknowledgments

We thank Martin Allen and Özgür Şimşek for helpful discussions of this work. This work was supported in part by the National Science Foundation under grants IIS-0219606 and IIS-9984952, by NASA under cooperative agreement NCC 2-1311, and by the Air Force Office of Scientific Research under grant F49620-03-1-0090.

References

[Aumann, 1974] Robert J. Aumann. Subjectivity and correlation in randomized strategies. *Journal of Mathematical Economics*, 1:67–96, 1974.

[Bernstein *et al.*, 2002] Daniel S. Bernstein, Robert Givan, Neil Immerman, and Shlomo Zilberstein. The complexity of decentralized control of Markov decision processes. *Mathematics of Operations Research*, 27(4):819–840, 2002.

[Cassandra *et al.*, 1997] Anthony Cassandra, Michael L. Littman, and Nevin L. Zhang. Incremental pruning: A simple, fast, exact method for partially observable Markov decision processes. In *Proceedings of UAI*, pages 54–61, 1997.

[Emery-Montemerlo *et al.*, 2004] Rosemary Emery-Montemerlo, Geoff Gordon, Jeff Schnieder, and Sebastian Thrun. Approximate solutions for partially observable stochastic games with common payoffs. In *Proceedings of AAMAS*, 2004.

[Feng and Zilberstein, 2004] Zhengzhu Feng and Shlomo Zilberstein. Region-Based incremental pruning for POMDPs. In *Proceedings of UAI*, pages 146–153, 2004.

[Hansen *et al.*, 2004] Eric A. Hansen, Daniel S. Bernstein, and Shlomo Zilberstein. Dynamic programming for partially observable stochastic games. In *Proceedings of AAAI*, pages 709–715, 2004.

[Hansen, 1998] Eric Hansen. Solving POMDPs by searching in policy space. In *Proceedings of UAI*, pages 211–219, 1998.

[Meuleau *et al.*, 1999] Nicolas Meuleau, Kee-Eung Kim, Leslie Kaelbling, and Anthony R. Cassandra. Solving POMDPs by searching the space of finite policies. In *Proceedings of UAI*, pages 417–426, 1999.

[Nair *et al.*, 2003] Ranjit Nair, David Pynadath, Makoto Yokoo, Milind Tambe, and Stacy Marsella. Taming decentralized POMDPs: Towards efficient policy computation for multiagent settings. In *Proceedings of IJCAI*, 2003.

[Ooi and Wornell, 1996] James M. Ooi and Gregory W. Wornell. Decentralized control of a multiple access broadcast channel: Performance bounds. In *Proceedings of the 35th Conference on Decision and Control*, pages 293–298, 1996.

[Peshkin *et al.*, 2000] Leonid Peshkin, Kee-Eung Kim, Nicolas Meuleau, and Leslie Pack Kaelbling. Learning to cooperate via policy search. In *Proceedings of UAI*, pages 489–496, 2000.

[Platzman, 1980] Loren K. Platzman. A feasible computational approach to infinite-horizon partially-observed Markov decision processes. Technical report, Georgia Institute of Technology, 1980. Reprinted in *Working Notes of the 1998 AAAI Fall Symposium on Planning Using Partially Observable Markov Decision Processes*.

[Poupart and Boutilier, 2003] Pascal Poupart and Craig Boutilier. Bounded finite state controllers. In *Proceedings of NIPS*, 2003.

[Singh *et al.*, 1994] Satinder P. Singh, Tommi Jaakkola, and Michael I. Jordan. Learning without state-estimation in partially observable markovian decision processes. In *Proceedings of ICML*, 1994.

[Smallwood and Sondik, 1973] Richard D. Smallwood and Edward J. Sondik. The optimal control of partially observable Markov processes over a finite horizon. *Operations Research*, 21(5):1071–1088, 1973.

[Witsenhausen, 1971] Hans S. Witsenhausen. Separation of estimation and control for discrete time systems. *Proceedings of the IEEE*, 59(11):1557–1566, 1971.

ASET: a Multi-Agent Planning Language with Nondeterministic Durative Tasks for BDD-Based Fault Tolerant Planning*

Rune M. Jensen and **Manuela M. Veloso**
Computer Science Department, Carnegie Mellon University,
Pittsburgh, PA 15213-3891, USA

Abstract

In this paper, we introduce a multi-agent planning language called ASynchronous Evolving Tasks (ASET). The main contribution of ASET is a novel explicit representation of temporally extended tasks that may be nondeterministic both with respect to duration and effects. Moreover, ASET explicitly models the environment as a set of uncontrollable agents. We formally define ASET descriptions and their transformation to a nondeterministic planning domain. Using a Boolean encoding, fault tolerant planning problems specified in ASET can be solved efficiently with state-of-the-art BDD-based planning systems. Our preliminary experimental results show that the transformation of ASET domains to nondeterministic planning domains is computationally efficient even for ASET descriptions with a high level of temporal detail.

Introduction

The most important obstacle for widespread application of automated planning is lack of scalability. Since the complexity of planning grows with the representational power of the planning language, a good strategy for solving a planning problem efficiently is to use a planning language that is sufficient for representing the problem at hand but among such languages has least representational power.

For this reason, the goal for planning language developers is to expose the representational power of the language by providing intuitive and explicit ways to state abstract real-world phenomena. In addition, well designed high-level languages makes it possible to write short and elegant descriptions of a domain. They further improve the ability of planning systems to exploit structure in domains.

Today powerful planners exist for the STRIPS planning language e.g., (Hoffmann & Nebel 2001). But STRIPS assumes a single agent executing instantaneous and deterministic actions, while most real domains involve multiple asynchronous agents executing temporally extended stochastic actions. There is no simple way of modeling stochastic behavior of actions and multi-agent domains in STRIPS. Its representational power is too low.

A wide range of planning languages have been developed to address the deficiencies of STRIPS including temporal languages e.g., (Fox & Long 2003; Bacchus & Ady 2001; Laborie & Ghallab 1995), nondeterministic languages e.g., (Piergiorgio *et al.* 2002; Giunchiglia, Kartha, & Lifschitz 1997; Jensen & Veloso 2000) and probabilistic languages e.g., (Younes 2003). None of them, however, have simple explicit ways of describing domains that combine all the aspects of real-world domains mentioned above. In particular, we are not aware of any planning language with a single unified construct to define actions that are nondeterministic both with respect to effect and duration. Temporal planning languages have deterministic actions and nondeterministic planning languages do not consider durative actions.

The representational power of some of these languages e.g., (Younes 2003; Musliner, Durfee, & Shin 1993) and classical representations like discrete event systems, timed automata, and Markov Decision Processes (MDPs) is strong enough to model such domains. But it is often tedious and error prone to define domains in these formalisms due to the implicit representation of abstract phenomena. Furthermore, the representational power may be so high that the planning problems become unnecessarily hard to solve.

The research reported in this paper investigates how low we can go in representational power and still be able to define a language in which stochastic durative actions and multi-agent domains can be stated in a unified, intuitive, and explicit way. More specifically, we consider a language with the representation power of a nondeterministic planning domain (i.e., an MDP with no transition probabilities). Our motivation is that stationary policies for nondeterministic planning problems can be synthesized efficiently (Cimatti *et al.* 2003; Jensen, Veloso, & Bryant 2003) using techniques developed in formal verification based on Binary Decision Diagrams (BDDs) (Bryant 1986; Burch, Clarke, & McMillan 1990).

Continuous time and probabilistic models are attractive, but come with a high computational fee. It is well-known that continuous time verification of asynchronous circuits is much harder than discrete time verification of synchronized circuits, and even though efficient symbolic techniques ex-

*This research is sponsored by BBNT Solutions LLC under its prime contract number FA8760-04-C-0002 with the U.S. Air Force and DARPA. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the sponsoring institutions, the U.S. Government or any other entity.

ist for solving MDPs (Hansen & Zilberstein 2001), it is our experience that nonprobabilistic versions of these problems have orders of magnitude lower complexity.

We are interested in high-level planing problems where the goal is to coordinate low-level activities and manage shared resources. Such domains are often combinatorial and discrete in nature. Imagine an automated job shop floor with robots moving objects between machines and storage buffers. Commands to machines and robots are high-level, but fairly accurate models exist of the behavior they trigger. The main problem is to deliver and remove objects from machines in a temporally coordinated manner and share resources such as space.

Our language is based on an action representation called Evolving Tasks (ETs). ETs are Directed Acyclic Graphs (DAGs) of guarded unit time transitions that define the temporal behavior of the task. They can represent temporally extended activities which are nondeterministic both with respect to duration and effect. We consider multi-agent planning domains where each agent is defined by the set of ETs it can execute. The resulting language is called ASynchronous Evolving Tasks (ASET). Like NADL (Jensen & Veloso 2000), ASET explicitly model the environment as a set of uncontrollable agents.

The low-level semantics of an ASET domain is a *unit time transition graph*. The domain, however, is not controllable at this level since tasks are uninterruptible. A *decision graph* is derived from the unit time transition graph by adding transitions between all states where some task is idle and removing all other states from the unit time transition graph. This can be done efficiently using a technique called *iterative squaring* (Burch, Clarke, & McMillan 1990). The decision graph is a nondeterministic planning domain that allows us to define solutions to ASET planning problems as strong, strong cyclic, and weak plans (Cimatti *et al.* 2003). These plans can be efficiently generated by state-of-the-art symbolic nondeterministic planning systems (Cimatti *et al.* 2003; Jensen, Veloso, & Bryant 2003).

Using this bottom-up approach, it is easy to define low-level temporal properties of the activities, but plan in a more abstract space. PDDL2.1 and other temporal languages extending STRIPS are based on a top-down approach where many features are used to define the temporal properties of actions. The result is less general languages with temporal semantics of actions that can be hard to understand. The unit time semantics of ETs further solve a general problem of augmenting first order logic with time for temporal planning (Bacchus & Ady 2001; Fox & Long 2003). This often leads to information "holes" caused by concurrent actions hiding the state of domain knowledge they are currently changing. This makes it hard to write domains with mutually dependent asynchronous activities.

The main limitation of ASET is the lack of transition probabilities. Often, however, stochastic behavior is caused by infrequent system failures. This allows us to avoid full-blown probabilistic planning and instead consider nondeterministic plans robust to a limited number of system failures (Jensen, Veloso, & Bryant 2004). Another limitation is that ASET assumes full observability. But this is a reasonable assumption for systems that are engineered to be highly controllable.

We have implemented a BDD-based planning system for ASET. Preliminary experimental results show that a unit time transition graph can be efficiently transformed into decision graph even when the duration of tasks is in the order of 500 time units. This level of temporal granularity is more than sufficient for most applications.

The remainder of the paper is organized as follows. We first define ASET descriptions and discuss how they relate to other planning domain representations. We then present the unit time transition graph of an ASET description and its Boolean encoding and show how to transform the unit time transition graph into a decision graph. The following section briefly reminds about the definition of strong nondeterministic plans and shows how to represent a fault tolerant planning domain in ASET. We then present our experimental results and finally draw conclusions and discuss plans for future work.

ASET Descriptions

An ASET description consists of a disjoint set of system and environment *state variables* with finite domains, and a description of *system* and *environment agents*.

The state variables can be *metric* with finite integer domains, *Boolean*, or *enumerations* with finite domains. The usual arithmetic and relational operations can be carried out on metric variables. The set of state variable assignments defines the state space of the world.

An agent's description is a set of *tasks*. The agents change the state of the world by executing tasks. Each agent is always in a state of activity executing some task. The agents are asynchronous, they may start and stop tasks at different time-points. The system agents model the behavior of the agents controllable by the planner, while the environment agents model the uncontrollable world. To ensure independence of the system and environment agents, they affect a disjoint set of state variables. Their tasks, however, may depend on the complete state of the world.

A task has two parts: a set of *state variables* that the task modifies and a set of *unit time transitions* that defines how the task evolves. Intuitively, the task is responsible for assigning new values to the variables it modifies. It further has exclusive access to the modified variables, no other concurrent task can modify these variables as long as it is active. Each agent is associated with a finite set of execution states. These states are shared between the tasks of the agent and define the transition states of the tasks. Each set of execution states has a special *idle state*. Each transition of a task has unit time duration. The outgoing transitions from the idle state are taken when a task starts. The incoming transitions to the idle state are taken when the task stops. The remaining transitions of a task form a DAG on the execution states causing all execution paths of the task to be finite. Each transition is *guarded*. The guard is an expression on the complete state. This may include the current task and execution state of any agent as well as the current value of any state variable. The transition is only enabled if the guard

expression is satisfied. This allows rich behavior models including strong synchronization schemes with tasks of other agents. The effect of the transition is given as an expression on the state variables it modifies and its execution state. If this expression holds for several assignments, one of these is nondeterministically chosen as the effect of the transition. In this way, tasks are nondeterministic both with respect to duration and effect on modified variables. Notice that there is no need for an explicit precondition. The precondition of a task is the disjunction of the guards of outgoing transitions from the idle state.

Time advances in discrete integer time points. In each unit time step, the currently active tasks perform a unit time transition. Variables not modified by any task maintain their value. The resulting unit time transition graph will *block* if no transition is enabled for some task.

As an example consider the simple job shop domain shown in Figure 1. The domain has four locations a, b, c, and d connected with corridors ab, bd, cd, and ac. The goal is to paint the object (O). It can be carried by the robot (R) to the painting machine (P). The robot spends time navigating between corridors and may have to backtrack to its source location. The robot and painting machine are controllable, but there is also an uncontrollable human operator (H). For security reasons, the robot is not allowed to load and unload the object when the human is at the same location.

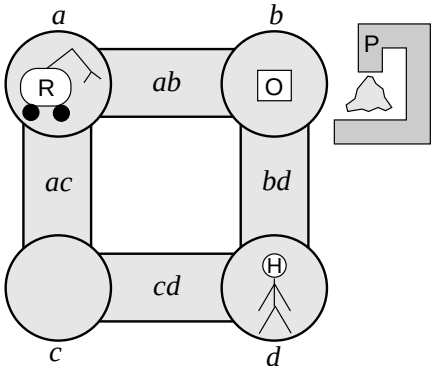


Figure 1: The job shop domain.

Figure 2 shows an ASET description of the job shop domain. Each task is a DAG where vertices are execution states and edges are unit time transitions. The idle execution state is marked by a double circle. Execution states are labeled by numbers (by convention idle states are labeled by zero, but these labels have been omitted to enhance readability). The guard expression or precondition of a unit time transition is shown above the associated edge. The effect of the transition is shown below the edge. The ASET description has two controllable system agents the robot (R) and the painting machine (P). It also has a single uncontrollable environment agent which is the human operator (H). The tasks of the robot are $drive(x, y)$[1], $take$, put, and $wait$. During the drive task, the robot navigates between the locations via the

[1]The figure shows $drive(a, b)$, but there are 7 other drive tasks.

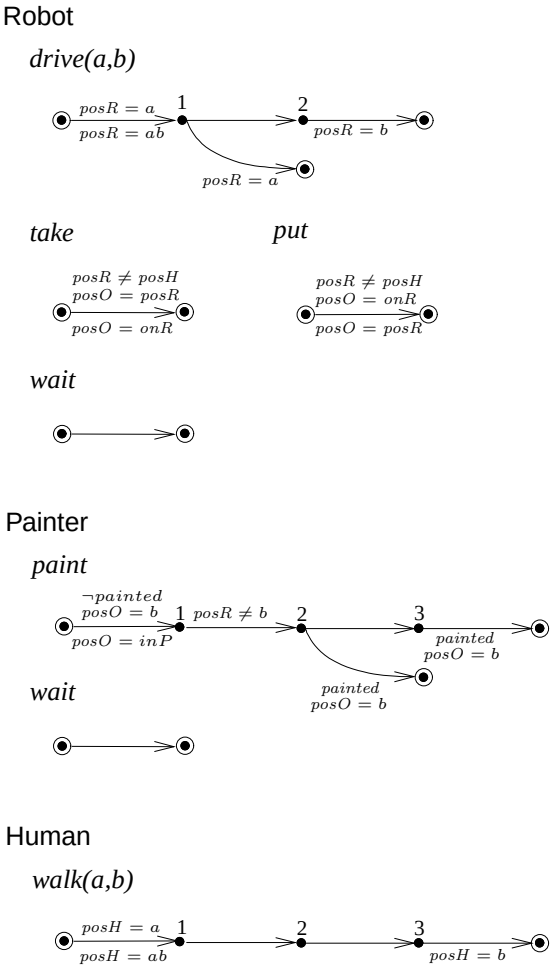


Figure 2: An ASET description of the job shop domain.

corridors. It may succeed after 2 time units and reach its destination, or fail after 3 time units in which case, it returns to the source location. The take and put tasks loads and unloads the object on the robot. They take one time unit and are conditioned by the human being at another location. The wait task also takes one time unit. It does not change any state variables, but only advances time to coordinate the robots activities with other agents. The tasks of the painter are $paint$ and $wait$. The paint task takes either 3 or 4 time units and requires that the object is at location b and is unpainted. Moreover, the robot must avoid location b when the actual painting happens. The wait task of the painter is identical to the wait task of the robot. The human has walk tasks similar to the robot's drive tasks except that the walk tasks are deterministic and have a duration of 4 time units. Since there is no wait task, the human must continuously walk between locations. This guarantees that the robot eventually can load and unload the object.

Formally, an ASET description is defined as follows.

Definition 1 (ASET Description) *An ASET description is a triple $\mathcal{M} = \langle V, E, T \rangle$, where*

- V is a finite domain of n^s *system state variables* and n^e *environment state variables* $V = V^s \times V^e$ where $V^x = \prod_{i=1}^{n^x} V_i^x$ for $x \in \{s, e\}$,
- E is a finite *execution space* of $m^s > 0$ *system agents* and $m^e \geq 0$ *environment agents* each associated with a set of execution states $E = E^s \times E^e$ where $E^x = \prod_{i=1}^{m^x} E_i^x$ for $x \in \{s, e\}$. Each set of execution states includes a special idle state $E_i^x \supseteq \{\text{idle}\}$ for $x \in \{s, e\}$ and $i = 1 \,..\, m^x$, and
- T is a finite *task space* of a non-empty set of tasks associated with each agent $T = \prod_{i=1}^{m^s} T_i^s \times \prod_{i=1}^{m^e} T_i^e$. Each task $t \in T_k^x$ is a pair $\langle M_t^x, R_t^x \rangle$, where
  - M_t^x is a set of indices of state variables modified by the task $M_t^x \subseteq \{1, \ldots, n^x\}$, and
  - R_t^x is a set of guarded unit time execution transitions of the task defining how modified variables are changed while the task is active $R_t^x \subseteq V \times E \times T \times \prod_{i \in M_t^x} V_i^x \times E_k^x$.

Compared with the durative action descriptions of PDDL2.1, TLplan, and IxTeT (Fox & Long 2003; Bacchus & Ady 2001; Laborie & Ghallab 1995), the most significant difference of ASET descriptions is that tasks are durative *and* nondeterministic. None of the above domain descriptions consider nondeterministic actions. Actually, we are not aware of any planning language with temporally extended and nondeterministic actions. Another important difference between ASET and the domain descriptions above is the use of state variables. This provides metric values, but so has PDDL2.1. What is probably more important is that our state variables are defined at every time point like state variables in physics and control theory (Cassandras & Lafortune 1999). When augmenting first order logic with time and preserving the precondition and effect notions from classical planning, domain knowledge may only exist at certain time points. An important exception from this, however, are the continuous durative actions of PDDL2.1. For these actions, update functions are provided to define the change of metric information. This approach, however, is not as general as ETs.

Another challenge for durative actions in the classical precondition-effect format is how to handle conditional effects. The problem is that conditional effects require information to be transfered from the state the action is being applied in, to the state the action is completed in. These states, however, may not be adjacent in the planning domain. The problem can be solved by introducing memory propositions (Fox & Long 2003) or instantaneous effects of actions (Bacchus & Ady 2001). For ETs the problem is solved explicitly, since conditional effects can be defined for each unit time transition as shown in the job shop example.

An important issue to address when introducing concurrent tasks is synergetic effects between simultaneously executing tasks (Lingard & Richards 1998). A common example of destructive synergetic effects is when two or more tasks require exclusive use of a single resource or when two tasks have inconsistent effects like $pos' = 3$ and $pos' = 2$.

Like actions in NADL, ASET tasks cannot be performed concurrently in the following two conditions: 1) they have inconsistent effects, or 2) they modify an overlapping set of state variables. The first condition is due to the fact that state knowledge is expressed in a monotonic logic that cannot represent inconsistent knowledge. The second condition addresses the problem of sharing resources. Consider for example two agents trying to eat the same ice cream. If only the first condition defined interfering tasks, both agents could simultaneously eat the ice cream, as the effect $iceCreamEaten$ of the two tasks would be consistent. With the second condition added, these tasks are interfering and cannot be performed concurrently.

We have chosen this definition of task interference due to our positive experience with it in NADL. There are, however, several issues to address. First, we need to show how to encode synergetic activity strong enough to solve Gelfond's soup problem (Gelfond, Lifschitz, & Rabinov 1991). The problem is to lift a soup bowl without spilling the soup. Two actions, lift left and lift right, can be applied to the bowl. If either is applied on its own the soup will spill, but if they are applied simultaneously then the bowl is raised from the table and no soup spills. The problem is that we cannot model the state of the soup bowl in ASET using just one state variable, since two concurrent lift tasks then would be unable to access that state variable. We can, however, represent such synergetic activity by letting the state of the bowl being expressed by several state variables. If we introduce two Boolean variables $force\_left$ and $force\_right$ the different states of the bowl can be represented by

$$\begin{aligned} onGround &= \neg force\_left \wedge \neg force\_right, \\ spill &= force\_left \text{ XOR } force\_right, \\ lift &= force\_left \wedge force\_right. \end{aligned}$$

Second, we need to address how to handle state variables that represent shared resources. In (Bacchus & Ady 2001) an example of a gas station with 6 refueling bays is given. If this resource is represented by a single state variable in ASET, we once more face the problem of at most one task accessing the resource at a time. Again, we can solve the problem by using several state variables (e.g., a Boolean variable for each refueling bay).

ASET Unit Time Transition Graphs

In order to transform an ASET description into a nondeterministic planning domain, we first compute its *unit time transition graph*. The unit time transition graph is a transition system that represents the combined effect of active tasks. As the name suggests, each transition in the unit time transition graph advances the clock one time unit.

Consider again the job shop domain shown in Figure 2. Assume that all agents are in an idle execution state in the situation depicted at the top of the figure. Suppose that the tasks $drive(a, c)$ and $paint$ are chosen for the robot and

painter, and that the human happens to choose $walk(d, c)$.[2] Figure 3 shows the reachable states in the unit time transition graph from this state until some agents are idle again. Each state is labeled with a vector showing the execution state of the robot, painter, and human respectively.

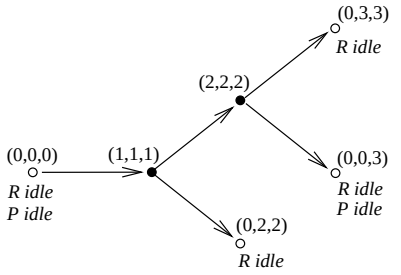


Figure 3: A subset of the unit time transition graph of the job shop domain.

For an ASET description $\mathcal{M} = \langle V, E, T\rangle$, let NC denote nonconflicting tasks of system and environment agents. We have, NC $=$ NC$^s \times$ NC^e, where NC$^x = \{\langle t_1, \ldots, t_{m_x}\rangle \in T^x : M^x_{t_i} \cap M^x_{t_j} = \emptyset \text{ for } i \neq j\}$. We can now define the unit time transition graph of an ASET description as follows.

Definition 2 (Unit Time Transition Graph) *A unit time transition graph of an ASET description $\mathcal{M} = \langle V, E, T\rangle$ is a transition system $\mathcal{T} = \langle S_\mathcal{T}, R_\mathcal{T}\rangle$, where*

$S_\mathcal{T}$ is a finite set of states $S_\mathcal{T} = V \times E \times NC$, and
$R_\mathcal{T}$ is a transition relation $R_\mathcal{T} \subseteq S_\mathcal{T} \times S_\mathcal{T}$.

For

$$\begin{aligned} s &= \langle v^s_{1..n^s}, v^e_{1..n^e}, e^s_{1..m^s}, e^e_{1..m^e}, t^s_{1..m^s}, t^e_{1..m^e}\rangle \\ s' &= \langle v'^s_{1..n^s}, v'^e_{1..n^e}, e'^s_{1..m^s}, e'^e_{1..m^e}, t'^s_{1..m^s}, t'^e_{1..m^e}\rangle \end{aligned}$$

We have $\langle s, s'\rangle \in R_\mathcal{T}$ iff

1. Running tasks transition,

   $\langle s, v'^x_{p(1)..p(n^x_{t'^x_i})}, e'^x_i\rangle \in R^x_{t'^x_i}$ for $x \in \{s, e\}, i = 1..m^x$,

   where $M^x_t = \{p(1), \ldots, p(n^x_t)\}$.
2. Non-idle tasks continue,

   $(e^x_i \neq \text{idle}) \Rightarrow (t'^x_i = t^x_i)$ for $x \in \{s, e\}$ and $i = 1..m^x$.
3. Unmodified variables maintain their value, and

   $v'^x_i = v^x_i$ for $x \in \{s, e\}$ and $i \in \{1, \ldots, m^x\} \setminus M$,

   where $M = \bigcup_{j=1}^{m^x} M^x_{t'^x_j}$.

In order to use symbolic nondeterministic planners to solve ASET planning problems, we need a Boolean encoding of unit time transition graphs. This is achieved by defining the *characteristic* function of the set of state pairs in $R_\mathcal{T}$ of the unit time transition graph. Let $\vec{s}$ and $\vec{s}'$ be two vectors of Boolean variables representing the current and next state of a unit time transition graph, where

$$\begin{aligned} \vec{s} &= \langle \vec{v}^s_{1..n^s}, \vec{v}^e_{1..n^e}, \vec{e}^s_{1..m^s}, \vec{e}^e_{1..m^e}, \vec{t}^s_{1..m^s}, \vec{t}^e_{1..m^e}\rangle, \\ \vec{s}' &= \langle \vec{v}'^s_{1..n^s}, \vec{v}'^e_{1..n^e}, \vec{e}'^s_{1..m^s}, \vec{e}'^e_{1..m^e}, \vec{t}'^s_{1..m^s}, \vec{t}'^e_{1..m^e}\rangle. \end{aligned}$$

Our goal is to define a Boolean function $R_\mathcal{T}(\vec{s}, \vec{s}')$ that is true iff the variables of $\vec{s}$ and $\vec{s}'$ are assigned values corresponding to a transition in $R_\mathcal{T}$. For an ASET description $\mathcal{M} = \langle V, E, T\rangle$, let r_i represent requirement i of Definition 2

$$\begin{aligned} r^x_1 &= \bigwedge_{i=1}^{m^x} \bigwedge_{t \in T^x_i} \Big[(\vec{t}^x_i = t) \Rightarrow R^x_t(\vec{s}, \vec{v}'^x_{p(1)..p(n^x_t)}, \vec{e}'^x_i)\Big] \\ &\quad \text{where } M^x_t = \{p(1), \ldots, p(n^x_t)\} \text{ and} \\ &\quad R^x_t(\vec{s}, \vec{v}'^x_{p(1)..p(n^x_t)}, \vec{e}'^x_i) \text{ is the characteristic} \\ &\quad \text{function of the set of tuples in } R^x_t, \\ r^x_2 &= \bigwedge_{i=1}^{m^x} \Big[(\vec{e}^x_i \neq \text{idle}) \Rightarrow (\vec{t}'^x_i = \vec{t}^x_i)\Big], \\ r^x_3 &= \bigwedge_{i=1}^{n^x} \Big[(\bigwedge_{j=1}^{m^x} \bigwedge_{t \in T^x_j(i)} \vec{t}^x_j \neq t) \Rightarrow (\vec{v}'^x_i = \vec{v}^x_i)\Big] \\ &\quad \text{where } T^x_j(i) = \{t \in T^x_j : i \in M^x_t\}. \end{aligned}$$

Further, let NCdenote the non-conflicting tasks

$$NC^x = \bigwedge_{\substack{i \in D_1 \\ j \in D_2}} \bigwedge_{\substack{t_1 \in T^x_i \\ t_2 \in T^x_j}} \begin{bmatrix} D^x_{t_1} \cap D^x_{t_2} = \emptyset \Rightarrow \\ \neg(\vec{t}^x_i = t_1 \wedge \vec{t}^x_j = t_2) \\ \wedge\neg(\vec{t}^x_i = t_1 \wedge \vec{t}^x_j = t_2) \end{bmatrix}.$$

where $D_1 = \{1, \ldots, m^x\}$ and $D_2 = \{1, \ldots, m^x\} \setminus \{i\}$.

