"Unrolling" Complex Task Models into MDPs

Robert P. Goldman
SIFT, LLC.
rpgoldman@sift.info

David J. Musliner
Honeywell International
david.musliner@honeywell.com

Mark S. Boddy
Adventium Labs
mark.boddy@adventiumlabs.org

Edmund H. Durfee and **Jianhui Wu**
University of Michigan
{durfee, jianhuiw}@umich.edu

Introduction

Markov Decision Processes (MDPs) provide a unifying theoretical framework for reasoning about action under uncertainty. For some domains (e.g., navigation in discrete space), MDPs may even provide a direct path to implementing a solution. However, many AI approaches, such as domain-independent planning, rely on richly expressive task models. For such problems, one way to proceed is to provide a translation from more expressive task models into MDP representations. We have developed a technique for "unrolling" such task models into an MDP state space that can be solved for an (approximately) optimal policy.

In this paper, we describe how we have built a decision-theoretic agent that solves planning/scheduling problems formulated in a dialect of the TÆMS hierarchical task language. One problem with bridging from TÆMS (and other expressive languages) to MDPs is that these other languages may not make the same simplifying assumptions as those behind MDPs. For example TÆMS task models have non-Markovian aspects in which actions taken have effects only after a delay period.

The task of reformulating a TÆMS planning problem into an MDP is complicated by the fact that TÆMS actions must be scheduled against global time, and by the presence of non-Markovian constructs in TÆMS models (actions with delayed effects). In this paper we describe both how we translate these TÆMS features into MDPs and how we cope with the state space explosion that can result.

TÆMS task problems translate into *finite-horizon* MDPs, and the utility of outcomes cannot, in general, be determined until reaching the final state; in this way they are goal-focused and offer limited traction to approaches that exploit interim reward information. We also describe several optimizations that make the planning problem easier, notably aggressive pruning of actions, collapsing together equivalent states in the state space, and lazy exploration of the state space.

The COORDINATORS Problem

Our work is being done in the context of the COORDINATORS program. COORDINATORS is a research program funded by DARPA IPTO to identify, prototype, and evaluate technical approaches to scheduling and managing distributed activity plans in dynamic environments. As a motivating example, consider the following scenario. A hostage has been taken and might be held in one of two possible locations. Rescuing the hostage requires that both possible locations are entered by special forces *simultaneously*. As the activities to move personnel and materiel into place are pursued, delays can occur, or actions to achieve precursor objectives might have unexpected results (e.g., failure). COORDINATOR agent systems will be associated with the various human participants. COORDINATOR agents should monitor the distributed plans and manage them as the situation evolves, to increase their effectiveness and make them more likely to succeed.

In general, a set of COORDINATOR agents is meant to work together to maximize the reward gained by the group as a whole. In other words, the problem is to compute an effective *joint* policy for the agent society, in which the actions taken by one agent can depend on the state of the group as a whole, not just the local state of that agent. The agents are time-pressured: each agent must make timely action decisions during execution. Furthermore, the problem must be solved in a distributed fashion.

Each agent's partial model includes the actions that the agent can execute, which are stochastic, rather than deterministic, and some of the actions its peers can perform. The model also provides *partial* information about the rewards that the society as a whole will receive for reaching various states. This model is not static: the agent can receive model updates during execution. Therefore, agents must be able to manage and reformulate policies reactively.

The problems are formulated as complex hierarchical task networks, encoded in TÆMS, which we translate to MDPs. In this paper, we will focus on the translation from TÆMS into MDPs, and largely ignore the issues of inter-agent cooperation, which we leave for other papers (see, for example, (Musliner *et al.* 2006)). In the following section, we will introduce the TÆMS task network notation, before proceeding to explain how we translate it into MDPs.

C-TÆMS

COORDINATORS researchers have jointly defined a common problem domain representation (Boddy *et al.* 2005) based on the original TÆMS language (Horling *et al.* 1999). The new language, C-TÆMS, provides a semantically sound subset of the original language, representing multi-agent hierarchical tasks with stochastic outcomes and complex hard and soft interactions. Unlike other hierarchical task representations, C-TÆMS emphasizes complex reasoning about the utility of tasks, rather than emphasizing interactions between agents and the state of their environment.

C-TÆMS permits a modeler to describe hierarchically-structured tasks executed by multiple agents. A C-TÆMS task network has *nodes* representing *tasks* (complex actions) and *methods* (primitives).[1] Nodes are temporally extended: they have durations (which may vary probabilistically), and may be constrained by release times (earliest possible starts) and deadlines. Methods that violate their temporal constraints yield zero quality (and are