We then have

$$R_\mathcal{T}(\vec{s}, \vec{s}') = \bigwedge_{x \in \{s,e\}} r^x_1 \wedge r^x_2 \wedge r^x_3 \wedge NC^x.$$[3]

ASET Decision Graphs

We now consider how to transform the unit time transition graph of an ASET description into a nondeterministic planning domain that we can solve efficiently with a state-of-the-art BDD-based nondeterministic planning system. The nondeterministic planning domains used by these systems are a generalization of classical deterministic planning domains where the effect of an action applied in some state is modeled by a nondeterministic choice from a set of possible next states.

[2] The result would have been the same for any of its walk tasks.

[3] Since the finite domains of ASET variables are embedded in a binary encoding, there may exist assignments to the Boolean variables that do not correspond to valid domain values. Conjoining an expression that removes these assignments from the Boolean transition relation has been omitted in the definition to simplify the presentation.

Definition 3 (Nondeterministic Planning Domain) *A nondeterministic planning domain is a tuple $\langle S, A, R\rangle$ where S is a finite set of states, A is a finite set of actions, and $R \subseteq S \times A \times S$ is a nondeterministic transition relation of action effects.*

A unit time transition graph is transformed into a nondeterministic planning domain by removing states where no planning decision can be made. A planning decision can be made in states where the task of one or more controllable agents is idle. We call such states *decision states*. For unit time transition graph of the job shop domain shown in Figure 3, all unfilled circles (the end states) are decision states. Let $D_{\mathcal{T}}$ denote these *decision states* of a unit time transition graph $\mathcal{T} = \langle S_{\mathcal{T}}, R_{\mathcal{T}}\rangle$. We have $D_{\mathcal{T}} = \{\langle \ldots, e^s_{1..m^s}, \ldots\rangle \in S_{\mathcal{T}} : e^s_i = \text{idle for some } 1 \leq i \leq m^s\}$.

The nondeterministic planning domain of an ASET description, however, also needs to include *blocking states* where some task is unable to transition. Without including these states, we may get an incorrect model that hides the fact that some decision may lead to a dead end (e.g., causing two tasks to "wait" on each other). In the job shop domain, any state where the robot is at location b and the painter is in execution state 1 of its paint task is a blocking state since the paint task is unable to transition due to the guard $posR \neq b$. Let $B_{\mathcal{T}}$ denote the blocking states of a unit time transition graph $\mathcal{T} = \langle S_{\mathcal{T}}, R_{\mathcal{T}}\rangle$. We have $B_{\mathcal{T}} = \{s \in S_{\mathcal{T}} : \langle s, s'\rangle \notin R_{\mathcal{T}} \text{ for all } s' \in S_{\mathcal{T}}\}$.

The nondeterministic planing domain associated with an ASET description is called a *decision graph*. Each transition in the decision graph corresponds to a path between decision states and blocking states in the unit time transition graph. For a set of states Q and a transition relation $U \subseteq Q \times Q$ a *path* of *length* k from v to w is a sequence of states $q_0 q_1 \cdots q_k$ such that $(q_i, q_{i+1}) \in U$ for $i = 0, \ldots, k-1$ and $v = s_0$ and $w = s_k$. We can now define the decision graph as follows.

Definition 4 (ASET Decision Graph) *Given an ASET description $\mathcal{M} = \langle V, E, T\rangle$ and a unit time transition graph $\mathcal{T} = \langle S_{\mathcal{T}}, R_{\mathcal{T}}\rangle$ of $\mathcal{M}$, an ASET decision graph of $\mathcal{M}$ is a nondeterministic planning domain $\mathcal{D} = \langle S, A, R\rangle$, where*

- S is the union of the decision and blocking states $S = D_{\mathcal{T}} \cup B_{\mathcal{T}}$,
- A is a finite set of actions $A = 2^{T^s}$, and
- R is a transition relation $R \subseteq S \times A \times S$.

For

$$\begin{aligned} s &= \langle v^s_{1..n^s}, v^e_{1..n^e}, e^s_{1..m^s}, e^e_{1..m^e}, t^s_{1..m^s}, t^e_{1..m^e}\rangle \\ s' &= \langle v'^s_{1..n^s}, v'^e_{1..n^e}, e'^s_{1..m^s}, e'^e_{1..m^e}, t'^s_{1..m^s}, t'^e_{1..m^e}\rangle \end{aligned}$$

We have $\langle s, a, s'\rangle \in R$ iff

- there exists a path $s_0 \cdots s_k$ in $R_{\mathcal{T}}$ between $s = s_0$ and $s' = s_k$ not visiting other states in S ($s_i \notin S$ for $i = 1, \ldots, k-1$), and
- the action a is the set of system tasks started in s ($a = \bigcup_{e^s_i = \text{idle}} \{t'^s_i\}$).

If $\langle s, a, s'\rangle$ is a transition in a decision graph, the current state s is a decision state and the next state s' is the first decision state or blocking state reached by some path from s when starting the tasks defined by a in the current state s.

It is nontrivial to compute the decision graph, since it is defined in terms of paths in the unit time transition graph. For symbolic nondeterministic planning, though, the decision graph can be efficiently computed using *iterative squaring* (Burch, Clarke, & McMillan 1990). Iterative squaring of a transition relation introduces transitions between all states connected by a path. The operation is defined recursively. R^0 is the original transition relation. R^1 includes all the transition in R^0, but in addition has transitions between all states in R^0 connected by a path of length 2. R^2 includes all transitions in R^1, but in addition has transitions between all states in R^1 connected by a path of length 2. Since R^1 includes R^0 this means that R^2 includes all the transitions in R^0, but in addition has transitions between all states in R^0 connected by a path of length 2,3, or 4. Thus, for each squaring of the transition relation the length of the paths for which transitions are added doubles.

Consider squaring the unit time transition graph of the job shop domain shown in Figure 3. Figure 4 shows the transitions in R^2.

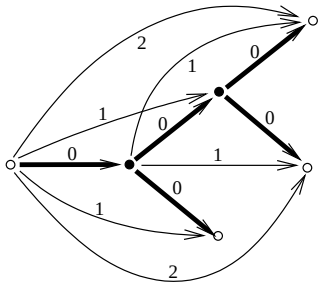


Figure 4: Squaring the unit time transition graph shown in Figure 3. Transitions in R^j for $j \leq i$ are labeled i.

We use a special version of the algorithm that ensures that all intermediate states on paths for which transitions are introduced are neither decision states nor blocking states. Let

$$\begin{aligned} B(\vec{s}) &= \neg\big[\exists \vec{s}' \,.\, R_{\mathcal{T}}(\vec{s}, \vec{s}')\big], \text{ and} \\ D(\vec{s}) &= \bigvee_{i=1}^{m^s} \vec{e}^{\,s}_i = \text{idle} \end{aligned}$$

denote the characteristic functions for the set of blocking states and decision states of a unit time transition graph with Boolean encoding $R_{\mathcal{T}}(\vec{s}, \vec{s}')$. Further, let $R^i(\vec{s}, \vec{s}')$ be defined recursively by

$$\begin{aligned} R^0_{\mathcal{T}}(\vec{s}, \vec{s}') &= R_{\mathcal{T}}(\vec{s}, \vec{s}'), \\ R^i_{\mathcal{T}}(\vec{s}, \vec{s}') &= R^{i-1}_{\mathcal{T}}(\vec{s}, \vec{s}') \vee \Big(\exists \vec{s}' \,.\, R^{i-1}_{\mathcal{T}}(\vec{s}, \vec{s}') \wedge \\ &\qquad \neg(D(\vec{s}') \vee B(\vec{s}')) \wedge R^{i-1}_{\mathcal{T}}(\vec{s}', \vec{s}'')\Big)[\vec{s}''/\vec{s}'], \\ &\qquad \text{for } i > 0. \end{aligned}$$

The operator $e[\vec{s}''/\vec{s}']$ renames double primed variables to single primed variables in the expression e. $R^0_{\mathcal{T}}$ is the transition relation of the unit time transition graph. $R^1_{\mathcal{T}}$ includes the transitions of R^0, but adds a transition $\langle s, s''\rangle$ for every path $ss's''$ where s' neither is a blocking state or decision state. Similarly R^2 adds transitions that may bypass up to 3 such states, and R^3 adds transitions that may bypass 7 etc.. In this way, we can define a Boolean encoding of the decision graph as

$$R(\vec{s}, \vec{s}') = R_{\mathcal{T}}^{\lceil \log d \rceil}(\vec{s}, \vec{s}') \wedge (D(\vec{s}) \vee B(\vec{s})) \wedge (D(\vec{s}') \vee B(\vec{s}'))$$

where d is the maximal duration of any task.

Figure 5 shows the decision graph of the unit time transition graph of the job shop domain shown in Figure 3.

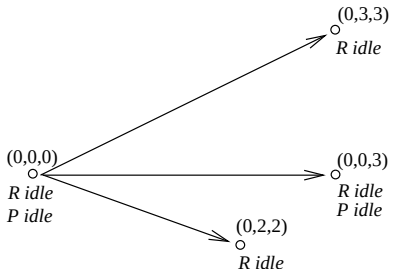


Figure 5: The decision graph of the unit time transition graph of the job shop domain shown in Figure 3.

Iterative squaring is known to be computationally complex. In our case, though, we only need to iterate to "compress" paths of length d, which often will be much less than the diameter of the transition graph. In addition, iterative squaring has been shown to be fairly efficient for digital systems dominated by clock counting (Gabodi *et al.* 1997). We may expect ASET domains where tasks have long duration to be structurally similar to this kind of circuits.

Solving ASET Planning Problems

The transformation of an ASET description to a nondeterministic planning domain and the Boolean encoding of the decision graph, allows us to use efficient symbolic nondeterministic planning algorithms (Cimatti *et al.* 2003; Jensen & Veloso 2000) including heuristic symbolic search algorithms (Jensen, Veloso, & Bryant 2003) to solve ASET planning problems. In the remainder of this section, we apply the machinery developed for nondeterministic symbolic planning to define ASET planning problems and solutions.

Definition 5 (Nondeterministic Planning Problem) *A nondeterministic planning problem is a tuple $\langle \mathcal{D}, s_0, G\rangle$ where $\mathcal{D}$ is a nondeterministic planning domain, s_0 is an initial state, and $G \subseteq S$ is a set of goal states.*

For a nondeterministic planning domain $\mathcal{D} = \langle S, A, R\rangle$, the set of possible next states of an action a applied in state s is given by $\textsc{Next}(s, a) \equiv \{s' : \langle s, a, s'\rangle \in R\}$. An action a is called *applicable* in state s iff $\textsc{Next}(s, a) \neq \emptyset$. The set of applicable actions in a state s is given by $\textsc{App}(s) \equiv \{a : \textsc{Next}(s, a) \neq \emptyset\}$. A nondeterministic plan is a set of *state-action pairs* (SAs).

Definition 6 (Nondeterministic Plan) *Let $\mathcal{D}$ be a nondeterministic planning domain. A nondeterministic plan for $\mathcal{D}$ is set of state-action pairs $\{\langle s, a\rangle : a \in \textsc{App}(s)\}$.*

The set of SAs define a function from states to sets of actions relevant to apply in order to reach a goal state. States are assumed to be fully observable. An execution of a nondeterministic plan is an alternation between observing the current state and choosing an action to apply from the set of actions associated with the state. Notice that the definition of a nondeterministic plan does not give any guarantees about goal achievement. The reason is that, in contrast to deterministic plans, it is natural to define a range of solutions classes. We are particularly interested in strong plans that guarantee goal achievement in a finite number of steps. Following (Cimatti *et al.* 2003), we define strong plans formally by as a CTL formula that must hold on a Kripke structure representing the execution behavior of the plan.

A set of states *covered* by a plan π is $\textsc{States}(\pi) \equiv \{s : \exists a\,.\,\langle s, a\rangle \in \pi\}$. The set of actions in a plan π associated with a state s is $\textsc{Act}(\pi, s) \equiv \{a : \langle s, a\rangle \in \pi\}$. The *closure* of a plan π is the set of possible end states $\textsc{Closure}(\pi) \equiv \{s' \notin \textsc{States}(\pi) : \exists \langle s, a\rangle \in \pi\,.\,s' \in \textsc{Next}(s, a)\}$.

Definition 7 (Execution Model) *An execution model with respect to a nondeterministic plan π for the domain $\mathcal{D} = \langle S, A, R\rangle$ is a Kripke structure $\mathcal{M}(\pi) = \langle Q, U\rangle$ where*

- $Q = \textsc{Closure}(\pi) \cup \textsc{States}(\pi) \cup G$,
- $\langle s, s'\rangle \in U$ *iff* $s \notin G,\ \exists a\,.\,\langle s, a\rangle \in \pi$ *and* $\langle s, a, s'\rangle \in R$, *or* $s = s'$ *and* $s \in \textsc{Closure}(\pi) \cup G$.

Notice that all execution paths are infinite which is required in order to define solutions in CTL. If a state is reached that is not covered by the plan (e.g., a goal state or a dead end), the postfix of the execution path from this state is an infinite repetition of it. Given a Kripke structure defining the execution of a plan, strong plans are defined by the CTL formula below.

Definition 8 (Strong Plans) *Given a nondeterministic planning problem $\mathcal{P} = \langle \mathcal{D}, s_0, G\rangle$ and a plan π for $\mathcal{D}$, π is a strong plan iff $\mathcal{M}(\pi), s_0 \models \texttt{AF}\, G$.*

The expression $\mathcal{M}(\pi), s_0 \models \texttt{AF}\, G$ is true if all execution paths lead to a goal state in a finite number of steps.

Fault Tolerance

A weakness of strong plans is that they can be very conservative. In real-world domains most actions may fail. If fault behavior is modeled via nondeterminism, a strong plan only exists if the worst case behavior of the plan, where all actions fail, still leads to a goal state. This is seldom the case. We would like to be able to state a weaker kind of plans that do not have to cover the most unlikely execution paths. As mentioned in the introduction, going all the way to probabilistic planning is not a solution due to the high computational cost. But we can rephrase a plan with high probability of success

as a plan with high tolerance for failures encountered during execution. Such plans can be defined fully within the framework of nondeterministic planning. Plans that guarantee goal achievement if no more than n actions fail during execution are called n-fault tolerant plans (Jensen, Veloso, & Bryant 2004). Fault tolerant plans can be computed via strong plans by adding fault counters to the domain.[4] This is also possible for ASET domains.

We define a failure of a task as a unit time transition leading to the idle state. In order to generate n-fault tolerant plans, we add a special fault counter state variable f_i for each controllable agent i. For each task of agent i that can fail, we extend the guard and effect of each unit time transition denoting failure with the expression $n > \sum_{i=1}^{m_s} f_i$ and $f_i' = f_i + 1$, respectively. For the remaining transitions of the task, we maintain the value of f_i by extending the effect with $f_i' = f_i$. Finally, the initial state is extended with $f_i = 0$ for $i = 1 \ldots m^s$ and the goal states are extended with $n \geq \sum_{i=1}^{m_s} f_i$. In this way failures can only happen in the fault extended problem if less than n failures have occurred so far. This is precisely the assumption of n-fault tolerant plans and ensures that a strong plan of the fault extended problem is a valid n-fault tolerant plan.

Experimental Evaluation

We have implemented a planning system in C++/STL using the BuDDy BDD package (Lind-Nielsen 1999). Given a textual ASET description, it computes two BDDs representing the transition relation of the unit time transition graph and its associated decision graph.

The experiment reported in this section investigates how fast the computational complexity of synthesizing the decision graph grows with the temporal granularity of the unit time decision graph. We consider a parameterized version of the job shop domain where each task is extended with extra unit time transitions such that the overall structure of the task is maintained. For instance, unit time transitions are added on both the left and right side of the early termination of the painter's paint task and the robot's drive task. Since the number of possible ways that tasks can be temporally aligned grows fast with their duration, computing the decision graph could potentially be hard.

We conducted the experiments on a 3GHz Pentium 4 with 1024KB L2 cache and 2GB RAM [5] running Linux kernel 2.4.25. Figure 6 shows the computation time of the unit time transition graph and the decision graph. As depicted, the CPU time for computing the unit time transition graph is very low for all versions of the domain. Despite the much longer time needed to compute the decision graph, the asymptotic complexity of this operation is low. Notice the jumps in computation time when the iterative squaring involves computing a new intermediate transition relation.

[4]While this approach is conceptually easy to understand, much better performance can be achieved in real-world domains by distinguishing semantically between failure effects and successful effects and use specialized planning algorithms (Jensen, Veloso, & Bryant 2004).

[5]The experiments, however, were limited to 500MB RAM.

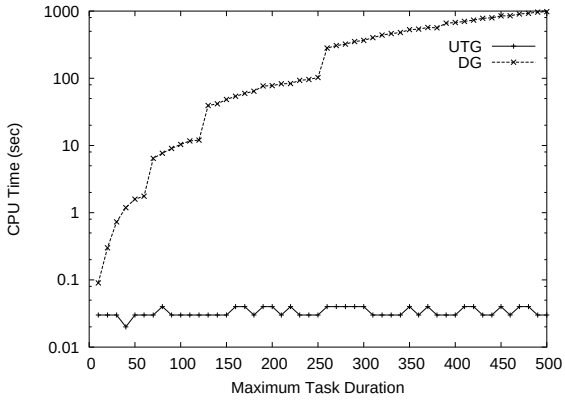


Figure 6: CPU time for synthesizing the unit time transition graph (UTG) and the decision graph (DG) as a function of maximum task duration.

Fortunately the distance between these jumps grows exponentially with the maximum task duration.

Figure 7 shows how the BDD size of the unit time transition graph and the decision graph grows as a function of the maximum task duration. As depicted, the BDD size of the

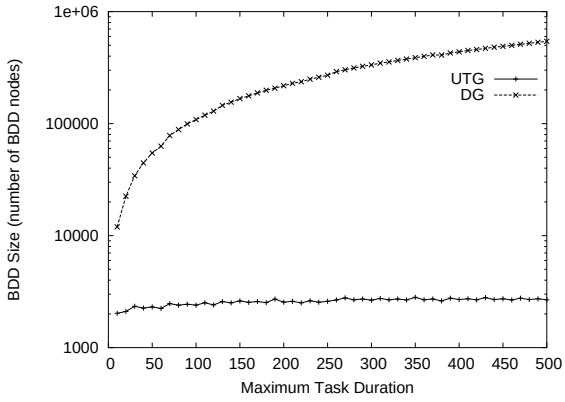


Figure 7: BDD size of the unit time transition graph (UTG) and the decision graph (DG) as a function of maximum task duration.

decision graph grows approximately linearly with the computation time of the decision graph. It may be surprising that the BDD of the decision graph is larger than the BDD of the unit time transition graph. Since BDDs represent transitions implicitly, there is no simple relation between the size of a BDD and the number of transitions it represent. That the BDD representing the decision graph is large merely indicates that the subspace of transitions in the decision graph is less structured than that of the unit time decision graph. The question is to what extend this will impair BDD based planning based on the decision graph. Future experiments will address this issue.

Conclusion

In this paper, we have introduced a new multi-agent planning language called ASET. The main contribution of ASET is Evolving Tasks (ETs). ETs are, as far as we know, the first action description that in an explicit and intuitive way can represent temporally extended activities which are nondeterministic both with respect to duration and effect. ETs are represented as directed acyclic graphs that in a natural way solves the problem of representing conditional effects and intermediate effects of durative actions.

We have formally defined ASET descriptions and shown how they can be transformed into nondeterministic planning domains. Using a Boolean encoding of these domains, efficient symbolic nondeterministic planning algorithms can be used to solve ASET planning problems.

ASET shows that it is possible to model essential aspects of time and stochastic behavior in a language with a representational power as low as a nondeterministic finite automata. This is encouraging since the main challenge of automated planning is to scale to the size of real-world domains, and since dense time and probabilistic models come with a high computational fee.

Preliminary results show that the decision graph of ASET domains can be generated efficiently even for domains with a high level of temporal detail. Future work includes further experiments investigating BDD-based planning based on ASET decision graphs and developing more efficient ways of generating and representing decision graphs (e.g., by using transition relation partitioning (Burch, Clarke, & Long 1991)).

References

Bacchus, F., and Ady, M. 2001. Planning with resources and concurrency: A forward chaining approach. In *International Joint Conference on Artificial Intelligence (IJCAI-01)*, 417–424.

Bryant, R. E. 1986. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers* 8:677–691.

Burch, J.; Clarke, E.; and Long, D. 1991. Symbolic model checking with partitioned transition relations. In *International Conference on Very Large Scale Integration*, 49–58. North-Holland.

Burch, J. R.; Clarke, E. M.; and McMillan, K. 1990. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science*, 428–439.

Cassandras, C. G., and Lafortune, S. 1999. *Introduction to Discrete Event Systems*. Kluwer Academic Publishers.

Cimatti, A.; Pistore, M.; Roveri, M.; and Traverso, P. 2003. Weak, Strong, and Strong Cyclic Planning via Symbolic Model Checking. *Artificial Intelligence* 147(1-2). Elsevier Science publishers.

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

Gabodi, G.; Camurati, P.; Lavagno, L.; and Quer, S. 1997. Disjunctive partitioning and partial iterative squaring. In *Proceedings of the 34th Design Automation Conference DAC-97*.

Gelfond, M.; Lifschitz, V.; and Rabinov, A. 1991. What are the limitations of the situation calculus. In *Essays in Honor of Woody Bledsoe*. Kluwer Academic. 167–179.

Giunchiglia, E.; Kartha, G. N.; and Lifschitz, Y. 1997. Representing action: Indeterminacy and ramifications. *Artificial Intelligence* 95:409–438.

Hansen, E., and Zilberstein, S. 2001. LAO*: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence* 129:35–62.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research (JAIR)* 14:253–302.

Jensen, R. M., and Veloso, M. M. 2000. OBDD-based universal planning for synchronized agents in non-deterministic domains. *Journal of Artificial Intelligence Research* 13:189–226.

Jensen, R. M.; Veloso, M. M.; and Bryant, R. E. 2003. Guided symbolic universal planning. In *Proceedings of the 13th International Conference on Automated Planning and Scheduling ICAPS-03*, 123–132.

Jensen, R. M.; Veloso, M. M.; and Bryant, R. E. 2004. Fault tolerant planning: Toward probabilistic uncertainty models in symbolic non-deterministic planning. In *Proceedings of the 14th International Conference on Automated Planning and Scheduling ICAPS-04*.

Laborie, P., and Ghallab, M. 1995. Planning with sharable resource constraints. In *Proceedings of (IJCAI-95)*, 1643–1649.

Lind-Nielsen, J. 1999. BuDDy - A Binary Decision Diagram Package. Technical Report IT-TR: 1999-028, Institute of Information Technology, Technical University of Denmark. `http://cs.it.dtu.dk/buddy`.

Lingard, A. R., and Richards, E. B. 1998. Planning parallel actions. *Artificial Intelligence* 99:261–324.

Musliner, D. J.; Durfee, E. H.; and Shin, K. G. 1993. CIRCA: A cooperative intelligent real time control architecture. *IEEE Transactions on Systems, Man, and Cybernetics* 23(6):1561–1574.

Piergiorgio, B.; Bonet, B.; Cimatti, A.; Giunchiglia, E.; Golden, K.; Rintanen, J.; and Smith., D. E. 2002. The NuPDDL home page. `http://sra.itc.it/tools/mbp/#nupddl`.

Younes, H. L. S. 2003. Extending PDDL to model stochastic decision processes. In *Proceedings of the 13th International Conference on Automated Planning and Scheduling (ICAPS-03) Workshop on PDDL*, 95–103.

Robust Distributed Coordination of Heterogeneous Robots through Temporal Plan Networks *

Andreas F. Wehowsky, Stephen A. Block and **Brian C. Williams**
Computer Science and Artificial Intelligence Laboratory,
Massachusetts Institute of Technology,
Cambridge, MA, 02139, USA
andreas@wehowsky.dk, {sblock, williams}@mit.edu

Abstract

Real-world applications of autonomous agents require coordinated groups to work in collaboration. Dependable systems must plan and carry out activities in a way that is robust to failure and uncertainty. Previous work has produced algorithms that provide robustness at the planning phase, by choosing between functionally redundant methods, and the execution phase, by dispatching temporally flexible plans. However, these algorithms use a centralized architecture in which all computation is performed by a single processor. As a result, these implementations require significant computational capabilities, introduce a single point of failure, do not scale well, and suffer from communication bottlenecks.

This paper introduces the plan extraction component of a robust, distributed executive for contingent plans. Contingent plans are encoded as Temporal Plan Networks (TPNs), which compose temporally flexible plans hierarchically and provide a choose operator. First, the TPN is distributed over multiple agents, by creating a hierarchical ad-hoc network and mapping the TPN onto this hierarchy. Second, candidate plans are extracted from the TPN with a distributed, parallel algorithm that exploits the structure of the TPN. Third, temporal consistency of the candidate plans is tested using a distributed Bellman-Ford algorithm. This algorithm is empirically validated on randomized contingent plans.

Introduction

The ability to command coordinated groups of autonomous agents is key to many real-world tasks, such as the construction of a Lunar habitat. In order to achieve this goal, we must perform robust execution of contingent, temporally flexible plans in a distributed manner. Methods have been developed for the dynamic execution (Morris & Muscettola 1999) of temporally flexible plans (Dechter, Meiri, & Pearl 1990). These methods adapt to failures that fall within the margins of the temporally flexible plans and hence add robustness to execution uncertainties.

To address plan failure, (Kim, Williams, & Abramson 2001) introduced a system called *Kirk*, that performs dynamic execution of temporally flexible plans with contingencies. These contingent plans are encoded as alternative choices between functionally equivalent sub-plans. In Kirk, the contingent plans are represented by a Temporal Plan Network (TPN) (Kim, Williams, & Abramson 2001), which extends temporally flexible plans with a nested choose operator. To dynamically execute a TPN, Kirk continuously extracts a plan from the TPN that is temporally feasible, given the execution history, and dispatches the plan, using the methods of (Tsamardinos, Muscettola, & Morris 1998). Dynamic execution of contingent plans adds robustness to plan failure. However, as a centralized approach, Kirk is extremely brittle to the loss of the processor performing execution and, in the case of multi-agent coordination, is brittle to loss of communication.

We address these two limitations through a distributed version of Kirk, which performs distributed dynamic execution of contingent temporally flexible plans. This paper focuses on the algorithm for dynamically selecting a feasible plan from a TPN. Methods for performing distributed execution of the plan are presented in (Stedl 2004). Our key innovation is a hierarchical algorithm for searching a TPN for a feasible plan in a distributed manner. In particular, our plan selection algorithm, called the Distributed Temporal Planner (DTP), is comprised of three stages.

1. Distribute the TPN across the processor network,
2. Generate candidate plans through distributed search on the TPN, and
3. Test the generated plans for temporal consistency.

This paper begins with an example TPN and an overview of the way in which DTP operates on it. We provide a formal definition of a TPN and then discuss the three stages of DTP. Finally, we discuss the complexity of the DTP algorithm and present experimental results demonstrating its performance.

Example Scenario

In this section, we discuss at a high level the three step approach taken by DTP to solve an example problem. A TPN is to be executed by a group of seven processors, $p1, \ldots, p7$. The TPN is represented as a graph in Fig. 1, where nodes represent points in time and arcs represent activities. A node at which multiple choices exist for the following path through the TPN is a *choice node* and is shown as an inscribed circle.

First, the TPN itself is distributed over the processors to allow the plan selection to take place in a distributed fash-

*This work was made possible by the sponsorship of the DARPA NEST program under contract F33615-01-C-1896

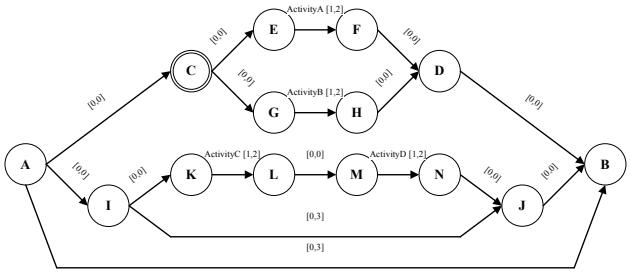


Figure 1: Example TPN

ion. To facilitate this, a leader election algorithm is used to arrange the processors into a hierarchy (Fig. 2). The hierarchical structure of the TPN is then used to map subnetworks to processors. For example, the head processor $p1$ handles the merging of multiple branches of the plan at the start node (node A) and the end node (node B). It passes responsibility for each of the two main subnetworks to the two processors immediately beneath it in the hierarchy. Nodes *C,D,E,F,G,H* are passed to $p2$ and nodes *I,J,K,L,M,N* are passed to $p3$.

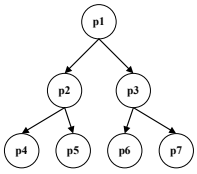


Figure 2: A three-level hierarchy formed by leader election

The processors then work together to extract a temporally consistent plan from the TPN. The first stage generates a candidate plan, which corresponds to selecting a single subnetwork from the plan at each of the choice nodes. This is done in a hierarchical fashion, where each processor sends messages to its neighbors, requesting that they make selections in the subnetworks for which they are responsible. These selections are made in parallel. In this example, only the subnetwork owned by $p2$ (nodes C,D,E,F,G,H) contains a choice of path, so $p2$ must decide between `ActivityA` and `ActivityB`, whereas $p3$ has no choice to make.

Having generated a candidate plan, the third and final step of DTP is to test it for consistency. Again, this is done in a hierarchical fashion, where consistency checks are first made at the lowest level and successful candidates are then checked at an increasingly high level. For example, $p2$ and $p3$ simultaneously check that their subnetworks are internally consistent. If so, $p1$ then checks that the two candidates are consistent when executed in parallel. In DTP, candidate generation and consistency checking are interleaved, such that some processors generate candidates while others simultaneously check consistency.

Temporal Plan Networks

A TPN augments temporally flexible plans with a `choose` operator and is used by DTP to represent a contingent, temporally flexible plan. The `choose` operator allows us to specify nested choices in the plan, where each choice is an alternative sub-plan that performs the same function.

The primitive element of a TPN is an `activity`$[l, u]$, which is a hardware command with a simple temporal constraint. The simple temporal constraint $[l, u]$ places a bound $t^+ - t^- \in [l, u]$ on the start time t^- and end time t^+ of the network to which it is applied. A TPN is built from a group of activities and is defined recursively using the `choose`, `parallel` and `sequence` operators, which derive from the Reactive Model-based Programming Language (RMPL) (Williams *et al.* 2003).

- `choose`$(TPN_1, \ldots, TPN_N)$ introduces multiple subnetworks of which only one is to be chosen. A choice variable is used at the start node to encode the currently selected subnetwork. A choice variable is *active* if it falls within the currently selected portion of the TPN.
- `parallel`$(TPN_1, \ldots, TPN_N)\,[l, u]$ introduces multiple subnetworks to be executed concurrently. A simple temporal constraint is applied to the entire network. Each subnetwork is referred to as a *child* subnetwork.
- `sequence`$(TPN_1, \ldots, TPN_N)\,[l, u]$ introduces multiple subnetworks which are to be executed sequentially. A simple temporal constraint is applied to the entire network. For a given subnetwork, the subnetwork following it in a `sequence` network is referred to as its *successor*.

Graph representations of the `activity`, `choose`, `parallel` and `sequence` network types are shown in Fig. 3. Nodes represent time events and directed edges represent simple temporal constraints.

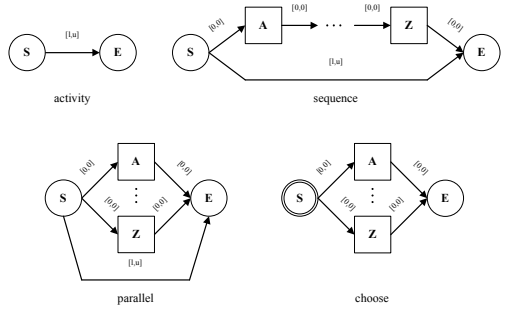


Figure 3: TPN Constructs

Definition 1 *A **feasible solution** of a TPN is an assignment to choice variables such that 1) all active choice variables are assigned, 2) all inactive choice variables are unassigned, and 3) the currently selected temporally flexible plan is temporally consistent. A temporally flexible plan is **temporally consistent** if there exist times that can be assigned to all events such that all temporal constraints are satisfied.*

TPN Distribution

The DTP algorithm distributes the computation involved in finding a feasible solution to the TPN over all available processors. Consequently, the processors must be able to communicate with each other, in order to coordinate their actions. We therefore establish an ad-hoc communication network such that adjacent processors are able to communicate. In addition, an overall leader must be selected to communicate with the outside world and initiate planning.

Ad-Hoc Processor Network Formation

We use the leader election algorithm in (Nagpal & Coore 1998) to arrange the processors into a hierarchical network, an example of which is shown in Fig. 2. For each node, the node immediately above it in the hierarchy is its *leader*, those at the same level within that branch of the hierarchy are its *neighbor leaders* and those directly below it in the hierarchy are its *followers*. The leader election algorithm forms the hierarchy using a message passing scheme and in doing so, ensures that every node can communicate with its leader, as well as all neighbor leaders and followers. In addition, the hierarchical nature of the network lends itself well to the distribution of the TPN, which is also hierarchical.

TPN Distribution over the Processor Network

We implement the distribution of the DTP computation by assigning to each processor responsibility for a number of nodes from the TPN graph representation. Each processor maintains all the data from the TPN relevant to the nodes for which it is responsible.

This distribution scheme requires that processors responsible for TPN nodes linked by temporal constraints are able to communicate. The algorithm in Fig. 4 distributes the TPN over the processor hierarchy such that this communication is available. It allows distribution down to the level at which a processor handles only a single node. This allows DTP to operate on heterogeneous systems that include computationally impoverished processors.

```
 1: wait for TPN
 2: n ← number of followers of p
 3: if TPN is of type activity then
 4:    assign start and end nodes of TPN to p
 5: else
 6:    k ← number of subnetworks
 7:    assign start and end nodes to p
 8:    if n = 0 then
 9:       if p has a neighbor leader v then
10:          send k/2 subnetworks of TPN to v
11:          assign k/2 subnetworks of TPN to p
12:       else
13:          assign TPN to p
14:       end if
15:    else if n ≥ k then
16:       for each of k subnetworks of TPN do
17:          assign subnetwork of TPN to a follower of p
18:       end for
19:    else if n < k then
20:       for each of n subnetworks of TPN do
21:          assign subnetwork to a follower of p
22:       end for
23:       assign remaining (k − n) subnetworks of TPN to p
24:    end if
25: end if
```

Figure 4: TPN Distribution Algorithm for node p

We now demonstrate the distribution algorithm using the TPN in Fig. 1 and the processor hierarchy in Fig. 2. The TPN is supplied from an external source, which establishes a connection with the top leader, *p1*. The TPN is a `parallel` network at the highest level, so processor *p1* assigns the start and end nodes (nodes *A,B*) to itself (line 7). There are two subnetworks, which *p1* assigns to its two followers, *p2* and *p3* (lines 15-18). *p1* passes the `choose` network (nodes *C,D,E,F,G,H*) to *p2* and the `sequence` network (nodes *I,J,K,L,M,N*) to *p3*. *p2* and *p3* then process their networks in parallel. *p2* assigns the start and end nodes (nodes *C,D*) to itself (line 7). The network has two subnetworks, which *p2* assigns to two of its followers, *p4* and *p5* (lines 15-18). *p2* passes `ActivityA` (nodes *E,F*) to *p4* and `ActivityB` (nodes *G,H*) to *p5*. Since activities can not be decomposed, *p4* and *p5* assign nodes *E,F* and *G,H*, respectively, to themselves (lines 3-4). Meanwhile, *p3* receives the `sequence` network and assigns the start and end nodes (nodes *I,J*) to itself (line 7). The network has two subnetworks, which *p3* assigns to two of its followers, *p6* and *p7* (lines 15-18). *p3* passes `ActivityC` (nodes *K,L*) to *p6* and `ActivityD` (nodes *M,N*) to *p7*. *p6* and *p7* then assign nodes *K,L* and nodes *M,N*, respectively, to themselves (lines 3-4).

Candidate Plan Generation

Having distributed the TPN across the available processors, DTP conducts search for candidate plans. These plans correspond to different assignments to the choice variable at each choice node (Mittal & Falkenhainer 1990). DTP uses parallel, recursive, depth first search to make these assignments. This use of parallel processing is one of the key advantages of DTP over traditional centralized approaches. DTP is implemented using a distributed message-passing architecture and uses the following messages during candidate plan generation.

- `findfirst` instructs a network to make the initial search for a consistent set of choice variable assignments.
- `findnext` is used when a network is consistent internally, but is inconsistent with other networks. In this case, DTP uses `findnext` messages to conduct a systematic search for a new consistent assignment, in order to achieve global consistency. `findnext` systematically moves through the subnetworks and returns when the first new consistent assignment is found. Therefore, a successful `findnext` message will cause a change to the value assigned to a single choice variable, which may in turn cause other choice variables to become active or inactive.
- `fail` indicates that no consistent set of assignments was found and hence the current set of assignments within the network is inconsistent.
- `ack`, short for acknowledge, indicates that a consistent set of choice variable assignments has been found.

Whenever a node initiates search in its subnetworks, using `findfirst` or `findnext` messages, the relevant processors search the subnetworks simultaneously. This is the origin of the parallelism in the algorithm.

DTP operates on three network types formed from the four types fundamental to a TPN. These are `activity`, `parallel-sequence` and `choose-sequence`, as shown in Fig. 5, where the subnetworks $A_i, \ldots, Z_i$ are of any of these three types. We handle the simple temporal constraint present on a `sequence` network by considering a `sequence` network as a special case of a `parallel-sequence` network, in which only one subnetwork exists.

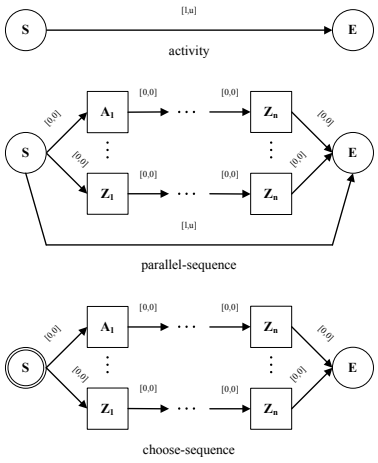


Figure 5: Constructs for DTP

This choice of network types requires that a network is able to communicate directly with any its successor. This is made possible by the Sequential Network Identifier (SNI), which is a pointer to the start node of the successor network.

The following three sections describe the actions carried out by the start node of each network type on receipt of a `findfirst` or `findnext` message. Note that while a simple temporal constraint $[l, u]$ is locally inconsistent if $l > u$, we assume that the TPN is checked prior to running DTP, to ensure that all temporal constraints are locally consistent. This assumption means that only `parallel-sequence` networks can introduce temporal inconsistencies.

Activity During search, an `activity` node propagates request messages forward and response messages backward.

Parallel-Sequence Network On receipt of a `findfirst` message, the start node v of a `parallel-sequence` network S calls `parallel-findfirst(`v`)` (Fig. 6). The node initiates a search of S's subnetworks and of any successor network, in order to find a temporally consistent plan. First, the start node sends `findfirst` messages to the start node of each child subnetwork of the `parallel` structure (lines 2-4) and to the start node of the successor network, if present (lines 5-7). These searches are thus conducted in parallel. If any of the child subnetworks or the successor network returns a `fail` message (line 12), then no consistent assignment to the choice variables exists and the start node returns `fail` (line 13).

Conversely, suppose that all child subnetworks and the successor network return `ack` messages, indicating that variable assignments have been made such that each is internally temporally consistent. The start node must then check for consistency of the entire `parallel` network S (line 15). This is performed by a distributed Bellman Ford consistency checking algorithm, which is explained in the next section. If the consistency check is successful, the start node returns an `ack` message to its parent (line 16) and the search of the `parallel` network is complete.

If, however, the consistency check is not successful, the start node must continue searching through all permutations

```
 1: parent ← sender of msg
 2: for each child do
 3:    send findfirst to w
 4: end for
 5: if successor B exists then
 6:    send findfirst to B
 7: end if
 8: wait for all responses from children
 9: if successor B exists then
10:    wait for response from B
11: end if
12: if any of the responses is fail then
13:    send fail to parent
14: else
15:    if check-consistency( v ) then
16:       send ack to parent
17:    else
18:       if search-permutations(v) then
19:          send ack to parent
20:       else
21:          send fail to parent
22:       end if
23:    end if
24: end if
```

Figure 6: `parallel-findfirst(node` v`)`

of assignments to the child subnetworks for a globally consistent solution. It calls `search-permutations(`v`)` (line 18) and sends an `ack` message to its parent if this is successful and a `fail` message otherwise.

In the `search-permutations(node` v`)` function (Fig. 7), the start node sends `findnext` messages to each subnetwork (lines 1-2). If a subnetwork returns `fail`, the start node sends a `findfirst` message to that subnetwork to reconfigure it to its original, consistent solution (lines 11-12) and we move on to the next subnetwork. If at any point, a subnetwork returns `ack`, the start node tests for global consistency and returns `true` if successful (lines 4-6). If the consistency check is unsuccessful, we try a different permutation of variable assignments (line 8) and continue searching. If all permutations are tested without success, the function returns `false` (line 15).

```
 1: for w = child-0 to child-n do
 2:    send findnext to w
 3:    wait for response
 4:    if response = ack then
 5:       if check-consistency( v ) then
 6:          return true
 7:       else
 8:          w ← child-0
 9:       end if
10:    else
11:       send findfirst to w
12:       wait for response
13:    end if
14: end for
15: return false
```

Figure 7: `search-permutations(node` v`)` function

When the start node v of a `parallel-sequence` network receives a `findnext` message, it executes

`parallel-findnext(`v`)` (Fig. 8). First, the start node calls `search-permutations(`v`)` to systematically search all consistent assignments to its subnetworks, in order to find a new globally consistent assignment (line 1). If this is successful, the start node sends `ack` to its parent (line 2). If it fails, however, the start node attempts to find a new assignment to the successor network. If a successor network is present, the start node sends a `findnext` message and returns the response to its parent (lines 3-6). If no successor network is present, then no globally consistent assignment exists and the node returns `fail` (line 8).

```
1: if search-permutations() then
2:    send ack to parent
3: else if successor B exists then
4:    send findnext to B
5:    wait for response
6:    send response to parent
7: else
8:    send fail to parent
9: end if
```

Figure 8: `parallel-findnext(node` v`)` function

Choose-Sequence Network When the start node of a `choose-sequence` network receives a `findfirst` message, it executes the `choose-findfirst()` function (Fig. 9). The node searches for a consistent plan by making an appropriate assignment to its choice variable. It also initiates a search in any successor network. To do so, it first sends a `findfirst` message to the successor network if present (lines 2-4). It then systematically assigns each possible value to the network's choice variable and, in each case, sends a `findfirst` message to the enabled subnetwork (lines 5-7). If a subnetwork returns `fail`, indicating that no consistent assignment exists, the current value of the choice variable is trimmed from its domain to avoid futile repeated searches (line 18), and the next value is assigned.

```
1: parent ← sender of msg
2: if successor B exists then
3:    send findfirst to B
4: end if
5: for w = child-0 to child-n do
6:    choicevariable ← w
7:    send findfirst to w
8:    wait for response from child w
9:    if response = ack then
10:      if successor B exists then
11:         wait for response from successor B
12:         send response to parent
13:      else
14:         send ack to parent
15:      end if
16:      return
17:   else
18:      remove w from child list
19:   end if
20: end for
21: send fail to parent
```

Figure 9: `choose-findfirst()` function

As soon as a subnetwork returns `ack`, indicating that a consistent assignment to the subnetwork was found, the start node waits for a response from the successor network (if present) to determine whether or not a consistent assignment was found to it too (line 11). Once a response has been received from the successor network, the start node forwards this response to its parent and the search terminates (line 12). If no successor network is present, the network is consistent and the start node returns `ack` to its parent (line 14).

If all assignments to the network's choice variable are tried without receipt of an `ack` message from a child subnetwork, the start node returns `fail` to its parent, indicating that no consistent assignment exists (line 21).

When the start node of a `choose` network receives a `findnext` message, it executes the `choose-findnext()` function (Fig. 10). The start node first attempts to find a new consistent assignment for the network while maintaining the current value of the choice variable. It does so by sending `findnext` to the currently selected subnetwork (lines 1-2). If the response is `ack`, a new consistent assignment has been found, so the start node returns `ack` to its parent and the search is over (lines 4-6).

```
1: w ← current assignment
2: send findnext to w
3: wait for response
4: if response = ack then
5:    send ack to parent
6:    return
7: end if
8: while w < child-n do
9:    w ← next child
10:   send findfirst to w
11:   wait for response
12:   if response = ack then
13:      send ack to parent
14:      return
15:   else
16:      remove w from child list
17:   end if
18: end while
19: if successor B exists then
20:   send findnext to B
21:   for w = child0 to child-n do
22:      choice variable ← w
23:      send findfirst to w
24:      wait for response from child w
25:      if response = ack then
26:         break
27:      end if
28:   end for
29:   wait for response from B
30:   send response to parent
31: else
32:   send fail to parent
33: end if
```

Figure 10: `choose-findnext()` function

If this fails, however, the start node searches through unexplored assignments to the network's choice variable, in much the same way as it does on receipt of a `findfirst` message (lines 8-18). Finally, if this strategy also fails, the start node attempts to find a new consistent assignment in any successor network, by sending a `findnext` message

to the node referenced by its SNI parameter (lines 19-20). Note that the start node must reset the local network to the previous consistent configuration, because the unsuccessful search has left it in an inconsistent state. This is achieved by repeating the search process used on receipt of a `findfirst` message (lines 21-28). Once the successor network has replied, the start node forwards the response to its parent (lines 29-30).

Temporal Consistency Checking

Each of the candidate assignments generated during search on the TPN must be tested for temporal consistency, which is implemented by the `check-consistency(node` v`)` function. Consistency checking is performed with the distributed Bellman-Ford Single Source Shortest Path algorithm (Lynch 1997), which is run on the distance graph corresponding to the currently active portion of the TPN. Temporal inconsistency is detected as a negative weight cycle (Dechter, Meiri, & Pearl 1990). The consistency checking process is interleaved with candidate generation, such that DTP simultaneously runs multiple instances of the distributed Bellman-Ford algorithm on isolated subsets of the TPN.

The distributed Bellman-Ford algorithm has two key advantages. First, it requires only local knowledge of the network at every processor. Second, when run synchronously, it runs in time linear in the number of processors in the network. DTP ensures synchronization by the fact that whenever a node initiates search in its subnetworks, it waits for responses from all processors in the form of `ack` or `fail` messages before proceeding.

Performance Analysis

The overall time complexity of the centralized planning algorithm is worst-case exponential. The backtrack search used to assign choice variables has worst-case time complexity N^e, where N is the number of nodes and e is the size of the domain of the choice variables. The Bellman-Ford algorithm used for consistency checking has complexity $N^2 logN + NM$, where M is the number of edges.

DTP also has exponential overall time complexity. The backtrack search remains N^e in the worst case, but we gain significant computational savings from the fact that the distributed Bellman-Ford algorithm runs in time N.

Discussion and Results

DTP was implemented in C++ and tested by simulating an array of processors searching for a feasible solution of a TPN, where exactly one node was assigned to each processor. The number of nodes in the TPN was varied between 1 and 100. In each case, the number of TPN constructs (`parallel`, `sequence` or `choose`) was varied between 3 and 30 and the maximum recursive depth was varied between 4 and 10. Performance was measured by the number of listen-act-respond cycles completed by the processor network.

Fig. 11 shows a plot of the number of cycles against the number of nodes. The results showed that the variation in the number of cycles, which is a measure of run-time, is approximately linear with the number of nodes. The worst-case time complexity of DTP is exponential, but this occurs only when the TPN is composed entirely of `choose` networks, in which case there is no opportunity for parallel execution. However, typical TPNs used in real applications consist largely of `parallel` and `sequence` networks. This allows processors to conduct parallel search and consistency checks, which greatly reduces the time complexity of DTP.

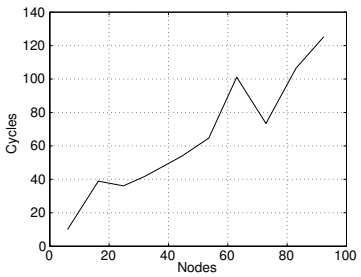


Figure 11: Number of cycles vs. number of nodes

This paper introduced the Distributed Temporal Planner (DTP), which is the plan selection component of a distributed executive that operates on contingent, temporally flexible plans. DTP distributes both data and processing across all available agents. First, DTP forms a processor hierarchy and assigns subnetworks from the TPN to each processor. It then searches the TPN to generate candidate plans, which are finally checked for temporal consistency. DTP exploits the hierarchical nature of TPNs to allow parallel processing in all three phases of the algorithm.

References

Dechter, R.; Meiri, I.; and Pearl, J. 1990. Temporal constraint networks. *Artificial Intelligence, 49:61-95, 1991*.

Kim, P.; Williams, B.; and Abramson, M. 2001. Executing reactive, model-based programs through graph-based temporal planning. In *Proc. of IJCAI 2001, Seattle, WA*.

Lynch, N. 1997. *Distributed Algorithms*. Morgan Kaufmann.

Mittal, S., and Falkenhainer, B. 1990. Dynamic constraint satisfaction problems. In *AAAI-1990*.

Morris, P., and Muscettola, N. 1999. Execution of temporal plans with uncertainty. In *AAAI-00*.

Nagpal, R., and Coore, D. 1998. An algorithm for group formation in an amorphous computer. In *Proc. of PDCS 1998, Las Vegas, NV*.

Stedl, J. L. 2004. A formal model of tight and loose team coordination. Master's thesis, MIT, Cambridge, MA.

Tsamardinos, I.; Muscettola, N.; and Morris, P. 1998. Fast transformation of temporal plans for efficient execution. In *AAAI-98*.

Williams, B. C.; Ingham, M.; Chung, S.; and Elliott, P. 2003. Model-based programming of intelligent embedded systems and robotic explorers. In *IEEE Proceedings, Special Issue on Embedded Software*.

Determining Task Valuations for Task Allocation

David C. Han and K. Suzanne Barber
Laboratory for Intelligent Processes and Systems
The University of Texas at Austin
{dhan, barber}@lips.utexas.edu

Abstract

The actions of agents are a reflection of the desires of their commanders (the people who design, implement, and deploy the agents). Self-interested agents will not act benevolently (taking up other agents' tasks), unless it is in their own best interests to do so. Given a set of tasks, rational agents calculate the costs and rewards for task accomplishment. The value of each task is determined in relation to (1) other tasks the agent is pursuing (and the order of accomplishment) and (2) interactions with other agents. A computational model for task evaluation is presented, constructing an overall value function out of individual rewards and costs describing the rational actions for an agent to take, effectively transforming the domain representation into a "task-oriented domain" for which protocol and mechanism design work is abundant. Additionally, algorithms for modification of the task evaluation model in response to task adoption or release are presented; enabling calculation of the marginal costs or rewards from accepting or rejecting task allocations, forming the basis for negotiation of coordination among multiple agents.

Introduction

Autonomous agents have the ability to say "no." Having autonomy means that an agent has some degree of independence from external control. Using this independence, autonomous agents have the ability to refuse tasks that are not in their best interests to perform. However, autonomous software agents do not spring into being through spontaneous generation. They are designed and deployed by some human as a proxy; to act in an environment where that human cannot or will not act. As a proxy for the human, henceforth referred to as the commander, the agent actions are determined according to the desires of the commander.

A chasm exists between the desires of the commander and domain actions. There exists a breadth of options to fill this chasm, linking the desires of the commander to goal representation, belief maintenance, planning, scheduling, and finally action. Agents can be designed to act individually or to act in concert with one another. Defining and implementing the level of coordination among agents is one of the primary responsibilities of the multi-agent system designer. If the commander deploys multiple agents, they share the desires of the commander and should be designed to intrinsically cooperate. If the multi-agent system is not a product of central design, each agent (deployed by a respective commander) is self-interested. For self-interested agents, any coordination must be pursued must be for the benefit of the self agent. Self-interested agents will not act benevolently (taking up other agents' tasks), unless it is in their own (i.e., their commander's) best interests to do so. Decision theory is well suited to performing cost/reward analysis of the various tasks to determine which ones are in an agent's best interests for both individual action and coordinated action (Boutilier 1996)(Boutilier, Dean, & Hanks 1999).

In general, an agent may not be able to achieve all of its tasks and must decide which tasks to pursue. This is a departure from the assumptions used in travelling salesman problems, where the solution includes all tasks. Instead, agents are faced with over-subscription problems (Smith 2004), and must select some subset of tasks to accomplish. Based on what their respective commanders decide, agents may consider some tasks "unprofitable." Profitability is determined by combining the desirability of a task (a reward) with the costs required to achieve that task. Profitability of a given task is dependent on (1) other tasks and (2) other agents. When tasks are aligned, an agent may benefit by a reduced cost for achieving a set of tasks compared to the sum of the individual costs of the constituent tasks. For example, combining trips to the grocer and the bank may be less resource intensive (in terms of time, gas, etc.) than making separate trips. Interactions with other agents may also change the profitability of tasks. In a competitive scenario, upon arrival at the grocer, one might find a desired ingredient out of stock due to another agent purchasing it. On the other hand, agents may often reduce their costs by reallocating tasks amongst themselves.

Due to differences among the agents in terms of resources, expertise, or even location, the costs to achieve tasks may differ among the agents. Tasks which are unprofitable for one agent may still be turned into profit through cooperation with other agents, i.e., through a subcontracting process where the rewards are shared. Cooperation is enacted through task allocation protocols, assigning tasks to individual agents. By their nature, protocols for task allocation are

based on the fundamental actions propose, accept, and reject. Proposals are necessary to inform other agents of the possibility for cooperation. Since agents must reach agreement for a task allocation to be enacted, acceptance and rejection are used to communicate agreement or disagreement with the proposed allocation.

Decisions on whether to accept or reject a given allocation are tied back to what the agent considers its best interests, or whether the allocation is deemed profitable. Depending on the interactions among the costs and rewards of the agent's tasks, profitability of an allocation should be determined in relation to the other tasks an agent holds.

In addition to the cooperative interactions, agents may interact competitively, e.g., when a task is not assigned to a particular agent but rather to a set of agents. The reward for completing the task provides an incentive for the agents to compete. Agent interactions, both competitive or cooperative, impact the profitability of tasks. This research analyzes the values and costs of tasks in relation to other tasks and agent interactions.

The next section describes the class of domains addressed in the paper. Following that, analyses for the value of tasks for an agent to maintain individual rationality are explored. A discussion of task evaluations in the context of coordinating with other agents in both cooperative and competitive settings is presented, followed by a summary and wrap-up of the paper.

Domain Characteristics

Due to the complexity of dealing with general domains, this research follows the approach taken by Lane and Kaelbling (Lane & Kaelbling 2002) and seeks (at first) not to address all domains, but rather to analyze an interesting subset of domains and build from there. Motivated by unmanned aerial vehicles (UAV) surveillance, the relevant domain characteristics are defined by the application domain. This is an interesting domain to analyze because it is easily discussed in terms of either tasks or actions and the linkage between tasks and actions is clear. In this domain, there exists a set of targets distributed around a battlefield. Each UAV, and its agent, is interested in a subset of the targets as determined by the commander of the respective UAVs. Since the commanders' interests do not necessarily overlap, there is no global utility evaluation for the system. Consequently, top-down, teamwork based formulations (e.g., COM-MTDP (Pynadath & Tambe 2002)) are not directly applicable. Instead this research follows a bottom up approach towards coordination, closer to that of GPGP (Lesser *et al.* 2004).

The UAV domain this work investigates is a "cost-to-move" domain. Each action taken incurs a cost as an abstraction of resource usage. In those states where a UAV services a target, that UAV receives some reward, the magnitude of which is related to the commander's interest in that target.

In the UAV domain, where tasks are located geographically through a space, the cost associated with a task is related to the distance an agent must travel to accomplish that task. Distance is also dependent upon the starting location of the agent. It is apparent that the order of task accomplishment affects the overall costs incurred for a set of tasks.

The agents decide which targets to service and the order in which those targets are serviced. This domain is equivalent to a postman or robot navigation domain with added flexibility in the ability to not perform unprofitable tasks. Figure 1 shows a graphic of the domain simulator used. In this figure, three UAVs are shown servicing a number of targets on the battlefield. The targets are shown with different sizes that correspond to their desirability, or the reward the agents expect to receive. The panel on the left side contains controls for the simulation.

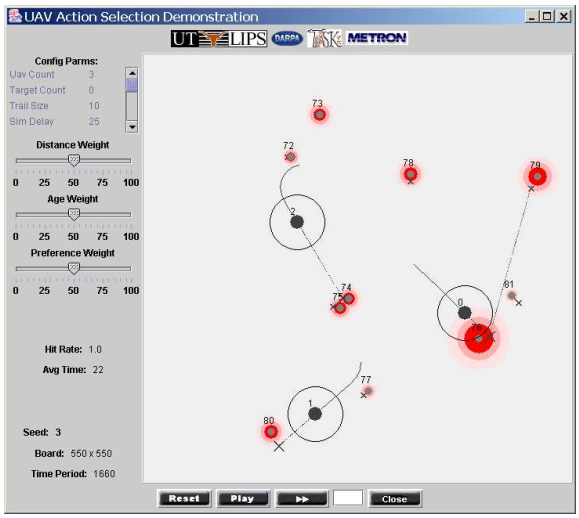


Figure 1: UAV Simulation

Many coordination approaches use abstractions at the task level through task allocation (Rosenschein & Zlotkin 1994)(Sandholm 1999). Task allocation protocols and mechanisms can take many forms; e.g., argumentation or economic models. To expedite the application of the body of work on task allocation, the UAV domain must be transformed into a form that lends itself to task allocation. TÆMS provides an expressive representation for task hierarchies (Lesser *et al.* 2004). In TÆMS, tasks are annotated with their relationships, such as "facilitates" and "hinders". Although, multiple relationships can be defined per task, TÆMS does not naturally capture the freedom of ordering of subsets of tasks or their collective affect on the values of the individual tasks. For instance, if $task_1$ can either facilitate or hinder $task_2$ depending on which other tasks the agent has selected and their order of execution, the individual relationships must be enumerated and encoded. In domains, like the UAV domain, where the interdependencies among the tasks are structured, that structure can be encoded to allow the agent to perform automated reasoning over the task relationships.

Instead, the UAV domain will be transformed into a task-oriented domain, with the following sections describing how to calculate the values of tasks based on the structure of the domain. A task-oriented domain (Rosenschein & Zlotkin

1994) is represented by a triple $\langle T, A, c \rangle$; T representing a set of tasks, A representing a set of agents, and c providing the costs of accomplishing a subset of tasks by a single agent. Two other important features of task-oriented domains are: (1) there are no negative interactions among the tasks, and (2) the tasks are interchangeable among the agents.

UAVs are tasked with servicing (an abstraction for some domain function like applying sensors to or firing upon) targets, taking into account the expected reward received for servicing the target and the cost required to move to the target location. Targets are independent, meaning that servicing one target has no effect on the reward for servicing any other target. Provided that each UAV has the ability to service any given target, tasks are interchangeable amongst the agents, satisfying the requirements of task-oriented domains.

Underlying all decisions an agent will make about which tasks to pursue is an evaluation for determining which tasks are good and which tasks are bad. For agents to properly coordinate through task allocation, each agent must be able to evaluate the costs of the subsets of tasks, the cost function $c : 2^{|T|} \mapsto \mathbb{R}$. Since the costs associated with tasks are affected by the task ordering, analysis at a lower abstraction level is required to calculate c.

Task Evaluation for Single Agent Operation

Before deciding whether to accept or reject a task allocation, an understanding must be formed about the value of the tasks involved.

Lane and Kaelbling analyze a mail delivery domain (very similar to the UAV domain) at an action level, where it is necessary that all tasks are achieved, and reduce it to a shortest path problem (i.e., the travelling salesman) (Lane & Kaelbling 2002). However, it may not always be the case that achievement of all tasks are required. Requiring that an agent satisfy all its tasks precludes the agent from declining tasks, infringing on that agent's autonomy. As an alternative, a decision-theoretic approach may prove more desirable; where the tasks are given individual reward values and the agent must reason about which tasks to pursue. Tasks are represented as consisting of two components, the state (labelled a goal) wherein the task can be executed, and the reward (desirability) the agent will receive from achieving that goal.

An important characteristic of an autonomous agent is the ability to decide which goals to pursue. Towards this end, the agent's desires may be combined in an "OR" fashion, where the agent may receive rewards for goals independently and additively. The agent must consider not only the order in which to achieve, but whether to try to achieve each particular goal at all. In practice, the cost to achieve a goal may outweigh the reward, in which case the agent should not pursue that goal. Additionally, since execution of actions can change the agent's distance to the respective goals, pursuing one goal may make it more or less profitable (even unprofitable) to pursue other goals.

A form of Markov Decision Process (MDP) is used to perform the reward/cost analysis. Macro actions can be used to combine low level domain actions into actions operating at a more abstract level, i.e., at the task level. Following the formulation of macros presented by Sutton, Precup, and Singh (Sutton, Precup, & Singh 1999), state variables can be abstracted away (factored out of the representation). By creating a macro for each goal, reasoning can be performed in terms of desire states, referred to paper as the desire space (Han & Barber 2004). Macro actions provide a bridge between the primitive actions, the building blocks for action execution, and tasks, the building blocks for reasoning about coordination. As an added benefit, macro actions encapsulate the uncertainty in the domain (e.g., non-deterministic actions) and can be treated semi-deterministically.

Figure 2 shows the desire space for three targets in the UAV domain. Each desire state is labelled with the set of goal variables, G_1, G_2, G_3, denoting which goals have not been achieved in that state. Initially, the agent is in the state marked by the full set of goals and the current location. Application of each macro leads the agent to the desire state where the appropriate goal is marked as achieved (removed from the set), leading up to the state with all goals being achieved (the empty set). Unfortunately, the complete domain space cannot be factored out because the cost function for the macro actions is dependent upon the agent's location in the domain space. Luckily, if an agent executes actions according to this decision-making mechanism, the only relevant locations in the domain space are the current location and the termination states of the macro actions (i.e., the location of the targets).

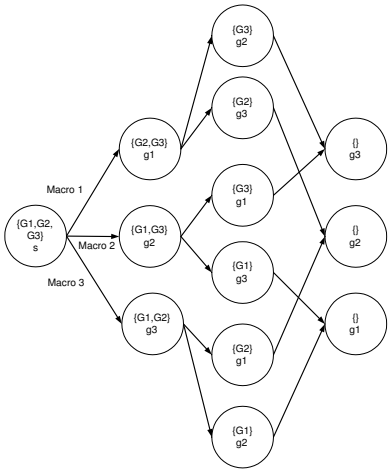


Figure 2: Desire space for the UAV domain with three targets (goals)

The motivations for reasoning in the desire space include: (1) the desire space is smaller than the complete state space (the desire space grows in the number of tasks, not the number of state variables), and (2) the structure of the desire space can be exploited algorithmically during computation. The model for reasoning about the desire space is defined as follows. Given the domain space of the problem S_{domain}, some subset of those states are marked as goals,

$$V(\langle\{\}, s\rangle) = 0 \tag{1}$$

$$V(\langle G_{unach}, s\rangle) = \max\left(0, \max_{macro_i \in A_{desire}} \begin{pmatrix} c_{action}(macro_i, s) \\ +R(g_i) \\ +V(\langle G_{unach} - g_i, g_i\rangle) \end{pmatrix}\right) \tag{2}$$

$G \subseteq S_{domain} = \{g_1, g_2, ...g_K\}$. The states of the desire space are built from the goal variables and the agent's location in the domain space. Each macro action is constructed to move the agent to a given goal state. The terminal states are represented as a probability distribution over the domain states. However, due to the nature of macro actions, the probability is concentrated on the goal state. It is possible for a macro to have termination states that represent failure of that macro to achieve its goal but, for simplicity of explanation, this paper expects the macro actions to always terminate in its goal state without fail. The desire states are denoted by a tuple $\langle G_{unach}, s\rangle$. The first element of the tuple, G_{unach} is the set of goals that have not been achieved. The second element of the tuple is the location of the agent in S_{domain}. The agent can only be located at the initial location $s_{initial}$, or as a result of executing a macro action, in an accomplished goal location g_i, hence, $S_{desire} = \{\langle G, s_{initial}\rangle, \langle G_{unach}, g_i\rangle$ s.t. $G_{unach} \subseteq G$ and $g_i \in Goals/G_{unach}\}$. The action set $A_{desire} = \{macro_1, macro_2, \ldots, macro_K\}$ is the set of macro actions, one for achieving each goal the agent holds. Finally, the reward function, $R : Goals \mapsto \mathbb{R}$, assigns a separate reward value to each goal. An action level cost function c_{action} is required to estimate the costs incurred by executing the macro action. This cost is related to the distance the agent must travel from a given domain state to the termination state of the macro.

Since the reward function is assigned slightly differently from that used in a standard MDP, i.e., the rewards are broken down according to the goals, the evaluation of states and actions is changed to match. Global termination states are those states in which there are no further profitable macro actions. States in which all goals have been achieved are global termination states since all rewards have already been collected. The global termination states (where all goals have been achieved) are assigned a value of 0, indicating that no further action will yield any reward. The expected value of desire states is defined in equations 1 and 2.

The value of a state is simply the sum of the cost of executing the macro from that state (a negative number), the reward for achieving the immediate goal through macro execution, and any expected value for being in the resulting state due to expected future goal achievement. Note that if no action is profitable (i.e., the cost of each action outweighs its benefits), then the state is also a global termination state and is given a value of 0.

Since tasks, once completed, cannot be undone in this domain, loops cannot exist in the graph. This enables calculation of the expected values to proceed through simple accumulation of the values from a single graph traversal rather than an iterative process. The nodes in the graph represent the values for achieving the various subsets of tasks, yielding the c we need to fully describe the task-oriented domain. Given an agent's starting location s, $c(T) = V(T, s)$ from the above model.

Task Evaluation for Agent Interactions

When dealing with multiple agents, the idea of task allocation comes into play. An agent, a_i, is assigned tasks, T_i, by its commander. In the system, the overall set of tasks is defined as the union of the individual agents' tasks, $T = \bigcup_{a_i inA} T_i$. If agent a_i is strictly operating independently from other agents, it only needs to calculate V for its own set of tasks, $t \in T_i$. Interaction with other agents changes the task evaluation model by either changing the allocation of tasks among the agents or by changing the values associated with the constituent tasks. Using the evaluations performed in the previous section, methods for modifying the task evaluation model to deal with interactions with other agents are presented.

Task Allocations

When each task is owned by a single agent ($T_i \cap T_j = 0$ for $i \neq j$), agents may be able to increase their profits by exchanging tasks. Since evaluations and not protocols are the focus of this research, examples are presented in the context of a simple allocation protocol. An agent can propose a contract for another agent to adopt some of its tasks following an announce, bid, award process (Smith 1980).

To maintain individual rationality, an agent should bid if and only if it is more profitable for that agent to pursue than it is not to pursue the contracted tasks. The task evaluation model is used to facilitate "what-if" reasoning, checking the value for accepting the task for profitability. By adding and removing tasks from the task evaluation model, an agent can calculate the marginal utility of a contract.

The task allocations after a contract is awarded differ from the original allocations by either adding or removing tasks. The following sections describe how to modify a task evaluation model for adoption of new tasks and the release of current tasks.

Task Adoption: Upon receiving an announcement of an allocation containing new tasks, the agent should check whether adoption of the tasks would be profitable. This is performed through "what-if" reasoning by adding the tasks into the task evaluation model. Profitability is determined by calculating the marginal value of the task shown in equation 3. For agent a_i, already owning tasks T_i, the marginal value is the difference between the value of $T_i \cup T_{add}$ and the value of T_i from the current state. If the addition of the tasks, T_{add} yield a positive marginal value for the agent, the agent

should bid on the allocation. Otherwise, the agent should reject, ignoring the announcement.

$$MU(T_{add}|T_i) = V(T_i \cup T_{add}, s) - V(T_i, s) \quad (3)$$

Since agent a_i's task evaluation model only contains values for tasks relevant to agent ($T \subseteq T_i$), the goal and reward for the announced tasks (T_{add}) must be added into the model. Addition of each goal can be handled in a single traversal of the graph.

Algorithm 1 describes a process for adding goal g to desire state d. For desire state d in the model, a new macro action is added for achieving goal g and the resulting desire state d' is created. The children of d are added to d'. After the addition of the children, the value of d' can be calculated, selecting the best macro to execute in that desire state. The new goal g are then added to each of the children of d, constructing the model while executing depth-first traversal of the tree. Finally, the value of s is updated, possibly changing the best macro to execute. Through this update process, V is maintained, rather than recalculated for each change in task structure, reusing previous computation based on the structure of the desire space.

Algorithm 1 AddGoal(d,g)

$d' =$ new STATE($\langle d.G_{unach}, g \rangle$)
$d.G_{unach} = d.G_{unach} + g$
for all $i \in d.children$ **do**
  ADDCHILD(d', i)
end for
UPDATE($V(d')$)
for all $i \in d.children$ **do**
  ADDGOAL(i,g)
end for
$d.children = d.children + d'$
UPDATE($V(d)$)

Model modification saves computational cost compared to building a new model by reusing calculations for subpaths that do not contain the new task. Figure 3 shows the result of adding g_1 to a model that already includes g_2 and g_3. Desire states marked in gray are replicated from the original model into the resulting model through ADDCHILD in the algorithm described above.

Algorithm 1 is essentially a depth-first search, but was included to illustrate how new nodes are added into the model during the search process. Heuristic usage can modify the presented algorithm to a best-first search to further reduce computational costs.

Additionally, since values are accumulated backwards, from the end of the path towards the head of the path, some desire state nodes are shown with multiple incoming edges. The value for these nodes needs only be calculated a single time, cached, then reused for each of the incoming edges. Replication saves the computational cost of recalculating the values for states which will have equivalent values to preexisting states. Rational action dictates that the agent will only pursue profitable tasks. To maintain rationality, the tail of the chosen sequence of tasks must always be profitable. For example, if the agent selects $\{g_1, g_2, g_3\}$ as a profitable sequence of tasks to pursue, then the sequence $\{g_2, g_3\}$, using the expected end state of g_1 as the starting state, must also be profitable. If this is not the case, then the sequence $\{g_1\}$ would be more profitable than $\{g_1, g_2, g_3\}$ and would be selected instead. Using this constraint, a cache of profitable sub-sequences can be built that represent the possible tails of the best sequence after addition of a goal, pruning the search space.

Task Release: By announcing a contract for a task and reward, an agent attempts to increase their profitability. To maintain individual rationality, the reward offered should be determined by the marginal utility of the task shown in equation 4. This value is the sum of rewards given to the agent for the tasks minus the value lost by not completing the tasks. An agent can profit from a contract if it pays out less for a set of tasks than the costs it would incur for completing those tasks itself. This is especially useful for tasks that are not on the most profitable course of action for the agent. In that case, since tasks T_{remove} are not included in the value $V(T_i, s)$, $V(T_i, s) = V(T_i/T_{remove}, s)$. The agent can make profit by contracting out unselected tasks for less than the reward given by the commanders.

$$MU(T_{remove}|T_i) = \begin{array}{l} \sum_{t \in T_{remove}} R(t) \\ -(V(T_i, s) - V(T_i/T_{remove}, s)) \end{array} \quad (4)$$

Goal removal also allows the agent to reduce the size of the desire space that it models. There are two cases for goal removal: (1) the goal has already been achieved and (2) the goal has not already been achieved but is being abandoned or contracted to another agent. Both cases are simple due to the structure of the desire space.

The first case is trivial due to the structure of the desire space. The agent needs only treat the current state as the new root of the model with no recalculation necessary. All desire states that are not reachable from the current desire state can be pruned from the model (e.g., all those desire states the goal being removed contributes value to). In fact, the goal variable itself can be removed from the representation. Since the value assigned to that goal variable will be equivalent for all remaining states, it can be safely factored out of the desire state representation without affecting any of the remaining desire state values. The marginal utility of removing a task whose goal has already been achieved is 0, since value is accumulated from the goals that have not been achieved.

When the goal being removed has not already been achieved (i.e., it is being adopted by another agent or abandoned), recalculation is necessary to remove the value of the goal from the action-selection reasoning. Due to the structure of the desire space (Figure 2), the value of any given node is dependent only on the unachieved goals and state of the agent at that node. $V(T_i/T_{remove})$, save for the cost of the immediate macro action to execute, has already been computed previously as an intermediate value in the calculation of $V(T_i)$. Computation is saved by caching the values of each node

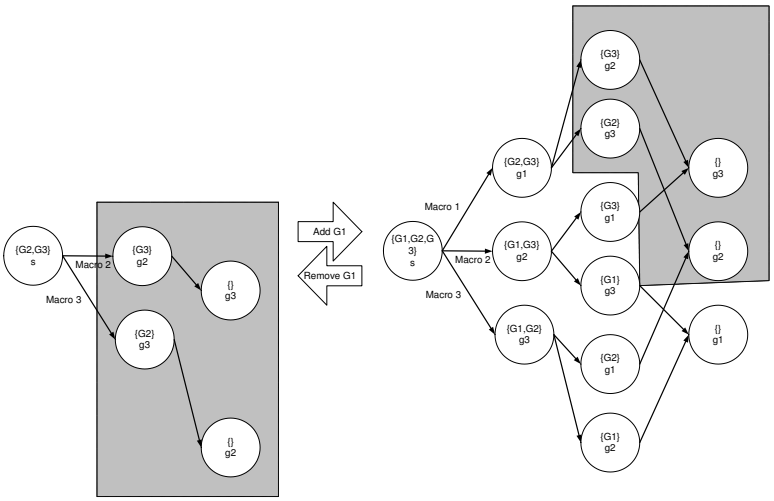


Figure 3: Modification of desire space for addition or removal of a goal

Algorithm 2 REMOVEGOAL(d,g)

$location = d.location$
$d =$CHILD(d, g)
$d.location = location$
UPDATE($V(d)$)

Algorithm 2 describes the removal of goal g. The function CHILD(d, g) selects and returns the desire state that results from executing the macro to achieve g in the desire state d. The agent transitions in the desire space as if it had achieved goal g. The resulting state in the desire space is then updated with the agent's current location in the state space. Finally, the value of the current new state is recalculated based on the new location. The values of the children states had previously been calculated, but due to the new location, the costs to reach the children have changed. This may cause a new macro to be selected as the most profitable when calculating the new $V(d)$.

Figure 3 illustrates the effect of removing goal g_1 from the desire space. The desire states highlighted in gray show the reused state values after the removal of g_1.

Competition and Cooperation

If tasks are simultaneously owned by multiple agents, those tasks are shared tasks. Depending on how the rewards are distributed for a shared task, the agents may be in either a competitive or cooperative scenario. For example, if only one agent is rewarded by its commander, two agents that share a task are in competition to receive that single reward. If rewards split amongst the agents (e.g., when the agents are deployed by the same commander, they care only for the combined value received rather than their individual rewards), the agents are motivated to cooperate. Equation 5 defines which tasks are shared by agents a_i and a_j.

$$T_{shared_{i,j}} = T_i \cap T_j \tag{5}$$

Marginal calculations can be performed for shared tasks, enabling agents to determine the value for pursuing a shared task. The rewards received for completing a task are directly influenced by the actions of other agents. Modification of equation 2 by replacing the reward $R(g_i)$ with the expected value $EV(g_i)$ enables an agent to incorporate uncertainty reasoning into the decision-making process. To improve the accuracy of the task evaluation model, it may be beneficial for an agent to use predictions or knowledge of the future actions of other agents.

Returning to the UAV domain, if two UAVs are competing to service a set of targets, then even knowledge about the location of the competing UAVs is useful. An agent may decide that targets that are closer to the competing UAV than to itself are more likely to be serviced by the competing UAV, lowering the expected value for servicing those targets. Additionally, information about the reward structure other agents hold also describe the likely targets other agents will pursue, as they will be more likely to pursue higher value targets.

Conclusion

Agents should strive to perform actions that are in their, and their commander's, best interests. This work addresses the problem of evaluating tasks, determining which tasks the agent should pursue individually and providing a computational model for task allocation in coordinated action. The value of a task is dependent upon the agent's state and the tasks the agent is already pursuing. This paper describes the desire space, a task evaluation model for computing the value of subsets of tasks by combining the costs and rewards of the constituent tasks.

The ability to say "no" is central to the idea of autonomy. Usage of this task evaluation model preserves individual rationality by calculating which subset of tasks is most profitable, allowing the agent to decide which tasks to pursue and not pursue. The task evaluation model provides infor-

mation about the profitability of tasks in relation to each other. It takes into account the location of the agent when calculating costs, and the order in which tasks are achieved.

Negotiation for task allocation depends on the agent knowing the value of its tasks. Using this model, profitability analysis can be performed for negotiating the adoption of a task from another agent in cooperative scenarios as generating expected values based on knowledge of other agents' behavior for competitive scenarios. Marginal utility calculations, generated by modifying the task evaluation model through goal addition and removal, are used to determine the profitability of forming contracts with other agents. In the case that a task is not on the most profitable path, meaning that the agent will choose not to pursue it, the agent is then free to negotiate for other agents to adopt that task. An agent can receive some value by subcontracting tasks out to agents for whom those tasks might be less costly to complete.

Future research directions include the extension of this work to include more complex domain assumptions, including conflicting tasks, deadlines, and other task interactions.

Acknowledgements

This research was funded in part by the Defense Advanced Research Projects Agency and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-00-2-0588. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed on implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory, or the U.S. Government.

The research reported in this document was performed in connection with Contract number DAAD13-02-C-0079 with the U.S. Edgewood Biological Chemical Command.

References

Boutilier, C.; Dean, T.; and Hanks, S. 1999. Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research* 11:1–94.

Boutilier, C. 1996. Planning, learning and coordination in multiagent decision processes. In *Theoretical Aspects of Rationality and Knowledge*, 195–201.

Han, D., and Barber, K. S. 2004. Desire-space analysis and action selection for multiple, dynamic goals. In *CLIMA V, Computational Logic in Multi-agent Systems*, 182–195.

Lane, T., and Kaelbling, L. P. 2002. Nearly deterministic abstractions of markov decision processes. In *Eighteenth National Conference on Artificial Intelligence (AAAI2002)*, 260–266.

Lesser, V.; Decker, K.; Wagner, T.; Carver, N.; Garvey, A.; Horling, B.; Neiman, D.; Podorozhny, R.; NagendraPrasad, M.; Raja, A.; Vincent, R.; Xuan, P.; and Zhang, X. 2004. Evolution of the GPGP/TAEMS Domain-Independent Coordination Framework. *Autonomous Agents and Multi-Agent Systems* 9(1):87–143.

Pynadath, D. V., and Tambe, M. 2002. The communicative multiagent team decision problem: Analyzing teamwork theories and models. *Journal of AI research* 16:389–423.

Rosenschein, J. S., and Zlotkin, G. 1994. *Rules of encounter: designing conventions for automated negotiation among computers.* MIT Press.

Sandholm, T. W. 1999. Distributed rational decision making. In Weiss, G., ed., *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. Cambridge, MA, USA: The MIT Press. 201–258.

Smith, R. G. 1980. The contract net protocol: High-level communication and control in a distributed problem-solver. *IEEE Transactions on Computers* 29(12):1104–1113.

Smith, D. E. 2004. Choosing objectives in over-subscription planning. In Zilberstein, S.; Koehler, J.; and Koenig, S., eds., *ICAPS*, 393–401. AAAI.

Sutton, R. S.; Precup, D.; and Singh, S. P. 1999. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence* 112(1-2):181–211.

Planning for Multiagent Environments

From Individual Perceptions to Coordinated Execution

Michael Brenner
Institute for Computer Science
Albert-Ludwigs-University
79110 Freiburg
Germany
brenner@informatik.uni-freiburg.de

Abstract

In this paper, we discuss the particular characteristics of planning for multiagent systems, and present a rich formal model for describing features like concurrency, individual and mutual beliefs of agents, acting under incomplete knowledge, control, perception, and communication. Our model allow agents to execute their individual plan fragments as autonomously as possible while provably guaranteeing synchronized behavior where necessary. Synchronization can be achieved by as different methods as communication, metric or quantitative temporal constraints, or copresence. We show the semantic relation of multiagent plans to classical plans, and informally describe a sound and complete variant of a POCL algorithm for multiagent planning.

1 Introduction

In this paper, we discuss the particular characteristics of planning for multiagent systems, and present a rich formal model for describing features like concurrency, individual and mutual beliefs of agents, acting under incomplete knowledge, control, perception, and communication. While previous research has acknowledged most of these characteristics to be relevant for multiagent planning (MAP), we are not aware of prior work modelling and integrating all of them and, above all, giving them a clear formal semantics that can be used to prove properties of both plans and planning algorithms.

Our model allow agents to execute their individual plan fragments as autonomously and flexibily as possible while provably guaranteeing synchronized behavior where necessary. Synchronization can be achieved by as different methods as communication, metric or quantitative temporal constraints, individual perception or copresence.

The purpose of this article is unusual in so far that it does not contain algorithmical or empirical results, but "only" provides a thorough discussion of MAP characteristics and, consequently, a thoroughly defined logical model that allows, e.g., to prove that a multiagent plan can by executed by multiple agents without further coordination or external synchronization. We believe that only based on such formal qmodels, there can be theoretical, algorithmical, and empirical progress in MAP. As one tool for this development we have designed a variant of PDDL for the semantics defined in this article. A parser and several sample domains (including the one presented in the next section) will be made available for download.

The next section will motivate and discuss the concepts formalized in the remainder of the article. At the end of the paper, we also sketch a sound and complete algorithm for planning in our formalism.

2 Motivation

Example 1 *A person wants to visit a friend. The friend's house can only be entered once its door has been opened.*

Consider the scenario described in Example 1. We can model it using the simple STRIPS-like operators given below. If we assume $closed \wedge outside \wedge \neg atHouse$ as the initial state of the world (intuitively stating that the door is closed and that the visitor has not yet reached the house) the following is a valid STRIPS plan for the scenario: $\langle$*move2house; open; enter*$\rangle$.

action	**precondition**	**effect**
move2house	$outside \wedge \neg atHouse$	$atHouse$
open	$closed$	$\neg closed$
enter	$outside \wedge atHouse \wedge \neg closed$	$\neg outside$

As the reader may have noticed, we have tried to obfuscate an important aspect of the scenario both in the verbal and the formal description, namely *who* is performing which action. In fact, in classical STRIPS-like planning there is no direct way to model the *agent* of an action (even if telling action names suggest a specific reading). This is unproblematic for Classical Planning which assumes centralized control of plan execution, but for MAP one must at least distinguish the different capabilities of different agents. Most prior MAP formalizations have recognized this and allow actions to be associated with an *executing* or *controlling agent*. For our example, let us assume that the visitor x can move to the house and enter it, but that only her friend y can open the door. Then, most existing MAP formalisms would accept the following as a valid plan for the scenario (where each action is annotated with the controlling agent): $\langle move2house_x; open_y; enter_x \rangle$.

However, the main point we will elaborate in the rest of the paper is that this plan may actually not be executable by the two autonomous agents! The reason is that the plan

constrains two autonomous agents to a specific temporal order of actions, but does not guarantee that they can actually *synchronize* their behavior accordingly. Note that this is not merely a problem of total-order (TO) vs. partial-order (PO) plans: even a less constrained PO representation like $\{move2house_x \prec enter_x, open_y \prec enter_x\}$ demands agent x to synchronize her *enter* action with a previous *open* action by another agent. How is this synchronization achieved?

The example is so simple that the solution seems obvious: x *knows* when she can enter the house because she can *perceive* the door to be open once she is outside the house. However, semantically there is no relation between the proposition $outside$ describing where x is and the proposition $closed$ describing what she is supposed to perceive. There is not even a distinction between the door being open and x being aware of it. The key to synchronization thus lies in modeling not only the state of the world, but also the beliefs that agents may have about it. Since facts can not only be believed to be true or false, but also *unknown*, we use multi-valued state variables in our framework instead of propositions. This has the additional advantage of leading to smaller state spaces. In our scenario, the propositions *outside* and *atHouse* could be fused to one state variable loc_x with possible values *inHouse, nearHouse,* and *elsewhere* (plus *unknown* in case of beliefs), thereby eliminating the combination $\neg atHouse \wedge \neg outside$ which is meaningless in our scenario.

Actually, many existing MAP formalisms, e.g. SharedPlans (Grosz & Kraus 1996), also employ beliefs and even mutual beliefs among agents. However, to the best of our understanding these concepts are only used there to describe beliefs *about* a plan (e.g. mutual beliefs about the joint commitment to the plan), but not for describing how (mutual) beliefs about the world develop and change *in* the plan. The most basic way to change one's beliefs is through *perception*. (In Section 5 we will also describe communication.) We model perceptions by *sensor rules* that are automatically triggered when the necessary conditions are satisfied. The key realization underlying this approach is that perception is not a consequence of one single action, but is an event emerging from, firstly, something happening and, secondly, somebody being there to watch it. In this sense, perception is a special case of concurrency. Indeed, we will use other kinds of *domain rules* to describe non-trivial effects of concurrent events, as in the following variation of our scenario:

Example 2 *Person y lives in a multi-storey building where she can operate the entrance door by a buzzer. x can only enter the house while the door is temporarily unlocked by the buzzer. Furthermore, x must ring the doorbell first to notify y that she is there.*

This example is fairly common in reality and introduces a number of new aspects. Firstly, actions must be performed concurrently in this scenario to achieve a nontrivial joint effect, namely x must push the door *while* the buzzer is activated to cause the door to swing open. Boutilier and Brafman (2001) show how such concurrent interacting actions can be modelled using concurrency constraints and special conditional effects. The authors note that post-planning synchronization will be necessary to ensure the concurrency constraints are obeyed by the executing agents. Since in our model the plan itself is intended to guarantee synchronization, we present an alternative model here which we call the *physical forces approach* to concurrency. The underlying idea is that action have only individual effects that do not directly interact, but which in combination may create a kind of "force" or "instability" that causes a natural event according to the *causal rules* of the domain. Fig. 1 shows the Causal Domain Rule (CDR) of our scenario in PDDL-like syntax. The rule is modeled as an action caused by the unique agent env representing the environment.

```
(:action swing-open
 :agent env
 :precondition (and (doorstate = pushed)
                    (buzzer))
 :effect (doorstate = open))
```

Figure 1: Causal Domain Rule

When synchronization by means of sensing is not possible, an alternative may be to agree on absolute time points for action execution. This is just one reason for including metric time in our model. For most practical problems featuring concurrency it is important to reason not only about whether some actions can be parallelized, but also about the quantitative relation between their durations. However, we also want to describe flexible or unknown durations; to ensure synchronized behavior in that case of uncertainty we must be able to reason about the qualitative relations between events. The temporal model used in this paper allows to describe both qualitative and quantitative temporal relations between events.

Example 3 *The door to y's house often stands open. If this is the case when x arrives she can simply walk in instead of ringing.*

One major reason for the difficulty of multiagent planning is the high dynamics of MAS and, consequently, the many facts that may be unknown at *planning time* – even if the planner in question centralizes knowledge of several executing agents. Usually, however, many things unknown at planning time will become perceivable to at least one executing agent at *execution time*. This fact can be exploited by a planner, since perception models form an explicit part of our model, and plans can thus include actions for *active knowledge gathering*. However, since the actual perceptions to be made are unknown at planning time, *conditional action execution* must be possible depending on the outcome of the perception. In Ex. 3, x must first move to the entrance of y's house to perceive the state of the door (open or locked) before she can decide whether she can simply walk in or must ring first. A multiagent plan for Ex. 3 is shown in Fig. 2; the sensor rule describing the circumstances under which the agent can perceive the state of the door is shown in Fig. 3 in the extended PDDL syntax we have defined for our model.

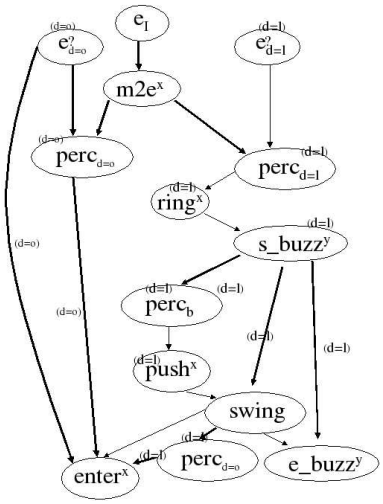


Figure 2: A multiagent plan for scenario 3.
For clearness, temporal constraints and facts supported by causal links have been omitted. Labels $d = l$ ($d = o$) denote the door being initally locked (open). Events labeled $perc$ are perception rules, the CDR $swing$ corresponds to the door swinging open when being pushed and kept unlocked (by buzzing) simultaneously.

3 Integrating Agency with Planning

In this section, we will describe how basic notions of agency can be integrated into a Planning formalism. We will, however, not attempt a definition of what constitues an "agent". Instead, an unspecified set of agents $\mathcal{A}$ will be the basic building block for all further definitions. $\mathcal{A}$ is always assumed to include the unique agent env, the **environment agent** with special characteristics described in Sec. 5.

Some necessary components of agency (like beliefs and capabilities for sensing and acting) will be defined and attributed to agents $a \in \mathcal{A}$. For this we use a function $agt(x) := a$ where x can be any such component. We will use the index notation x_a to denote $agt(x) = a$ and also extend this notation straightforwardly to sets.

Facts, beliefs, and mutual beliefs Instead of the propositional representation used in most Planning formalisms, our model uses **non-boolean state variables** (cf. (Bäckström & Nebel 1995; Helmert 2004) for a discussion of the SAS$^+$ formalism where this extension is borrowed from). There are several reasons for this design choice:

1. Multi-valued state variables occur naturally in most planning domains. For example, the positions of an agent a can be encoded by one state variable loc_a with a set of possible values, the **domain** of loc_a, dom_{loc_a}. Not only becomes modeling such domains considerably easier, also the size of the state space can often be dramatically reduced (Helmert 2004). Moreover, since propositions are state variables over the domain {true,false}, propositional planning formalisms like STRIPS, ADL and PDDL are subsumed by state-variable model, anyway. Syntactically, compatibilty with propositional planning can be maintained by allowing the notations `(prop)` and `(not (prop))` instead of `(prop = true)` and `(prop = false)`.
2. *Beliefs* of agents can straightforwardly be modeled, without the need for a possible world semantics, by simply allowing state variables to assume a specific additional value *unknown*[1]. We will call such variables **belief state variables**.
3. Distributed Systems are usually modelled by means of private and shared variables. Classical concepts like *read-write conflicts* or variable *locks* can be easily recognized in multiagent plans when using a state variable representation. For example, the Classical Planning concept of *mutually exclusive* propositions (Blum & Furst 1997) can then be expressed as *read-write locks* between state variables. This shift in perspective is helpful especially when applying Distributed Algorithms concepts to Multiagent Planning (Brenner 2003).

Let $\mathcal{V}$ be a set of **state variables**, each $v \in \mathcal{V}$ with an associate finite **domain** dom_v. A **partial variable assignment (PVA)** over $\mathcal{V}$ is a function s on some subset of $\mathcal{V}$ such that $s(v) \in dom_v$ wherever $s(v)$ is defined. undef_s is the set of **undefined variables** in s. If $s(v)$ is defined for all $v \in \mathcal{V}$, s is called a **state**. If $s(v)$ is defined (with value x) then the pair (v, x) is called an **assignment** (also written $v \doteq x$). Two PVAs s and s' are called **consistent** if the following holds: if both $s(v)$ and $s'(v)$ are defined then $s(v) = s'(v)$.

STRIPS, PDDL, and other languages based on propositional logic use sets of propositions where our model (due to having non-boolean variables) must use PVAs. To make this relation easier to see, we will often use set notations for PVAs, too, e.g. we write $(v \doteq x) \in pre_e$ instead of $pre_e(v) = x$, and denote the completely undefined PVA by $\emptyset$. In particular, we define the **union** of two *consistent* PVAs s_1 and s_2 as the PVA $s = s_1 \cup s_2$ in which if $s_1(v) = x$ or $s_2(v) = x$ then also $s(v) = x$.

For a given agent $a \in \mathcal{A} \setminus \{\mathsf{env}\}$, a set $\mathcal{V}$ of state variables induces a set of **belief state (BS) variables** $\mathcal{V}_a$ where for each $v \in \mathcal{V}$ there is a $v_a \in \mathcal{V}_a$ with $dom_{v_a} = dom_v \cup \{\mathsf{unknown}\}$. The function agt, defined as $agt(v_a) := a$, returns the **owner** of a BS variable. The environment env does not have beliefs; to keep some of the following definitions simple, we define $\mathcal{V}_{\mathsf{env}} := \mathcal{V}$ and $agt(v) = \mathsf{env}$ for $v \in \mathcal{V}$.

We can further generalize this concept to beliefs shared among subgroups of $\mathcal{A}$: the sets $\mathcal{V}$ and $\mathcal{A}$ induce a set of **mutual belief state (MB) variables** $\mathcal{V}_\mathcal{A}$ where for each $v \in \mathcal{V}$ and each subgroup $A \subseteq \mathcal{A}$ there is a $v_A \in \mathcal{V}_\mathcal{A}$ with $dom_{v_A} = dom_v \cup \{\mathsf{unknown}\}$. The definition of MB variables includes mutual belief among a singleton set of agents which is equivalent to individual belief. Thus the definition

[1] It must be noted that some beliefs can be expressed within a possible world semantics, but not in our model, in particular constraints *between* state variables. For example, the constraint $loc_a \doteq x \leftrightarrow loc_b \not\doteq x$ could describe that no two agents can be believed to be at the same position at the same time. Often, though, such constraints can be modeled by introducing complementary state variables, for example $occupant_x$ would describe who is standing at position x and could have value a or b (plus some dummy unoccupied), but not both.

of MB variables subsumes the one for BS variables of individual agents. For convenience, however, we will keep the distinction between individual and mutual beliefs, and also continue to use the notation $v_a := v_{\{a\}}$ for individual beliefs. Since furthermore $\mathcal{V}_{\mathsf{env}} = \mathcal{V}$, the PVAs over $\mathcal{V}_{\mathcal{A}}$ enumerate all possible states, beliefs, and mutual beliefs for a given domain[2]. A PVA s is **knowledge consistent** if all mutual beliefs correspond to the facts, i.e. they are actually **common knowledge**. Formally, a PVA s is knowledge consistent if $s(v_A) = x$ implies that also $s(v) = x$ for all variables v and all $A \in \mathcal{A}$. In particular, knowledge consistency implies **MB consistency**, i.e. if $s(v_A) = x$ then $s(v_{A'}) = x$ for all $A' \subseteq A$.

4 Modelling Multiagent planning domains

We can now define what constitutes a MAP domain. While the definition contains many agent-specific particularities that will be explained in the rest of this section, it was nevertheless designed to be compatible wherever possible with PDDL 2.1 (Fox & Long 2003). Roughly, our definitions extend PDDL 2.1, level 1 and 3, and by "compatibility" we mean that there is a large class of domains that are both MAP and PDDL domains. The main difference, apart from the notions of agency described in the previous sections, is a temporal model that allows more "qualitative" relations between events than the solely metric time model of PDDL 2. For a similar treatment of time, see (Younes & Simmons 2003). We have discussed the importance of a "qualitative" temporal framework in addition to a quantitative one like PDDL 2 in (Brenner 2003).

Definition 1 *A **multiagent planning domain** is a tuple $\mathcal{D} = (\mathcal{A}, \mathcal{V}, \mathcal{E}, \mathcal{O})$ where*

- *$\mathcal{A}$ is the set of **agents***
- *$\mathcal{V}$ is the set of **state variables**, each $v \in \mathcal{V}$ with an associate finite **domain** dom_v*
- *$\mathcal{E}$ is the set of **events**, each $e \in \mathcal{E}$ of the form $e = (a, pre, \mathit{eff})$ where*
  - *$a \in \mathcal{A}$ is the **controlling agent***
  - *pre is a knowledge consistent PVA over $\mathcal{V} \cup \mathcal{V}_a$ called the **precondition***
  - *eff is a knowledge consistent PVA over $\mathcal{V} \cup \mathcal{V}_{\mathcal{A}}$ called the **effect** of e.*
- *$\mathcal{O}$ is the set of **processes**, each $o \in \mathcal{O}$ of the form $o = (a, e^s, e^e, \Delta, inv)$ where*
  - *$a \in \mathcal{A}$ is the **controlling agent***
  - *$e^s \in \mathcal{E}$ (with $agt(e^s) = a$) is the **start event***
  - *$e^e \in \mathcal{E}$ (with $agt(e^e) \in \{a, \mathsf{env}\}$) is the **end event***
  - *inv is a PVA over $\mathcal{V} \cup \mathcal{V}_a$ called the **process invariant***
  - *the interval $\Delta \subseteq \mathbb{R}^+$ is called the **duration range** of o*

[2]The reader will note that we only define individual and mutual beliefs, but do not attempt to model arbitrary nested beliefs (like "A believes that B believes that C believes that x"). Firstly, this would lead to an infinite number of nested belief variables. Secondly, nested beliefs (other than mutual beliefs) almost never seem to play a role in multiagent behavior. Thirdly, if, for specific problems or domains, beliefs nested to some finite level were needed, the model could easily be extended.

Events Roughly, events corresponds to instantaneous actions in PDDL. We use the neutral term "event" to hint at the fact that what for one agent constitutes an action that she can execute at will is an uncontrollable event for another.

Events differ from classical actions in having *knowledge preconditions* and *knowledge effects*. Interestingly, knowledge preconditions are more restricted than knowledge effects. The reason for this is that the controlling agent can only refer to her own beliefs when checking whether she can execute an action. In contrast, we allow agents to *change* the beliefs of other agents directly, at least in principle: this is the most basic way to model *communication*, i.e. actions with knowledge effects can be regarded as speech acts.

We have already seen that for an action to be executable by an agent a not only must its usual preconditions be satisfied, but the agent a must also know about it[3]. We therefore demand the following for all $e_a \in \mathcal{E}$: if $(v \doteq x) \in pre(e_a)$ then also $(v_a \doteq x) \in pre(e_a)$. For effects, we will enforce a similar constraint: if $(v \doteq x) \in \mathit{eff}(e_a)$ then also $(v_a \doteq x) \in \mathit{eff}(e_a)$. The meaning, however, is somewhat different: an agent will know when it has executed an action and therefore will believe in its effects to have occured. Of course, both kinds of constraints can be automatically computed and need not be specified explicitly.

Processes Processes are similar to durative actions in PDDL, but must be extended for MAP with the notion of **control** which was introduced by Vidal and Fargier (1999) and is extended to the multiagent case here. Control describes the kind of influence that an agent has on a process. For some process o where $a = agt(e^s)$ we say that a has **occurrence control** over o. If, additionally, $a = agt(e^e)$ then a also has **duration control** over o. The key semantic difference can be illustrated by the two processes of reading a book and boiling water with a kettle: I can decide for both processes whether I want to execute them in my plan (and thus have occurence control over both), but I have duration control only of my reading the book, i.e. I can tighten duration interval Δ at will in my plan. In contrast, for boiling the water the plan must be guaranteed to work for all possible durations $\delta \in \Delta$.

The process invariant inv is used just as in PDDL to describe facts that must hold throughout the whole process. To model this semantics an artifical event $e^{inv} = (\mathsf{env}, inv, \emptyset)$ will be used. e^{inv}, e^s, and e^e together form the set $\mathcal{E}^o$ of events appearing in process o. The set of all events appearing in a set of operators O is denoted $\mathcal{E}^O$.

To simplify our later definition of multiagent plans, we model instantaneous actions as processes, too: if $e_a \notin \mathcal{E}_{\mathcal{O}}$ then we extend $\mathcal{O}$ by $(a, e_a, e_a, \emptyset, [0, 0])$.

Semantics of events The semantics of events is defined exactly as in other planning formalisms: given a state s and an event e, e is **applicable** in s if whenever $(v \doteq x) \in pre_e$ then also $(v \doteq x) \in s$. Applying an applicable event e in a state s results in state $app(s, e)$ where $(v \doteq x) \in app(s, e)$ iff $(v \doteq x) \in \mathit{eff}_e$ or $[(v \doteq x) \in s$ and

[3]Although sometimes one may want to give up this constraint, resulting in a "leap-before-you-look" approach (Golden 1998).

$v \in \mathsf{undef}_{eff_e}]$. The occurence of a **sequence** of events can be defined inductively in the usual manner: $res(s, \langle\rangle) := s$ and $res(s, \langle e_1, ..., e_n\rangle) := app(res(s, \langle e_1, ..., e_{n-1}\rangle), e_n)$ if e_n is applicable in $res(s, \langle e_1, ..., e_{n-1}\rangle)$, otherwise $res(s, \langle e_1, ..., e_n\rangle)$ is undefined. We will later show how this Classical Planning semantics relates to our complex temporal multiagent plans.

5 Modeling Causal Laws of MA Systems

The events and processes controlled by the environment agent env differ from those of all other agents in one crucial aspect: the environment does not act deliberately and willfully; instead events necessarily occur according to the "physical laws" of the domain, its **causal domain rules** (CDRs). Formally the CDRs simply consist of all processes $\mathcal{O}_{\mathsf{env}}$ controlled by env. The semantic difference is the fact that preconditions of normal actions describe conditions *necessary* for an action to be exectubable, while conditions of CDRs are *sufficient* to trigger the corresponding event or process. (The concept of automatically triggered events was inspired by research in the Theory of Actions community, in particular by Thielscher (1995)).

Causal domain rules are meant to model the "laws of nature" of a domain. Whenever a rule is triggered the world is considered to be in an *unstable* state leading to an event or the start of a process which in turn removes the instability (but might create a new one). To capture that aspect and to prevent the same rule to be triggered repeatedly with infinitesimal delays, we enforce rules to destroy their own triggering conditions. $pre(e^s)$ and $\mathit{eff}(e^s)$ must be inconsistent, i.e. e^s destroys one of its preconditions.

Furthermore, two rules $r_1, r_2 \in \mathcal{O}_{\mathsf{env}}$ that are triggered by the same situation could have inconsistent effects, thereby introducing nondeterminism into our model. Just as in Classical Planning we will forbid this, and formally constrain: If $\mathit{eff}(r_1)$ and $\mathit{eff}(r_2)$ are inconsistent, then $pre(r_1)$ and $pre(r_2)$ must be inconsistent, too.

Within the constraints just defined (destroying the own precondititon, no rules leading to nondeterminism), causal domain rules are a powerful tool. In particular, we can use them to naturally model interactions between concurrent actions as was demonstrated in Ex. 2. Due to lack of space, a detailed comparison to alternative models of concurrency like the one of Boutilier and Brafman (2001) will be done in an extended version of this paper.

Perception, communication, and mutual belief

Causal domain rules are also used to describe sensor models of agents. In contrast to other "physical laws" of a domain, **perception rules** have *knowledge effects*. To simplify reasoning about perception rules we will enforce the following format for them: perception rules $r \in \mathcal{O}_{\mathsf{env}}$ must be instantaneous actions, i.e. $r = (\mathsf{env}, e, e, \emptyset, [0, 0])$. Furthermore e has exactly one effect $(v_a \doteq x)$ where $a \in \mathcal{A}$ and $(v \doteq x) \in pre_e$. We call $pre_e \setminus \{(v \doteq x)\}$ the **perception condition** for $(v \doteq x)$. Usually, we can assume that perception does not depend on a specific value x of v and that there are corresponding rules for all $x \in dom_v$. The set of these rules is called a **sensor model** and we write $\mathsf{sensor}(a, v, cond)$ to denote that for all $x \in dom_v$ there is a rule $(\mathsf{env}, cond \cup \{(v \doteq x)\}, \{(v_a \doteq x)\})$. Fig. 3 shows how the sensor model $\mathsf{sensor}(a, doorstate, \{loc_a \doteq entrance\})$ is described in PDDL-like syntax. It specifies that an agent will perceive the state of the door (*open* or *closed*) when she is at the entrance.

```
(:sensor door-sensor
 :agent ?a
 :precondition (loc ?a = entrance)
 :sense (doorstate))
```

Figure 3: Perception rule

We have previously explained how knowledge effects can be used as an easy means to model speech acts. Additionally, we have assumed that an agent executing an action will believe its effect to be true afterwards. In combination, those premises lead to an interesting effect. Assuming that agent a communicates a fact $p = (v \doteq x)$ to agent b, the effect $v_b \doteq x$ could be expressed as $B_b p$ in some standard epistemic logic. However, since a knows this to be the effect of his action also $B_a B_b p$ will be true. We have explicitly not included such nested beliefs in our framework, but we can do something else: If me make the additional assumption (not yet explicit in the semantics) that b will know who has communicated p to her, she will be able to infer $B_b B_a B_b p$, which in turn a may infer, etc. In short, under the assumption of perfect communication and speaker detection, our modeling of speech acts induces mutual belief. This is not surprising (Fagin *et al.* 1995), yet welcome, since it allows us to replace simple knowledge effects with mutual belief effects (among the speaker and hearer) in speech acts.

Communication is not the only way to achieve mutual belief. Another possibility, *copresence* (or coperception) was described already by Lewis (1969). Informally, agents are copresent when they are in a common situation where they can not only perceive the same things but also each other. Such a situation can lead to mutual belief since the agents can mutually infer their perceptions, the beliefs about other agents' perceptions, etc.

We can describe copresence situations as special kinds of sensor models $\mathsf{sensor}(A, v, cond)$ that have effects on a mutual belief variable v_A for a group of agents A. A basic example could be a copresence model stating that agents achieve mutual belief about their respective locations whenever those are identical. Based on this "precursory" MB more MB can be inferred wherever a perception rule is triggered the condition of which does not only hold, but is already mutual belief. In that situation all copresent agents could infer the perceptions of the others, plus their inferences, etc. A more formal treatment of this topic will be given in a future publication where we also describe an approach to automatically deriving copresence models from individual sensor models.

6 Plans, Problems, and Solutions

Definition 2 *A **multiagent plan** for a domain $\mathcal{D} = (\mathcal{A}, \mathcal{V}, \mathcal{E}, \mathcal{O})$ is a tuple $P_\mathcal{D} = (A, O, T, L, B)$ where*

- $A \subseteq \mathcal{A}$ *is the set of **agents***
- $O \subseteq \mathcal{O}$ *is the set of **operators***
- T *is a set of **temporal constraints** of the form $t = (e, e', \Delta)$ where $e, e' \in \mathcal{E}^O$ and $\Delta \subseteq \mathbb{R}$.*
- $L = L^+ \cup L^-$ *is a set of **positive** and **negative causal links** of the form $l = (e, v \doteq x, e')$ where $(v \doteq x) \in pre(e')$ and*
  - *$(v \doteq x) \in \mathit{eff}(e)$ if $l \in L^+$*
  - *$(v \doteq x') \in \mathit{eff}(e)$ if $l \in L^-$ (for some $x' \neq x$)*
- B *is a function labeling each event and each causal link with a PVA. It is called the **branching context**.*

This definition of MA plans is related to single-agent formalisms for conditional temporal planning (Tsamardinos, Pollack, & Horty 2000), but extends prior work with multiple agents, causal domain rules, and (mutual) beliefs. To show the relation to classical PO representations of plans, we say an event e is **precedes** another one e' if $(e, e', \Delta) \in T$ and $\Delta \subseteq \mathbb{R}^+$. In that case, we also write $(e \prec e) \in T$.

In the following, we will assume that in every given plan P the set of constraints T is complete and unambiguous, i.e. that there is exactly one constraint (e, e', Δ) for all $e, e' \in \mathcal{E}^O$. This is no restriction, but can be achieved easily by extending T with $(e, e, [0, 0])$ for all events $e \in \mathcal{E}^O$ and with $(e, e', (-\infty, \infty))$ for previously unrelated events $e \neq e$. We further assume that T is pairwise consistent, i.e. $(e, e', \Delta) \in T$ iff. $(e', e, -\Delta) \in T$. If $(e, e', \Delta) \in T$ and $\Delta \subseteq \mathbb{R}^+$, we say that e must occur **before** e' and write $(e \prec e) \in T$.

The temporal constraints T of a plan P form a Simple Temporal Network (STN) (Dechter, Meiri, & Pearl 1991). A basic prerequisite for giving the plan a meaningful semantics is that the underlying STN is **consistent**. Consistency can be checked in small polynomial time; cf. (Younes & Simmons 2003) for a description of how STNs can be used in a state-of-the-art single-agent planner. In the following we will assume only plans with consistent underlying STN.

Temporal constraints in a plan must not only form consistent STNs, they must also not violate the duration range defined for the processes $\mathcal{O}$ as defined in $\mathcal{D}$. The duration range, however, has a different semantics depending on who has *duration control* of a process o: if env controls the duration, the plan must be be valid for any possible duration in the duration range. If, on the other hand, the agent controlling the duration of o is different from env the planner may tighten the duration constraints at will[4]. Formally, we define a plan $P_\mathcal{D}$ to be **process-consistent** with $\mathcal{D}$ if for all processes $o = (a, e^s, e^e, \Delta, inv)$ in $P_\mathcal{D}$, $T \supseteq \{(e^s, e^e, \Delta_1), (e^s, e^{inv}, \Delta_2), (e^{inv}, e^e, \Delta_3)\}$ where $\Delta_1, \Delta_2, \Delta_3 \subseteq \Delta$. If $agt(e^e) = \mathsf{env}$ then $\Delta_1 = \Delta$.

Following the literature on conditional planning we demand that B must be defined such that labels are propagated along temporal constraints in the plan (Peot & Smith 1992; Tsamardinos, Pollack, & Horty 2000; Tsamardinos, Vidal, & Pollack 2002).

The reader may have noted that, in contrast to, e.g, PDDL 2, there is no way in our formalism to specify absolute time points for events. However, absolute time points can be described by referring to a special, mutually known **temporal reference event** e_0, virtually occuring before all other actions. Note, however, that in many domains exact time points will only complicate plan monitoring, since in general it cannot be determined whether a plan should still be considered correct when some event occured with a slight temporal difference to its precise scheduled time. The qualitative model we propose is thus more flexible than the metric one of PDDL 2.

We can now generalize the POCL notions of threats and open conditions to our metric temporal and conditional plan formalism.

Definition 3 *In a plan $P = (A, O, T, L, B)$, an event e has an **open condition** $c = (v \doteq x)$ if $c \in pre(e)$ and there is no causal link $l \in L$ which supports c, i.e. which is of the form $l = (e', c, e)$ for some e'. An event $e_t \in \mathcal{E}^o$ **threatens** a causal link $l = (e_p, v \doteq x, e_c) \in L$ if*

- *e_t has an effect $v \doteq x'$ where $x' \neq x$ if $l \in L^+$, and $x' = x$ if $l \in L^-$*
- *e_t might occur between e_p and e_c, i.e. there exist $\Delta, \Delta' \subseteq \mathbb{R}^+$ for which $T \cup \{(e_p, e_t, \Delta), (e_t, e_c, \Delta')\}$ is consistent*
- *e_p and e_t occur in consistent branching contexts*

Natural events *necessarily* follow the causal rules defined for the domain. As a consequence, valid plans must not only contain actions that achieve the goals, but must also ensure that no harmful natural events can be triggered. An operator o is said to **enable** a rule r if it achieves some trigger condition of r. Formally, we define a relation $\mathsf{enables} \subseteq \mathcal{O} \times \mathcal{O}_\mathsf{env}$ where $\mathsf{enables}(o, r)$ if there exists $(v \doteq x)$ with $(v \doteq x) \in \mathit{eff}(o)$ and $(v \doteq x) \in pre(r)$. Note that o might itself be a causal rule which enables another one. Note further that since there may be several trigger conditions for r, occurence of o alone is not sufficient to actually trigger r.

Definition 4 *A plan $P_\mathcal{D} = (A, O, T, L, B)$ for a domain $\mathcal{D} = (\mathcal{A}, \mathcal{V}, \mathcal{E}, \mathcal{O})$ is **closed wrt. the domain rules** $\mathcal{O}_\mathsf{env}$ if the following holds:*

- *if $\mathsf{env} \in \mathcal{A}$ then $\mathsf{env} \in A$*
- *if $\mathsf{enables}(o, r)$ then $r' \in O$ and $(o, r', \mathbb{R}^+) \in T$ (where $o \in O, r \in \mathcal{O}_\mathsf{env}$ and r' is a unique copy of r)*[5].

In words, a closed plan contains instances of all rules that might possibly be triggered during its execution. Since rules may themselves trigger other rules, computing the closure

[4] This definition of *control* is sufficient for the situation assumed in this paper where there are basically only two *planning* (but many *executing*) agents: a centralized planner who can add and remove actions for all executing agents, and the environment agent env. Our model of *control* is, however, designed to be used also in a Distributed Planning paradigm where some planner $\mathit{Planner}_a$ is responsible but for one executing agent a. In that case, $\mathit{Planner}_a$ may not change the duration range of any process o controlled by agents $b \neq a$. In fact, $\mathit{Planner}_a$ can not even simply remove o from its current plan, since this would not force the planner controlling o, namely $\mathit{Planner}_b$, to do likewise.

[5] Such a copy of (or mapping to) a base action is usually called a *step*. Following most of the planning literature, we will ignore the distinction between steps and base events wherever possible.

for a given plan P amounts to a fixpoint computation, i.e. P is extended with the enabled rules repeatedly until a stable plan is reached[6].

A **MAP problem instance** is a tuple $\Pi = (\mathcal{D}, I, G)$ where $\mathcal{D} = (\mathcal{A}, \mathcal{V}, \mathcal{E}, \mathcal{O})$ is a MAP domain. I and G are knowledge consistent PVAs over $\mathcal{V}_\mathcal{A}$ called the **initial** and **goal knowledge distribution** and which, as usual, will be represented by the dummy actions $e_I = (\mathsf{env}, \emptyset, I)$ and $e_G = (\mathsf{env}, G, \emptyset)$. I might be incompletely specified (although the deman for knowledge consistency at least enforces that there are no false beliefs). Therefore, a solution plan for Π must be valid for all possible undefined values. To ensure this we define the set of possible additional initial events $\mathcal{E}_I^? := \{(\mathsf{env}, \emptyset, \{(v \doteq x)\}) \mid v \in \mathsf{undef}_I \wedge x \in dom_v\}$. All of these events will conditionally appear in the plan, labeled with their own effect, thereby defining the only branching context where they can occur.

Definition 5 *A plan $P_\mathcal{D} = (A, O, T, L, B)$ is a **solution** to $\Pi = (\mathcal{D}, I, G)$ if*

- $O = O' \cup \{e_I, e_G\} \cup \mathcal{E}_I^?$ *where* $O' \subseteq O_A$
- $e_I \prec e \prec e_G$ *and* $e^? \prec e$ *for all* $e \in O'$ *and all* $e^? \in \mathcal{E}_I^?$
- $B(e_I) = B(e_G) = \emptyset$ *and* $B(e) = \mathit{eff}(e)$ *for all* $e \in \mathcal{E}^?$
- T *is process-consistent with* $\mathcal{D}$ *and forms a consistent STN*
- P *is closed wrt. the domain rules* $\mathcal{O}_\mathsf{env}$
- P *contains no threatened causal links*
- *the only open conditions in* P *are in rules* $r \in \mathcal{O}_\mathsf{env}$ *not supporting causal links*

This definition is a straightforward extension of what constitutes solutions in POCL planning. One major difference is the role of individual beliefs in a plan, expressed by the knowledge preconditons and effects of events. A solution plan must, in particular, not contain open knowledge conditions. Thus, the definition forces planners to make sure that knowledge necessary for synchronized actions is shared among the executing agents. Either agents must be brought into positions to perceive changes themselves or communicative actions must be previewed in a plan.

Another novelty is the role of control (embedded in the notion of process-consistency) that different agents have over different actions in the plan, and especially the role that natural events play for modelling concurrency, perception, and, generally, complex ramifications of events caused by agents. Since natural events need not happen necessarily, the definition allows conditions of CDRs not used in causal links to be left open. In this respect, CDRs are similar to conditional effects in POCL planning whose conditions must only be supported if their effect is needed in a causal link.

[6]Since we do not prevent cyclic triggering of rules, the closure of a plan P might be infinite. For example, *day* may be a durative action with an end event triggering another process, *night*, which in turn triggers *day* again. This is a natural way to model recurring events. For space reason, we will not discuss it in detail here, but assume that either cyclic rules do not exist or that the planning and execution horizon is restricted to some *time window* within which the closure is finite.

As the final result of all formalizations, the only theorem in this paper confirms our definition of a "solution" to a MAP problem to be consistent with the state transition model of Classical Planning.

To show this relation, we firstly need complete states to compute transitions on. For a problem instance $\Pi = (\mathcal{D}, I, G)$, a completely defined PVA s is a **possible initial state** if $s(v) = I(v)$ whenever $I(v)$ is defined. $\mathcal{I}_\Pi$ is the set of **possible initial states** for Π.

Secondly, we must clarify the relation between the temporal constraint networks of MA plans and the transition sequences of Classical Planning. We first note that each possible initial state $s \in \mathcal{I}_\Pi$ is a branching context for the execution of a solution plan P, i.e. s induces an unconditional plan $P_s = (A, O', T', L', \emptyset)$ which only contains those processes that in the original solution P where labelled consistently with I'. Formally: $e \in \mathcal{E}^{O'}$ iff $B(e)$ is consistent with s. An execution schedule for an unconditional plan $P_s = (A, O', T', L', \emptyset)$ is a plan $sched(P_s) = (A, O', T'', L', \emptyset)$ where T'' is an extension of T' such that for each pair of events $e_1, e_2 \in \mathcal{E}^{O'}$ either $(e_1 \prec e_2) \in T''$, $(e_2 \prec e_2) \in T''$, or $(e1, e2, [0, 0]) \in T$. An execution schedule is valid, if, despite the new constraints, the underlying STN remains consistent and process-consistent. A valid execution schedule describes a possible sequence of events when executing P_s. This schedule, however, may still include simultaneous events. This is, however, unproblematic since the definition of threats (Def. 3) prevents simultaneous occurence of conflicting events. Therefore, to construct a transition sequence, it is possible to allow those events to virtually occur in arbitrary order: we define the set of totally ordered transition sequences of P_s to consist of all sequences $seq = \langle e_1, \ldots, e_n \rangle$ that are topological sortings of valid execution schedules $sched(P_s)$.

Given the set of possible initial states and the induced set of possible transition sequences for a plan P, we can finally relate the semantics of multiagent plans to classical plans by the following theorem:

Theorem 1 *Let $P_\mathcal{D} = (A, O, T, L, B)$ be a solution to a MAP problem $\Pi = (\mathcal{D}, I, G)$. Then, for all possible initial states $s \in \mathcal{I}_\Pi$, $G \subseteq res(s, p)$ for all $p \in TO(P_s)$.*

Proof sketch: Analogously to the semantics of classic POCL plans which is also defined in terms of topological sortings of a partially ordered sequence of events, we use valid execution schedules $sched(P_s)$ to define what Fox & Long (2003) call a "happening sequence" of a temporal plan. Def. 5 explicitly orders the possible initial events $\mathcal{E}_I^?$ before all other events in a solution plan. Therefore, since those events do not threaten any others in P_s (otherwise P_s would violate Def. 5 and thus would not be a solution), each $sched(P_s)$ must be executable in s. Since P_s must also contain neither open conditions nor threats, P_s is executable in s (Penberthy & Weld 1992). In particular, the goal event e_G that is scheduled after all other events will be the last event occuring in a topological sortings of $sched(P_s)$. Therefore G is true in the final state of the execution.

We have left out another, more interesting theoretical result, namely how individually executable plan fragments can

be generated from a global plan that are guaranteed to be jointly executable. While the result is rather obvious (since knowledge preconditions ensure that agents wait until they *perceive* the satisfied preconditions, even if they don't know about events that have caused the perception) its description in terms of mergeable individual plans is beyond the space of this paper.

7 Planning for Multiple Agents

To show that planning is indeed possible for MAP domains we will now sketch an algorithm for planning in our formalism. It is, however, not specifically tailored to MA Planning, but mainly consists in a transformation of the planning problem to a well-known representation (POCL plans with conditional effects) and the subsequent application of a standard POCL Planning algorithm. As such, the algorithm is certainly less efficient than the special-purpose MAP algorithm we will present in forthcoming work.

In a preprocessing step, all induced belief and mutual belief variables are generated, and all knowledge preconditions and effects a added explicitly to events. In particular, mutual belief effects are added to speech acts, and copresence rules are derived from sensor models. Then for each instantaneous CDR r that is enabled by an event e we extend e by a conditional effect corresponding to r. Afterwards, the original CDRs are removed from the MAP domain.

The actual planning algorithm works like UCPOP (Weld 1994): it resolves open conditions by supporting them with causal links, thereby adding processes to the plan if necessary, and threats by promotion, demotion, or confrontation. In particular, confrontation will ensure that no harmful CDRs are triggered. If the current partial plan enables non-instantaneous CDRs its closure must explicitly be computed for threat and validity checking. Generally, when a non-instantaneous process o is added, this means that all events $\mathcal{E}_o$ and constraints between them must be added to the plan. Similarly, when promoting or demoting processes, their duration constraints must be preserved by moving the start, end, and invariant event simultaneously.

Based on the soundness and completeness of UCPOP we can easily prove soundness and completeness of the modified algorithm for the case where duration constraints are merely ordering constraints. For metric duration constraints, we must further guarantee that the Simple Temporal Network underlying the plan is consistent. This can be achieved in low polynomial time (Dechter, Meiri, & Pearl 1991; Younes & Simmons 2003). The definition of threats from Def. 3 which uses temporal instead of relational ordering constraints, ensures that every plan found by the algorithm can indeed be executed in every possible total order without endangering causal links.

8 Related Work

This work integrates ideas from several research communities, in particular Classical and Distributed Planning, Multiagent Systems, Epistemic Logic, and Reasoning about Actions and Change.

Boutilier & Brafman (Boutilier & Brafman 2001) developed a formalism for multi-actuator plans and a planning algorithm based on classical POCL techniques. They model interacting effects of concurrent actions by specific kinds of conditional effects of the individual agents. A plan must provide simultaneity constraints ensuring that the interaction really takes place as planned. The authors assume that an external synchronization mechanism will ensure that during execution the constraints are met by the agents. Our formalism, however, rests on the assumption that executing agents are truly autonomous and there is no external instance to synchronize them. Therefore it must allow agents to synchronize on their own. This is achieved by explicit representation of changing knowledge and reasoning about individual and joint perceptions.

The events and temporal constraints in multiagent plans form a Simple Temporal Network (STN) (Dechter, Meiri, & Pearl 1991). Earlier work using this approach to extending PO plans with quantitative temporal constraints include (Ghallab & Laruelle 1994; Younes & Simmons 2003). These approaches subsume the temporal model of PDDL 2 (Fox & Long 2003), but extend it with flexible action durations that are necessary for our "qualitative" approach multiagent synchronization based on perception, rather than on absolute time points.

Conditional single-agent plans based on STNs were used by Tsamardinos et al. (Tsamardinos, Pollack, & Horty 2000; Tsamardinos, Vidal, & Pollack 2002). The notion of different kinds of *control* over intervals in a temporal constraint network was introduced by Vidal and Fargier (Vidal & Fargier 1999). In this paper we provide an extension to these approaches by specifying conditional temporal *multiagent* plans. However, we permit flexible action durations, but no other temporal constraints, which, for the time being, allows us to abstract from the subtler points of control and plans with observation nodes.

We do not know of other work on (multiagent) planning that formalizes the notion of causal domain laws or provides a similar approach to describing complex ramifications of concurrent multiagent actions. Our approach is inspired by work of Thielscher (Thielscher 1995) in the Theory of Actions community.

None of the above mentioned research describes execution time synchronization, sensor modeling, or communication. Our approach to planning in the presence of sensing is inspired by work of Etzioni, Weld & colleagues (Etzioni *et al.* 1992; Golden & Weld 1996; Smith & Weld 1999), Levesque (Levesque 1996), and Petrick & Bacchus (Petrick & Bacchus 2002; 2004). Again, we extend previous work to the multiagent case, thereby providing the basis for synchronized action at execution time. In particular, our explicit modeling of sensing and communication in multiagent environments complements BDI-inspired MAP models like (Grosz & Kraus 1996) that describe the role of (mutual) beliefs in necessary conditions for planful MA behavior, but do not explain how these conditions can be achieved during plan execution.

Since the focus of this paper is Distributed Plan Execution rather Distributed Planning, we will only briefly relate

our representation to some of the formal models used in that area. An excellent survey on techniques for Distributed Planning can be found in the paper by desJardins et. al. (DesJardins *et al.* 2000). Within this field there is a huge body of work relying on a hierarchical representation of multiagent plans (Durfee & Lesser 1987; Durfee & Montgomery 1991; DesJardins & Wolverton 1999; Clement & Durfee 1999b; 1999a). Hierarchical plans are very important in practical applications and therefore we are planning to extend our formalism to account for action decompositions. We believe that this extension should prove not to be complicated, since durative actions and their "invariant conditions", as used in our model, may be employed to "hide" an action decomposition and its "inconditions".

9 Conclusion and Future Work

We have presented a rich formal model of multiagent planning that includes and clarifies many important characteristics of MAP missing or underspecified in previous work. In particular, our model describes sensing and communication, and how both explain the evolution of (common) knowledge during a plan. Perceptions and knowledge provide the basis for "qualitative" synchronization of plans, i.e. quantitative notions like exact time points and durations become less important, thereby giving multiagent plans the flexibility needed by truly autonomous agents.

We have sketched a planning algorithm for MAP domains. A more elaborate algorithm will be presented soon. It is based on an extension of state-space forward-search techniques to POCL planning which we call Progressive Partial-Order Planning. This technique also allows to easily reason about triggered domain rules (or prevent their firing).

Although efficient algorithms and empirical results are yet missing in this paper, it is our hope that by defining not only the formal semantics, but also providing sample domains, a parser, and a plan validator for a concrete PDDL-like syntax, we can provide helpful tools for other MAP researchers and thus help to advance this interesting field of research.

References

Bäckström, C., and Nebel, B. 1995. Complexity results for SAS^+ planning. *Computational Intelligence* 11(4):625–655.

Blum, A., and Furst, M. L. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90(1-2).

Boutilier, C., and Brafman, R. 2001. Partial order planning with concurrent interacting actions. *JAIR*.

Brenner, M. 2003. Multiagent planning with partially ordered temporal plans. In *Proc. IJCAI '03*. Extended version: TR 190, Inst. for Computer Science, Freiburg Univ.

Clement, B., and Durfee, E. 1999a. Top-down search for coordinating the hierarchical plans of multiple agents. In *Agents-99*.

Clement, B. J., and Durfee, E. H. 1999b. Theory for coordinating concurrent hierarchical planning agents using summary information. In *Proc. of AAAI '99*, 495–502. Menlo Park, CA: AAAI Press.

Dechter, R.; Meiri, I.; and Pearl, J. 1991. Temporal constraint networks. *Artificial Intelligence* 49.

DesJardins, M., and Wolverton, M. 1999. Coordinating a distributed planning system. *AI Magazine* 20(4).

DesJardins, M.; Durfee, E.; C. Ortiz, J.; and Wolverton, M. 2000. A survey of research in distributed, continual planning. *AI Magazine*.

Durfee, E. H., and Lesser, V. R. 1987. Using partial global plans to coordinate distributed problem solvers. In *Proc. IJCAI-87*.

Durfee, E., and Montgomery, T. 1991. Coordination as distributed search in hierarchical behavior space. *IEEE Transactions on Systems, Man, and Cybernetics*.

Etzioni, O.; Hanks, S.; Weld, D.; Draper, D.; Lesh, N.; and Williamson, M. 1992. An approach to planning with incomplete information. In *Proc. of KR '92*, 115–125.

Fagin, R.; Halpern, J. Y.; Moses, Y.; and Vardi, M. Y. 1995. *Reasoning About Knowledge*. MIT Press.

Fox, M., and Long, D. 2003. PDDL 2.1: an extension to PDDL for expressing temporal planning domains. *JAIR* 20:61–124.

Ghallab, M., and Laruelle, H. 1994. Representation and control in IxTeT, a temporal planner. In *Proc. of AIPS '94*.

Golden, K., and Weld, D. S. 1996. Representing sensing actions: The middle ground revisited. In *Proc. of KR '96*, 174–185.

Golden, K. 1998. Leap before you look: Information gathering in the puccini planner. In *AIPS*, 70–77.

Grosz, B. J., and Kraus, S. 1996. Collaborative plans for complex group action. *Artificial Intelligence* 86(2):269–357.

Helmert, M. 2004. A planning heuristic based on causal graph analysis. In *Proc. ICAPS 2004*, 161–170.

Levesque, H. J. 1996. What is planning in the presence of sensing? In *Proc. AAAI*, 1139–1146.

Lewis, D. 1969. *Convention. A Philosophical Study*. Cambridge, Massachusetts: Harvard University Press.

Penberthy, J. S., and Weld, D. 1992. UCPOP: a sound, complete partial order planner for adl. In *Proc. KR'92*.

Peot, M. A., and Smith, D. E. 1992. Conditional nonlinear planning. In *Proc. AIPS-92*.

Petrick, R., and Bacchus, F. 2002. A knowledge-based approach to planning with incomplete information and sensing. In *Proc. ICAPS-02*.

Petrick, R. P. A., and Bacchus, F. 2004. Extending the knowledge-based approach to planning with incomplete information and sensing. In *Proc. of ICAPS '04*, 2–11.

Smith, D., and Weld, D. 1999. Temporal planning with mutual exclusion reasoning. In *Proc. IJCAI*.

Thielscher, M. 1995. The logic of dynamic systems. In *Proc. IJCAI-95*.

Tsamardinos, I.; Pollack, M. E.; and Horty, J. F. 2000. Merging plans with quantitative temporal constraints, temporally extended actions, and conditional branches. In *Proc. AIPS-00*.

Tsamardinos, I.; Vidal, T.; and Pollack, M. 2002. CTP: A new constraint-based formalism for conditional, temporal planning. *Constraints Journal*.

Vidal, T., and Fargier, H. 1999. Handling contingency in temporal constraint networks. *Journal of Experimental and Theoretical AI*.

Weld, D. S. 1994. An introduction to least commitment planning. *AI Magazine* 15(4):27–61.

Younes, H., and Simmons, R. 2003. VHPOP: Versatile heuristic partial order planner. *JAIR* 20:405–430.

From Multiagent Plan to Individual Agent Plans

Olivier Bonnet-Torrès
Supaero & Onera-cert DCSD
2, av. E.Belin
31055 Toulouse cedex 4 FRANCE
olivier.bonnet@onera.fr

Catherine Tessier
Onera-cert DCSD
catherine.tessier@onera.fr

Abstract

This paper focuses on a framework for representing a team plan and its projections on individual agents. The team plan is represented with a coloured Petri net. Some structures in the net provide basic team management possibilities and illustrate notions such as (sub)team splitting or merging and agent transfer. They in fact describe the dynamic team organisation. Through net reduction they provide means to build an agenticity hierarchy, *i.e.* a hierarchical organisation of the team in accordance with the goals to be achieved. At each level of agenticity a local plan is derived from the team plan reduction.

Mission, Agents and Team Organisation

The general framework is a mission specified in terms of objectives: agents are operated in order to carry out the mission and they are hierarchically organised in a team.

This paper aims at formalising the relationship between the team plan and individual agents' plans through the use of Petri nets. The plan model facilitates plan information management during mission execution.

Related Work

Hierarchical task networks (HTN) (Erol, Hendler, & Nau 1994) consist in decomposing tasks into subtasks until elementary tasks. A set of methods to achieve each task is then organised into an agent plan. In the wake of HTN, Grosz *et al.* (Grosz & Kraus 1996) base the *SharedPlan* approach on the hierarchical decomposition of shared plans into a sequence of recipes to be applied by a team of agents. Their work also inherits the logics of beliefs and intentions (Cohen & Levesque 1990; 1991; Rao & Georgeff 1995). Tambe *et al.* (Tambe 1996; 1997) have focused on team behaviour in STEAM. The planning module in STEAM uses rules to produce team reactions to external events.

From a different standpoint the representation of the plan itself tends to make use of the automata theory and the Petri net formalism (Murata 1989). For instance El Fallah *et al.* have modified Petri nets (El Fallah-Seghrouchni & Haddad 1996) in order to take into account the refining of actions, to be compared to task decomposition. The multi-agent hue is treated in merging individual plans. Another approach (El Fallah-Seghrouchni, Degirmencyan-Cartault, & Marc 2004) has led to using hybrid automata to formalise and execute agent plans. The automata are generalised into synchronised automata in order to represent the team plans. However, in the domain of individual planning, operational use of Petri nets is appearing for representing an itinerary and controlling the execution of the subsequent plan (Chanthery, Barbier, & Farges 2004) or even as a task planning and scheduling tool compatible with Petri net design and analysis environments (Kristensen 2003).

The rest of the section presents the Petri net formalism and introduces the notion of agenticity to denote the organisation of the team. The next section formalises team plan Petri nets and their relations to team organisation. The final section exposes a way to extract individual plans from the team plan using reduction rules and a projection operator.

A Petri Net Reminder

A *Petri net* $< P, T, F, B >$ is a bipartite graph with two types of nodes: $P = \{p_1, ..., p_i, ..., p_m\}$ is a finite set of places; $T = \{t_1, ..., t_j, ..., t_n\}$ is a finite set of transitions (Murata 1989). Arcs are directed and represent the forward incidence function $F : P \times T \rightarrow \mathbb{N}$ and the backward incidence function $B : P \times T \rightarrow \mathbb{N}$ respectively. An *interpreted Petri net* is such that conditions and events are associated with places and transitions respectively. When the conditions corresponding to some places are satisfied, tokens are assigned to those places and the net is said to be marked. The evolution of tokens within the net follows transition firing rules. Petri nets allow sequencing, parallelism and synchronization to be easily represented.

Mission and Goals

The mission is characterised by an *objective* to be reached by the agents team. The objective is decomposed into mission *goals*, which are in turn decomposed into subgoals until reaching elementary goals.

Definition 1 *(Agent) an* agent *is a physical entity equipped with resources (sensors, actuators, communication devices) that is implemented to achieve some goals within the mission, therefore contributing to the achievement of the objective. An* elementary agent *is an indivisible entity (e.g. a robot, a drone) whereas a* composite agent *is a set of agents that may themselves be organised as composite agents.*

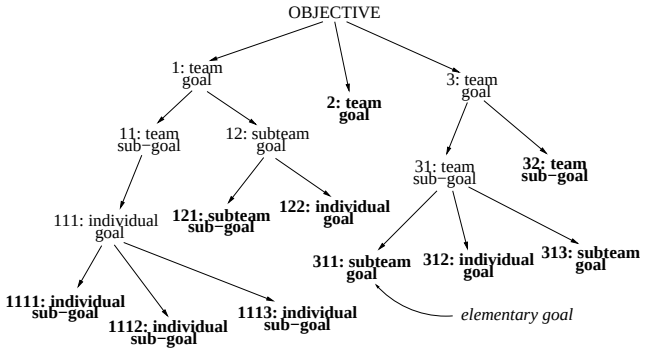


Figure 1: Decomposing the objective into a hierarchy of goals

Following Shoham's remark that a group of closely interacting agents can be considered as an agent in itself (Shoham 1993) a team of agents is equivalent to a composite agent.

Definition 2 *(Goal) for an agent a goal corresponds to a possible state of the environment such that the actions of the agent tend to bring the environment to that state.*

The decomposition of the objective gives a hierarchy of goals that must be carried out (Tambe 1996) (see fig. 1). Some goals involve elementary agents, other involve composite agents, *i.e.* subteams or even the team itself.

Definition 3 *(Recipe) a recipe is the specification of a course of actions to be performed by an agent, either composite or individual, resulting in the achievement of a goal.*

Definition 4 *(Elementary goal) an elementary goal is such that there exists a known recipe to achieve it (see fig. 1).*

Several recipes may be available to achieve one elementary goal. The team plan is extracted by organising a subset of the set of recipes. The initial plan is attached a possible organisation of the team.

Agenticity

When an agent is involved in a group of agents, some characteristics of the group are inherited by the agent. In particular if the group is involved in some activity, each individual agent is committed to that activity and to the interaction with its fellow agents (Cohen & Levesque 1991). To make use of this property we suggest to consider a team as an *agenticity hierarchy*, whose leaves are elementary agents and whose nodes are subteams, *i.e.* composite agents. Each node has for children nodes the agents that compose the subteam it represents (see fig. 2). One can notice that there is no requirement that an individual agent be represented only once.

More formally the team X is composed of elementary agents $\{x_1, x_2, \ldots, x_n\}$. It is hierarchically organised and each node in hierarchy $\mathcal{H}_X$ is considered as an agent a_i (Shoham 1993). Let $A = \{a_1, a_2, \ldots, a_m\}$ be the set of agents in team X. Preliminary properties are that:

1. the team is an agent: $X \in A \Leftrightarrow \exists p \in [1, \ldots, m], X = a_p$,
2. and each individual has a counterpart in the complex agent set: $x_i \in X \Rightarrow \exists j \in [1, \ldots, m], x_i = a_j$.

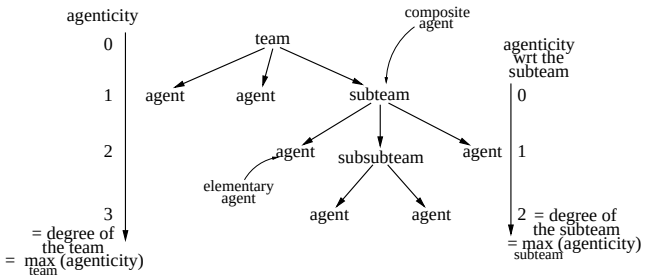


Figure 2: Agenticity Hierarchy

The father of agent a_i is denoted $father(a_i)$. $child(a_i)$ is the set of children of a_i. $child(.)$ and $father(.)$ are functions and as such can be composed. The hierarchy $\mathcal{H}_X$ is an application: $\begin{cases} A \setminus \{x_1, x_2, \ldots, x_n\} \to A \\ a_i \mapsto child(a_i) \end{cases}$.

Definition 5 *(Agenticity) the agenticity of agent a_i with regards to team X is its depth in hierarchy $\mathcal{H}_X$ whose root is the team: $Ag_X(a_i) = depth(a_i, \mathcal{H}_X) = (u | father^u(a_i) = X)$.*
The agenticity of agent a_i with regards to any subteam $a_j, a_i \subset a_j$ is its depth in hierarchy $\mathcal{H}_{a_j}$ whose root is the considered subteam: $Ag_{a_j}(a_i) = depth(a_i, \mathcal{H}_{a_j}) = u$ such that $father^u(a_i) = a_j$.

Definition 6 *The father agent of agent a_j is agent $a_k = father(a_j)$ corresponding to the father node in hierarchy $\mathcal{H}_X$. The father's agenticity is less than the child's by 1: $a_j \subset child(a_k) \Rightarrow Ag_X(a_j) = Ag_X(a_k) + 1$.*

Examples

1. The agenticity of an agent pertaining to no subteam is 1 with regards to the team: $X = a_p, x_i = a_j, \nexists k \in [1, \ldots, m] \setminus \{j, p\} : a_j \subset a_k \Rightarrow Ag_X(x_i) = 1$ (see fig. 3).

2. If all agents belong to the same team, the agenticity of the team is 0 with regards to the agent population: $\forall i \in \{1, \ldots, n\} : a_i \in X \Rightarrow Ag_A(X) = 0, A = \{x_i, i \in \{1, \ldots, n\}\} \cup \{X\}$.

Definition 7 *(Degree) the degree of an agent is the highest agenticity of the individual agents that belong to this agent: $deg(a_j) = \max(Ag_{a_j}(x_i, i \in \{1, \ldots, n\}, x_i \in a_j)$. An elementary agent has a null degree: $deg(x_i) = 0$.*

Example

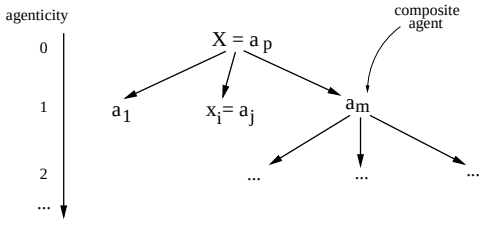


Figure 3: Example 1

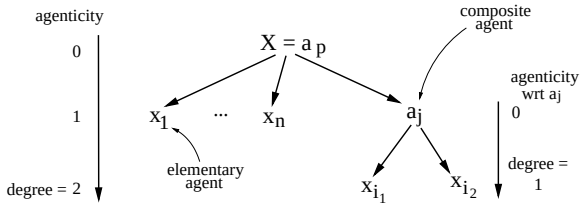


Figure 4: Example 3

3. If two elementary agents compose the only subteam of a given team, the team has a degree of 2: $a_j = \{x_{i_1}, x_{i_2}\}, A = \{X\} \cup \{a_j\} \cup \{x_i, i \in \{1, \ldots, n\} \setminus \{i_1, i_2\}\} \Rightarrow deg(X) = 2$ (see fig. 4). The father a_j of the two agents x_{i_1} and x_{i_2} is the composite agent representing the subteam.

Team Plan Representation

Team Plan Definition

The team plan is designed in terms of a detailed sequence of tasks achieving the identified elementary goals, represented as a Petri net (Bonnet-Torrès & Tessier 2005).

Let $\mathcal{P}_X$ be the detailed team plan. $\mathcal{P}_X$ is a coloured Petri net (Jensen 1997): $\mathcal{P}_X = (P, T, S, N, \mathcal{C}, F)$, such that (see fig. 5):

1. P is a finite set of places p_i, each place p_i represents the activity associated to an elementary goal;
2. T is a finite set of transitions t_j;
3. S is a finite set of arcs s_k;
4. N is a node function from S to $P \times T \cup T \times P$;
5. $\mathcal{C}$ is the colour set;
6. F is a colour function from P into $\mathcal{C}$.

$F : \left\{ \begin{array}{l} P \rightarrow \mathcal{C} \\ p_i \mapsto \mathcal{H}_X(p_i) \end{array} \right.$. The set of token colours $\mathcal{C}$ is the set of agenticity hierarchies. The colour of a given token in a given place p_i, $c(p_i)$, is the branch corresponding to the activity associated to the place in the agenticity hierarchy: $c(p_i) = \mathcal{H}_X(p_i) = \{father^k(X|_{p_i}), k \in \{0, \ldots, deg(X|_{p_i})\}\}$, where the elementary agents involved in p_i are $X|_{p_i}$. Hence each reachable marking $\mathcal{M}$ corresponds to an agenticity hierarchy $\mathcal{H}_X(\mathcal{M})$ of the whole team X.

Analysing the Team Plan

Petri net analysis can be performed through the use of the incidence matrix $\mathcal{A}$. $\mathcal{A}$ represents the relations between places and transitions, namely the arcs (Murata 1989): $\mathcal{A}_{i,k} = 1$ if p_i is an output place of t_k and $\mathcal{A}_{i,k} = -1$ if p_i is an input place of t_k. In the analysis of the team plan through the incidence matrix, colours will be abstracted.

The team plan bears some typical structures that can be identified as modifications of the team organisation. Let us recall the notations ${}^\circ t_j$ for the input places of t_j, $t_j{}^\circ$ for its output places, ${}^\circ p_i$ for the input transitions of place p_i and $p_i{}^\circ$ for its output transitions. The $^\circ$ is readily composable: for instance ${}^{\circ\circ}p_i$ designates the set of input places of all input transitions of p_i.

Definition 8 *(Source) Let* `source` *be the structure represented by a place p_i and a transition t_k such that ${}^\circ p_i = t_k$ and ${}^\circ t_k = \emptyset$ (see fig. 6).*
The hierarchy born by the structure has an agenticity of 1 with respect to the team: $\mathcal{H}_X(p_i) = a_s$ and $child(a_s) = \{a_{s_1}, \ldots, a_{s_q}\}$.

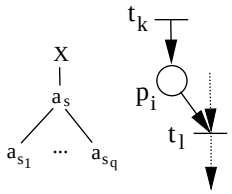


Figure 6: Source structure and its associated agenticity hierarchy

The `source` structure allows the introduction of q agents into the team. It is worth noticing that t_k cannot bear two or more output places because this would mean that a group of agents is introduced in the team and immediately split. Common sense does not allow this, all the more since transitions in Petri nets are considered indivisible and instantaneous. The signature in the incidence matrix $\mathcal{A}$ is the following:

$$\mathcal{A} = \begin{array}{c} \\ (p_i) \end{array} \begin{array}{c} \begin{array}{ccccccc} & & (t_k) & & (t_l) & & \end{array} \\ \left(\begin{array}{ccccccc} & & 0 & & & & \\ & & \vdots & & \vdots & & \\ & & 0 & & & & \\ \cdots & 0 & 1 & & -1 & 0 & \cdots \\ & & 0 & & & & \\ & & \vdots & & \vdots & & \\ & & 0 & & & & \end{array} \right) \end{array}$$

$$\sum_{u=1}^{|P|} \mathcal{A}_{u,k} = 1$$

Definition 9 *(Sink) Let* `sink` *be the structure represented by a place p_i and a transition t_k such that $p_i{}^\circ = t_k$ and $t_k{}^\circ = \emptyset$ (see fig. 7).*
The hierarchy born by the structure has an agenticity of 1 with respect to the team: $\mathcal{H}_X(p_i) = a_s$ and $child(a_s) = \{a_{s_1}, \ldots, a_{s_q}\}$.

The `sink` structure allows the withdrawal or the abduction of q agents from the team. It is worth noticing that t_k cannot bear two or more input places because this would mean that several groups of agents withdraw from the team at the same time while not being synchronised since they do not pertain to the same subteams. Common sense does not allow this, all the more since transitions in Petri nets are considered indivisible and instantaneous. The signature in the incidence matrix $\mathcal{A}$ is the following:

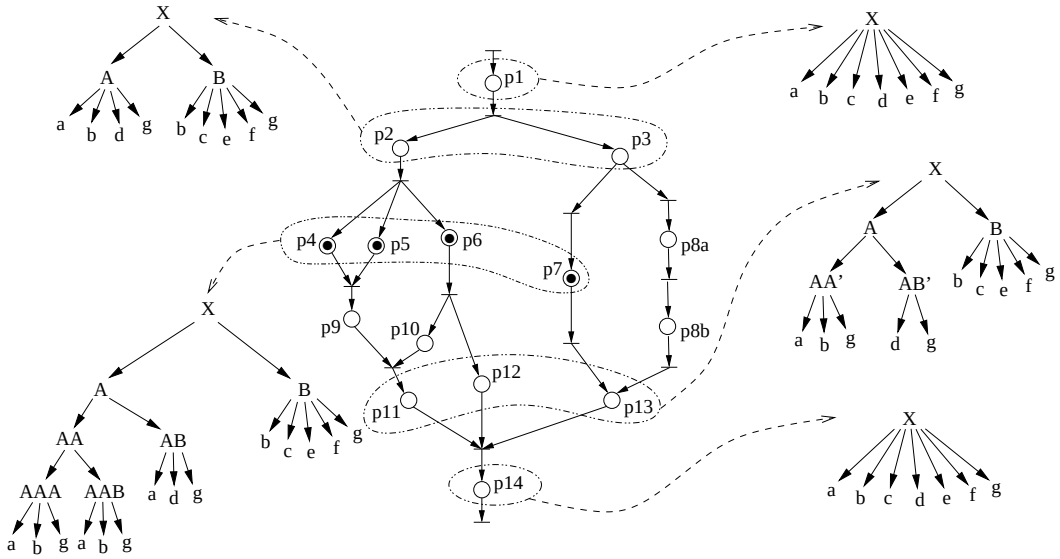


Figure 5: Team plan with some agenticity hierarchies

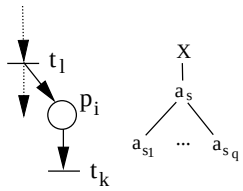


Figure 7: Sink structure and its associated agenticity hierarchy

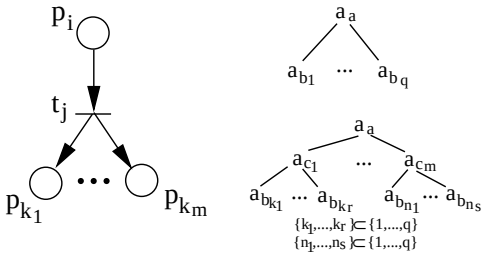


Figure 8: Fork structure and its associated agenticity hierarchies

$$\mathcal{A} = \begin{array}{c} \\ (p_i) \\ \\ \\ \\ \\ \\ \end{array} \begin{array}{c} \begin{array}{ccccccc} & & (t_k) & (t_l) & & & \end{array} \\ \begin{pmatrix} & & 0 & & & & \\ & & \vdots & \vdots & & & \\ & & 0 & & & & \\ \cdots & 0 & -1 & 1 & 0 & \cdots & \\ & & 0 & & & & \\ & & \vdots & \vdots & & & \\ & & 0 & & & & \end{pmatrix} \\ \sum_{u=1}^{|P|} \mathcal{A}_{u,k} = -1 \end{array}$$

Definition 10 *(Fork) Let* `fork` *be the structure based on transition t_j such that $^\circ t_j = p_i$ and $t_j^\circ = \{p_{k_1}, p_{k_2}, \ldots, p_{k_m}\}$ (see fig. 8).*
Firing transition t_j inserts before the individual level — for which $Ag = deg(\mathcal{H}_X(p_i))$ — a level of agenticity whose (composite) agents share out the individual agents among themselves: $deg(\mathcal{H}_X(t_j{}^\circ)) = deg(\mathcal{H}_X(^\circ t_j)) + 1$. If in p_i, $child(a_a) = \{a_{b_1}, \ldots, a_{b_q}\}$, in $p_{k_p}, p \in \{1, \ldots, m\}$, $child(a_a) = \{a_{c_1}, \ldots, a_{c_m}\}$ and $\cup_{s=1}^{m} child(a_{c_s}) = \{a_{b_1}, \ldots, a_{b_q}\}$.

The `fork` structure allows creating from a subteam m subteams whose levels of agenticity are increased by 1. The signature in the incidence matrix $\mathcal{A}$ is the following:

$$\mathcal{A} = \begin{array}{c} \\ (p_i) \\ \\ \\ \\ \\ (p_{k_1}) \\ \vdots \\ (p_{k_m}) \end{array} \begin{array}{c} \begin{array}{c} (t_j) \end{array} \\ \begin{pmatrix} \vdots \\ 0 \\ \cdots \quad -1 \quad \cdots \\ 0 \\ \vdots \\ 0 \\ \cdots \quad 1 \quad \cdots \\ \cdots \quad \vdots \quad \cdots \\ \cdots \quad 1 \quad \cdots \\ 0 \\ \vdots \end{pmatrix} \end{array}, \quad \begin{array}{l} \sum_{u=1}^{|P|} \mathcal{A}_{u,j} = m - 1 > 0 \\ \sum_{u=1}^{|P|} |\mathcal{A}_{u,j}| = m + 1 > 2 \end{array}$$

Definition 11 *(Merge) Let* `merge` *be the structure based on transition t_j such that $^\circ t_j = \{p_{i_1}, p_{i_2}, \ldots, p_{i_m}\}$ and $t_j^\circ = p_k$ (see fig. 9).*
Firing transition t_j suppresses the level of agenticity before the individual level. It thus fuses the composite agents of the truncated level: $deg(\mathcal{H}_X(t_j{}^\circ)) = deg(\mathcal{H}_X(^\circ t_j)) - 1$. If in $p_{i_p}, p \in \{1, \ldots, m\}$, $child(a_a) = \{a_{c_1}, \ldots, a_{c_m}\}$

and $\cup_{s=1}^{m} child(a_{c_s}) = \{a_{b_1}, \ldots, a_{b_q}\}$*, in* p_k*,* $child(a_a) = \{a_{b_1}, \ldots, a_{b_q}\}$.

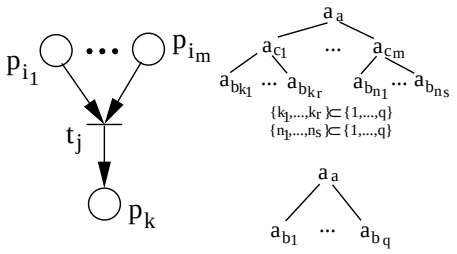


Figure 9: Merge structure and its associated agenticity hierarchies

The `merge` structure allows fusing m subteams to form a single subteam whose level of agenticity is decreased by 1. The signature in the incidence matrix $\mathcal{A}$ is the following:

$$\mathcal{A} = \begin{array}{c} \\ (p_k) \\ \\ \\ (p_{i_1}) \\ \vdots \\ (p_{i_m}) \end{array} \begin{array}{c} (t_j) \\ \left(\begin{array}{c} \vdots \\ 0 \\ \cdots \quad 1 \quad \cdots \\ 0 \\ \vdots \\ 0 \\ \cdots \quad -1 \quad \cdots \\ \cdots \quad \vdots \quad \cdots \\ \cdots \quad -1 \quad \cdots \\ 0 \\ \vdots \end{array} \right) \end{array}, \quad \begin{array}{l} \sum_{u=1}^{|P|} \mathcal{A}_{u,j} = 1 - m < 0 \\ \\ \sum_{u=1}^{|P|} |\mathcal{A}_{u,j}| = m + 1 > 2 \end{array}$$

Definition 12 *(Reorganise) Let* `reorganise` *be the structure based on transition* t_j *such that* $^\circ t_j = \{p_{i_1}, p_{i_2}, \ldots, p_{i_m}\}$ *and* $t_j{}^\circ = \{p_{k_1}, p_{k_2}, \ldots, p_{k_p}\}$ *(see fig. 10).*
Combining characteristics of the two preceding structures, firing transition t_j *modifies the composition and possibly the number of agents at level* $deg(\mathcal{H}_X(^\circ t_j)) - 1$*. However it does not affect the degree of the subteam:* $deg(\mathcal{H}_X(^\circ t_j)) = deg(\mathcal{H}_X(t_j{}^\circ))$*. If in* $p_{i_s}, s \in \{1, \ldots, m\}$*,* $child(a_a) = \{a_{c_1}, \ldots, a_{c_m}\}$ *and* $\cup_{u=1}^{m} child(a_{c_u}) = \{a_{b_1}, \ldots, a_{b_q}\}$*, in* $p_{k_s}, s \in \{1, \ldots, p\}$*,* $child(a_a) = \{a_{d_1}, \ldots, a_{d_p}\}$ *and* $\cup_{u=1}^{p} child(a_{d_u}) = \{a_{b_1}, \ldots, a_{b_q}\}$.

The `reorganise` structure allows fusing m subteams to form p new subteams, all of them bearing the same level of agenticity. The signature in the incidence matrix $\mathcal{A}$ is the following:

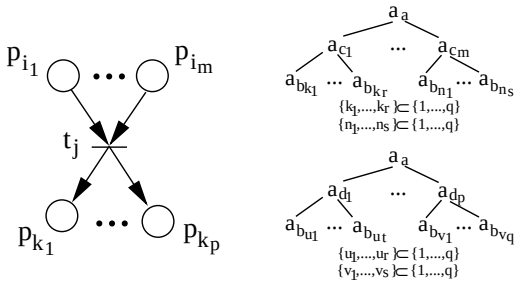


Figure 10: Reorganise structure and its associated agenticity hierarchies

$$\mathcal{A} = \begin{array}{c} \\ (p_{i_1}) \\ \vdots \\ (p_{i_m}) \\ \\ \\ (p_{k_1}) \\ \vdots \\ (p_{k_p}) \end{array} \begin{array}{c} (t_j) \\ \left(\begin{array}{c} \vdots \\ 0 \\ \cdots \quad -1 \quad \cdots \\ \cdots \quad \vdots \quad \cdots \\ \cdots \quad -1 \quad \cdots \\ 0 \\ \vdots \\ 0 \\ \cdots \quad 1 \quad \cdots \\ \cdots \quad \vdots \quad \cdots \\ \cdots \quad 1 \quad \cdots \\ 0 \\ \vdots \end{array} \right) \end{array}, \quad \begin{array}{l} \sum_{u=1}^{|P|} \mathcal{A}_{u,j} = p - m \\ \\ \sum_{u=1}^{|P|} |\mathcal{A}_{u,j}| = m + p > 3 \end{array}$$

Definition 13 *(Transfer) Let* `transfer` *be the structure based on a place* p_t *such that* $t_{j_1} = p_{i_1}{}^\circ = {}^\circ p_{k_1} = {}^\circ p_t$ *and* $t_{j_2} = {}^\circ p_{k_2} = p_{i_2}{}^\circ = p_t{}^\circ$ *(see fig. 11).*
It modifies the composition but does not change the number of agents at level $deg(\mathcal{H}_X(^\circ t_{j_{1,2}})) - 1$*: there always remains two of them. The places in the structure correspond to the following agents:*

- $p_{i_1} \rightarrow a_r = \{a_{r_u}, u \in \{1, \ldots, m\}\}$*;*
- $p_{i_2} \rightarrow a_s = \{a_{s_u}, u \in \{1, \ldots, p\}\}$*;*
- $p_t \rightarrow a_t = \{a_{t_u}, u \in \{1, \ldots, q\}\}$*;*
- $p_{k_1} \rightarrow a'_r = \{a_{r_u}, u \in \{1, \ldots, m\}\} \setminus \{a_{t_u}, u \in \{1, \ldots, q\}\}$;
- $p_{k_2} \rightarrow a'_s = \{a_{s_u}, u \in \{1, \ldots, p\}\} \cup \{a_{t_u}, u \in \{1, \ldots, q\}\}$.

The `transfer` structure allows transferring q agents from the activity associated with p_{i_1} to the activity associated with p_{k_2}. This is equivalent to collocating a `source` structure and a `sink` structure where p_t represents the withdrawing agents on one side and the arriving agents on the other. The signature in the incidence matrix $\mathcal{A}$ is the following:

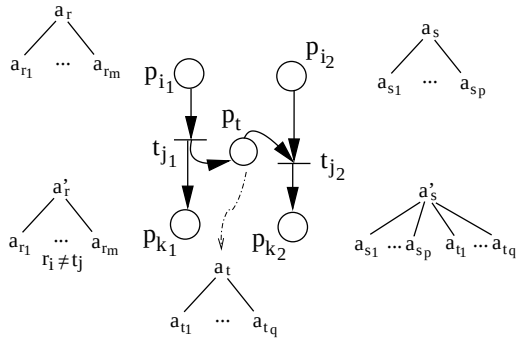


Figure 11: Transfer structure and its associated agenticity hierarchies

$$\mathcal{A} = \begin{array}{c} \\ (p_{i_1}) \\ (p_{i_2}) \\ (p_t) \\ (p_{k_1}) \\ (p_{k_2}) \end{array} \begin{pmatrix} (t_{j_1}) & (t_{j_2}) \\ 0 & 0 \\ \vdots & \vdots \\ 0 & 0 \\ \cdots \ \cdots & -1 \ 0 \ \cdots \ \cdots \\ \cdots \ \cdots & 0 \ -1 \ \cdots \ \cdots \\ \cdots \ 0 & 1 \ -1 \ 0 \ \cdots \\ \cdots \ \cdots & 1 \ 0 \ \cdots \ \cdots \\ \cdots \ \cdots & 0 \ 1 \ \cdots \ \cdots \\ 0 & 0 \\ \vdots & \vdots \\ 0 & 0 \end{pmatrix},$$

$$\sum_{u=1}^{|P|} \mathcal{A}_{u,j_1} = 1, \quad \sum_{u=1}^{|P|} \mathcal{A}_{u,j_2} = -1, \quad \sum_{u=1}^{|P|} |\mathcal{A}_{u,j_1}| = 3,$$
$$\sum_{u=1}^{|P|} |\mathcal{A}_{u,j_2}| = 3, \quad \sum_{v=1}^{|T|} \mathcal{A}_{t,v} = 0, \quad \sum_{v=1}^{|T|} |\mathcal{A}_{t,v}| = 2$$

Definition 14 *(Choice) Let* `choice` *be the structure located between two places p_i and p_j such that $p_i{}^\circ = \{t_{k_1}, t_{k_2}, \ldots, t_{k_m}\}$ and $\forall u \in \{1, \ldots, m\}, t_{k_u}{}^\circ = p_{l_u}$, $p_{l_u}{}^\circ = t_{n_u}$ and $t_{n_u}{}^\circ = p_j$ (see fig. 12).*
The hierarchy is not modified by the structure: $\mathcal{H}_X(p_i) = \mathcal{H}_X(p_j) = \mathcal{H}_X(p_{l_u}), \forall u \in \{1, \ldots, m\}$.

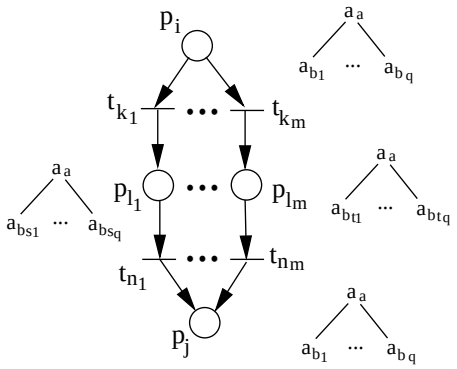


Figure 12: Choice structure and its associated agenticity hierarchies

The `choice` structure allows exposing m possible activities for the considered subteam. The signature in the incidence matrix $\mathcal{A}$ is the following:

$$\mathcal{A} = \begin{array}{c} \\ (p_i) \\ (p_{l_1}) \\ \vdots \\ (p_{l_m}) \\ (p_j) \end{array} \begin{pmatrix} (t_{k_1}) & \cdots & (t_{k_m}) & & (t_{n_1}) & \cdots & (t_{n_m}) \\ \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\ 0 & \cdots & 0 & & 0 & \cdots & 0 \\ \cdots \ -1 & \cdots & -1 & 0 & \cdots & \cdots & \\ 0 \ 1 & 0 & \cdots & 0 & -1 & 0 & \cdots \ \cdots \\ \cdots & \ddots & \ddots & \ddots & \cdots & \ddots & \ddots \ \ddots \ \cdots \\ \cdots & \cdots \ 0 & 1 & 0 & \cdots & 0 & -1 \ 0 \\ \cdots \ 0 & \cdots & \cdots & 0 & 1 & \cdots & 1 \ \cdots \\ 0 & \cdots & 0 & & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \end{pmatrix},$$

$$\sum_{v=1}^{|T|} \mathcal{A}_{i,v} = |{}^\circ p_i| - m \quad and \quad \sum_{v=1}^{|T|} |\mathcal{A}_{i,v}| = |{}^\circ p_i| + m$$

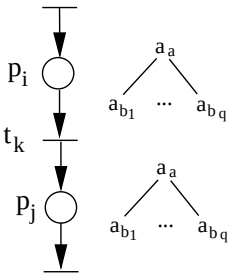
$$\forall s \in \{1, \ldots, m\}, \sum_{u=1}^{|P|} \mathcal{A}_{u,k_s} = 0 \quad and \quad \sum_{u=1}^{|P|} |\mathcal{A}_{u,k_s}| = 2$$

$$\forall s \in \{1, \ldots, m\}, \sum_{v=1}^{|T|} \mathcal{A}_{l_s,v} = 0 \quad and \quad \sum_{v=1}^{|T|} |\mathcal{A}_{l_s,v}| = 2$$

$$\forall s \in \{1, \ldots, m\}, \sum_{u=1}^{|P|} \mathcal{A}_{u,n_s} = 0 \quad and \quad \sum_{u=1}^{|P|} |\mathcal{A}_{u,n_s}| = 2$$

$$\sum_{v=1}^{|T|} \mathcal{A}_{j,v} = m - |p_j{}^\circ| \quad and \quad \sum_{v=1}^{|T|} |\mathcal{A}_{j,v}| = m + |p_j{}^\circ|$$

Remark: one can notice that the simple structure with a transition t_k and two places p_i and p_j such that $p_i{}^\circ = t_k$, ${}^\circ p_j = t_k$, $t_k{}^\circ = p_j$ and ${}^\circ t_k = p_i$ (see fig. 13) does not modify the agents involved. The signature in the incidence matrix $\mathcal{A}$ is the following:



$$\mathcal{A} = \begin{array}{c} \\ (p_i) \\ (p_j) \end{array} \begin{pmatrix} (t_k) \\ 0 \\ \vdots \\ 0 \\ \cdots \ -1 \ \cdots \\ \cdots \ 1 \ \cdots \\ 0 \\ \vdots \\ 0 \end{pmatrix},$$

Figure 13: Sequence structure and its associated agenticity hierarchy

$$\sum_{u=1}^{|P|} \mathcal{A}_{u,k} = 0,$$

$$\sum_{v=1}^{|T|} \mathcal{A}_{i,v} = |{}^\circ p_i| - 1 \geq 0, \quad \sum_{v=1}^{|T|} \mathcal{A}_{j,v} = -|p_j{}^\circ| + 1 \leq 0$$

Abstracting the Team Plan

Representing a team plan using hierarchical coloured Petri nets (HCPN) (Huber, Jensen, & Shapiro 1989; Lakos 1995) — or modular coloured Petri nets (MCPN) (Christensen &

Petrucci 1992; Lakos 1995) — allows for more flexibility than coloured Petri nets (CPN) and reduces the amount of duplicated information.

The net $\mathcal{P}_X$ can be abstracted so as to represent the activities at each level of agenticity. To build this information we extend the ordinary Petri net reduction rules. The team plan Petri net structure is reduced according to the semantics of basic team management structures, namely `source`, `sink`, `fork`, `merge`, `reorganise`, `transfer` and `choice`. Each reduction step builds a sub-hierarchy of agenticity.

Rule 1 — Reduction of late arrival: *(fig. 14) If t_k and p_i constitute a source structure,* i.e. $^\circ p_i = t_k$, $^\circ t_k = \emptyset$, $p_i{}^\circ = t_l$ *and* $\exists j \neq i, p_j{}^\circ = t_l$*, they are absorbed by a single place* p_{j,i^*}.

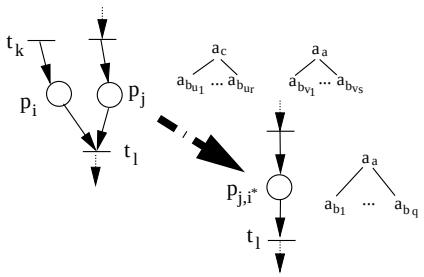


Figure 14: Rule 1 and its effect on hierarchy

Rule 1 preserves the level of agenticity. However the token is modified so as to encompass the newly introduced (individual or composite) agent. The effect on $\mathcal{A}$ is to suppress the line and column corresponding to p_i and t_k, respectively.

Rule 2 — Reduction of early withdrawal: *(fig. 15) If p_i and t_k constitute a sink structure,* i.e. $p_i{}^\circ = t_k$, $t_k{}^\circ = \emptyset$, $^\circ p_i = t_l$ *and* $\exists j \neq i, ^\circ p_j = t_l$*, they are absorbed by a single place* $p_{j^*,i}$.

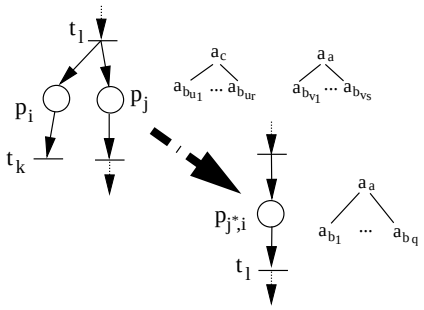


Figure 15: Rule 2 and its effect on hierarchy

Rule 2 preserves the level of agenticity. However the token is modified so as to encompass the leaving (individual or composite) agent. The effect on $\mathcal{A}$ is to suppress the line and column corresponding to p_i and t_k, respectively.

Rule 3 — Fusion of consecutive activities: *(fig. 16) If $p_{i_1}, p_{i_2}, \ldots, p_{i_m}$ are m consecutive places,* i.e. $^\circ p_{r+1} = p_r{}^\circ, \forall r \in \{i_1, \ldots, i_{m-1}\}$*, they are substituted by a unique place* $p_{i_1,i_2,\ldots,i_m}$.

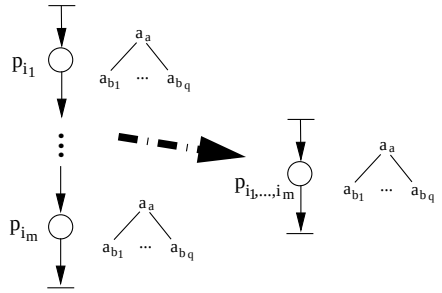


Figure 16: Rule 3 and its effect on hierarchy

Rule 3 is a transposition of the substitution rule for consecutive places in ordinary Petri nets. It preserves the level of agenticity: the token is not modified. The effect on $\mathcal{A}$ is fusing the m lines corresponding to $p_{i_1}, \ldots, p_{i_m}$ and suppressing the $m-1$ columns corresponding to the relevant transitions.

Rule 4 — Fusion of choices between activities: *(see fig. 17) If $p_{l_1}, p_{l_2}, \ldots, p_{l_m}$ are m possible places,* i.e. $^{\circ\circ}p_r = {}^{\circ\circ}p_s$, $p_r{}^{\circ\circ} = p_s{}^{\circ\circ}$, $^\circ p_r \neq {}^\circ p_s$, $p_r{}^\circ \neq p_s{}^\circ$, $\forall (r,s) \in \{l_1, \ldots, l_m\}, r \neq s$*), they are fused into a single place* $\tilde{p}_{l_1,\ldots,l_m}$.

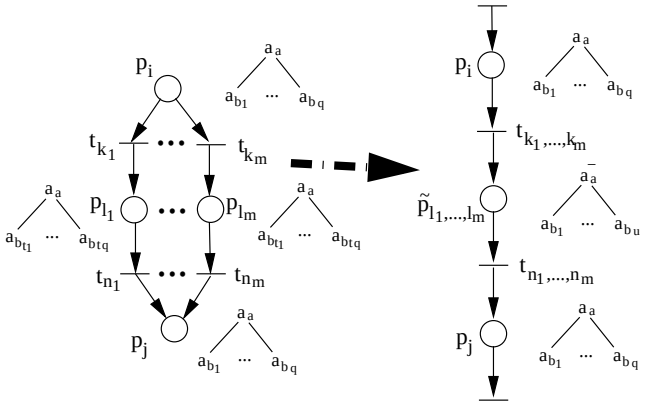


Figure 17: Rule 4 and its effect on hierarchy

Rule 4 preserves the level of agenticity. However the token is modified so as to bear, if needed, the different possible agenticity sub-hierarchies. The agent will be tagged as encompassing multiple possible organising structures. The effect on $\mathcal{A}$ is to suppress all lines and columns corresponding to the m choices but the l_1th line and the two columns corresponding to t_{k_1} and t_{n_1}.

Rule 5 — Fusion of parallel activities: *(see fig. 18) If $p_{i_1}, p_{i_2}, \ldots, p_{i_m}$ are m places in parallel,* i.e. $^\circ p_r = {}^\circ p_s$, $p_r{}^\circ = p_s{}^\circ, \forall (r,s) \in \{i_1, \ldots, i_m\}$*), they are replaced by a single place* $p_{i_1,i_2,\ldots,i_m}$.

Rule 5 is derived from the suppression rule for implicit places in ordinary Petri nets. It decreases the level of agenticity by 1: the structure born by the token is shifted upwards. Since parallel activities have the same input and output transitions (*i.e.* have the same line in $\mathcal{A}$) the effect on $\mathcal{A}$

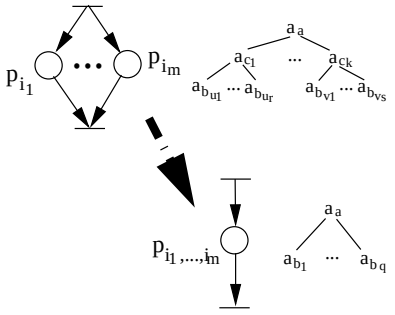


Figure 18: Rule 5 and its effect on hierarchy

is to keep a unique line for the structure, for instance the line corresponding to p_{i_1}.

Rule 6 — Reduction of agent transfer: *(see fig. 19) If p_{i_1}, p_{i_2}, p_{k_1} and p_{k_2} are the four places of a transfer structure through p_t,* i.e. $p_{i_1}{}^\circ = {}^\circ p_{k_1} = {}^\circ p_t$ *and* ${}^\circ p_{k_2} = p_{i_2}{}^\circ = p_t{}^\circ$, *they are reduced into two separate branches with p_{i_1},p^*_{k1} and p^*_{i2}, p_{k_2}.*

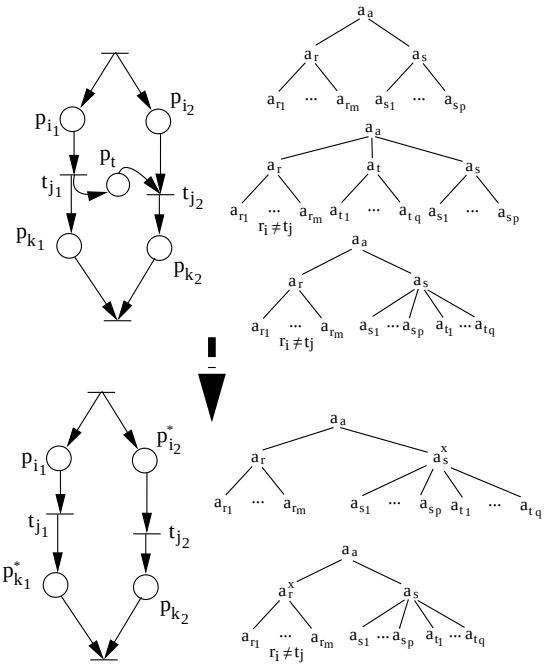


Figure 19: Rule 6 and its effect on hierarchy

Rule 6 does not decrease the level of agenticity but modifies the contents of the structure: the structure born by the token is transformed so that the transferred agents are passed on. The father agents corresponding to each branch are tagged as operating a transfer. As a matter of fact the reduction is performed by splitting the transfer place p_t and then simultaneously applying rule 1 and rule 2 on the two separate branches of the structure. The effect on $\mathcal{A}$ is to suppress the line corresponding to p_t.

Reduction and Projection: from Team Plan to Individual Plans by the Example

Reduction

The rules are iteratively applied. At the first step rules 1 and 2 are applied. For any following step rule 5 is applied first on all possible parallel structures. If no parallel structure has been reduced rule 4 is applied. If the previous two rules do not apply rule 6 reduces transfer structures. Then rule 3 compresses the sequences without modifying the level of agenticity: the reduction of all sequences ends the current step in the algorithm. The algorithm stops when the net is reduced to a single place. Hence the process reduces the team plan and builds the dynamic hierarchy of agenticity.

Algorithm 1 — Reduction of team plan

- **Initialisation:**
  1. while possible, *apply rule 1;*
  2. while possible, *apply rule 2;*
- **Iterate:**
  1. *if* ∃ *parallel structure ,* while possible, *apply rule 5;*
  2. *elseif* ∃ *choice structure,* while possible, *apply rule 4;*
  3. *elseif* ∃ *transfer structure,* while possible, *apply rule 6;*
  4. while possible, *apply rule 3;*
- **Until:** *the net is reduced to a single place.*
- **End**

The reduced places are stored along with their substitutes at each step. The algorithm is traced back so that each reduction place is hierarchically unfolded and is linked as the father of its reduced places. The resulting plan then consists of a hierarchical Petri net whose levels correspond to the levels of agenticity in the team. Each place develops into a sub-net of higher agenticity. The tokens in the subnet hold the agents performing the activities corresponding to the marked places as well as the children of these agents.

Figure 20 shows an example of a hierarchical team plan. The plan that appears in figure 5 is gradually reduced in order to yield a single-place Petri net. Let us consider the marking in greater details. p_4 and p_5 are parallel activities. Their tokens are similar and bear hierarchies respectively AAA and AAB and their children a, b and g. They are reduced into $p_{4,5}$ according to rule 5. The resulting token bears the piece of hierarchy AA with its children AAA and AAB. At the next level several reductions are possible. First rule 3 is applied on two sequences. $p_{8a} - p_{8b}$ is reduced into p_8 and $p_{4,5} - p_9$ becomes $p_{4,5,9}$. Then on one hand $p_{4,5,9}$, p_6, p_{10}, p_{11} and p_{12} show a transfer structure: they are reduced into $p^* = p^*_{4,5,6,9,10,11,12}$ according to rule 6, rule 3 and rule 5. In that structure tokens bear from left to right AA and its children AAA and AAB, and AB and its children a, d and g. On the other hand p_8 is an alternative to p_7. They are reduced into $\tilde{p}_{7,8}$ according to rule 4. The token is not changed while moving through the sequence and bears b, c, e, f and g. Sequences $p_2 - p^*$ and $p_3 - \tilde{p}_{7,8} - p_{13}$ are aggregated into respectively $p_{2,*}$ and $p_{3,\tilde{7,8},13}$ according to rule 3. At this stage the structure resulting from all previous reductions bears these two parallel activities. The

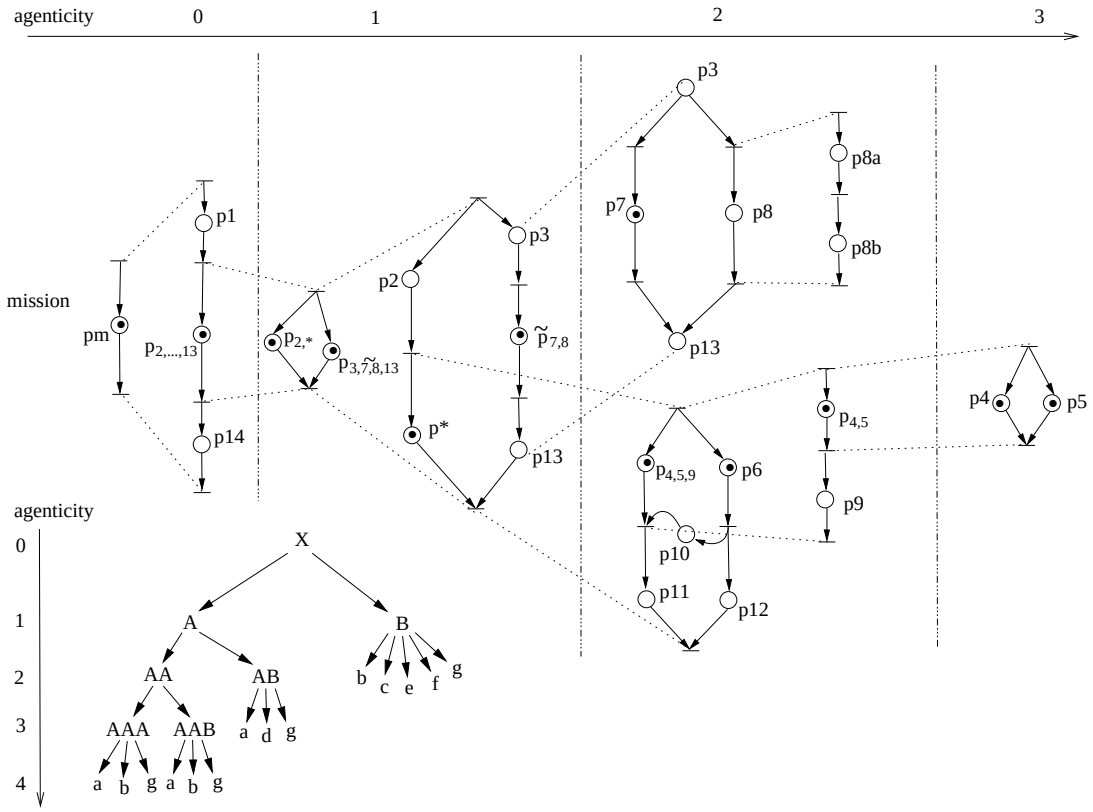


Figure 20: A hierarchical team plan with agenticity hierarchy tokens

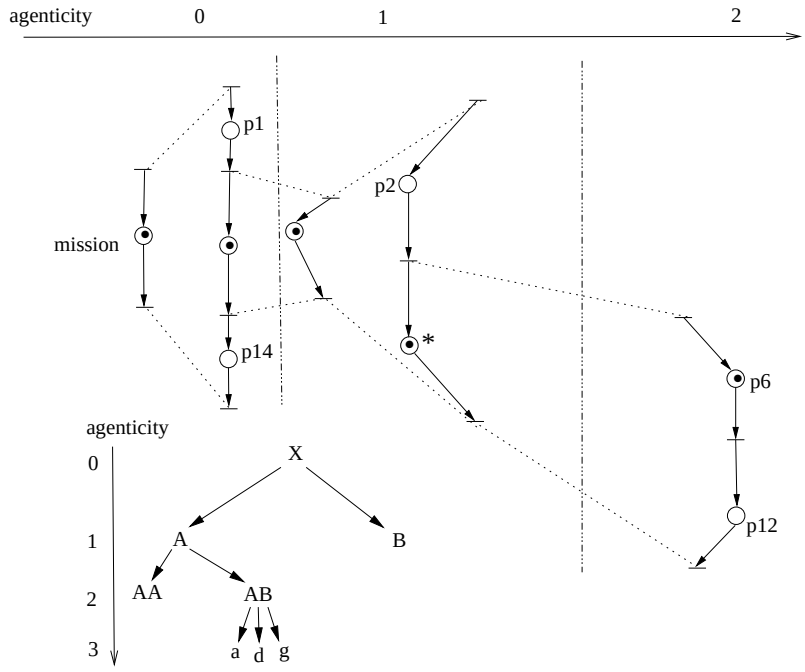


Figure 21: Projection of the team plan on agent d

structure is reduced into $p_{2,\ldots,13}$ using rule 5. The last stage of the reduction concatenates sequence $p_1 - p_{2,\ldots,13} - p_{14}$ into a single place p_m that represents the mission. The token in the sequence is composed of the hierarchy $team$ and its children. For p_1 the children are a, b, c, d, e, f and g. For $p_{2,\ldots,13}$ they are A and B.

The Petri net in figure 5 in fact corresponds to the *detailed global plan* built from the leaf-places of the hierarchical team plan in figure 20.

Projection

The hierarchical structure of the team plan now allows the agents' individual plans to be derived. This is done through a projection mechanism.

Definition 15 *(Projection) the projection of the team plan on agent a_i is an agent plan whose hierarchy of places has been cut to level $Ag_X(a_i)$ and the hierarchies of agenticity are cut to level $Ag_X(a_i)$: $deg(\mathcal{H}_X(a_i)) = Ag_X(a_i)$.*

Definition 16 *(Agent plan) the plan of agent a_i consists in the path of a_i's token in the detailed global plan and all levels above. The corresponding activities all involve a_i or its ancestors in the agenticity hierarchies.*

The projection of the team plan on agent a_i consists in isolating the places at all levels of agenticity in which a_i is involved and extracting the hierarchies of places and of agenticity associated to the places. This definition extends the projection operator in coloured Petri nets to hierarchical nets. In the detailed global plan (see fig. 5) the projection on agent a_i eliminates the other colours, *i.e.* all the other agents, and prunes the branches of the Petri net that are not coloured by a_i. The result of the projection is the Petri net corresponding to the leaves of the hierarchical plan of a_i.

Let us unfold the previous example. Figure 21 shows the agent plan for the elementary agent d. At each level the team plan Petri net has been pruned so that the remaining places involve d or its ancestors. One can notice that the same operation can be performed locally for agent AA. Locality is a consequence of the fugacity of AA due to its being a composite agent. In this case locality means that the projection on AA can only be performed on $p_{4,5,9}$ and p_{11} and their ancestors p^*, $p_{2,*}$, $p_{2,\ldots,13}$ and p_m.

Conclusion

In the general framework of agents carrying out a mission specified in terms of objectives, a Petri net-based representation of team plans is presented. In this approach agents are hierarchically organised in a team. Each node in the agenticity hierarchy can be regarded as an agent. The plan itself is represented by a hierarchical Petri net whose places are agent's activities. The organisation of the team dynamically changes as the marking of the net evolves. Therefore the team plan can be used for team activity monitoring.

From the team plan a projection operator allows to derive individual plans so that each elementary agent knows for any activity its interacting agents. The information is held in the tokens of its plan as sub-hierarchies. The conjunction of individual plans allows distributed team coordination.

The distribution of the information at all levels of agenticity in each agent may facilitate team management. In particular, in the context of teams of robots, it may help in dynamically responding to an unforeseen event, such as a failure or an external action. A modification to the initial plan — a repair — will be provided, involving agents at the most local level possible. Current and future works concern the development of EΛAIA, a Petri net-based decision architecture for local replanning within the team (Bonnet-Torrès 2005).

References

Bonnet-Torrès, O., and Tessier, C. 2005. From team plan to individual plans: a Petri net-based approach. In *AAMAS'05*.

Bonnet-Torrès, O. 2005. Local replanning within a team of cooperative agents. In *ICAPS'05 Doctoral Consortium*.

Chanthery, E.; Barbier, M.; and Farges, J.-L. 2004. Integration of mission planning and flight scheduling for unmanned aerial vehicles. In *ECAI'04 - Workshop on Planning and Scheduling: Bridging Theory to Practice*.

Christensen, S., and Petrucci, L. 1992. Towards a modular analysis of coloured Petri nets. In *ATPN'92*, 113–133.

Cohen, P., and Levesque, H. 1990. Intention is choice with commitment. *Artificial intelligence* 42:213–261.

Cohen, P., and Levesque, H. 1991. Teamwork. *Noûs* 25(4):487–512.

El Fallah-Seghrouchni, A., and Haddad, S. 1996. A recursive model for distributed planning. In *ICMAS'96*.

El Fallah-Seghrouchni, A.; Degirmencyan-Cartault, I.; and Marc, F. 2004. Modelling, control and validation of multi-agent plans in dynamic context. In *AAMAS'04*.

Erol, K.; Hendler, J.; and Nau, D. 1994. HTN planning: complexity and expressivity. In *AAAI'94*, 1123–1128.

Grosz, B., and Kraus, S. 1996. Collaborative plans for complex group action. *Artificial Intelligence* 86(2):269–357.

Huber, P.; Jensen, K.; and Shapiro, R. 1989. Hierarchies in coloured Petri nets. In *ATPN'89*, 192–209.

Jensen, K. 1997. *Coloured Petri nets. Basic concepts, analysis methods and practical use*. Monographs in Theoretical Computer Science. Springer-Verlag, 2nd edition.

Kristensen, L. 2003. Using coloured Petri nets to implement a planning tool. 4th Advanced Course on Petri Nets.

Lakos, C. 1995. From coloured Petri nets to object Petri nets. In *ATPN'95*, 278–297.

Murata, T. 1989. Petri nets: properties, analysis and applications. In *Proc. of the IEEE*, volume 77-4, 541–580.

Rao, A., and Georgeff, M. 1995. BDI agents: from theory to practice. In *ICMAS'95*.

Shoham, Y. 1993. Agent-oriented programming. *Artificial intelligence* 60:51–92.

Tambe, M. 1996. Teamwork in real-world, dynamic environments. In *ICMAS'96*.

Tambe, M. 1997. Towards flexible teamwork. *Journal of Artificial Intelligence Research* 7:83–124.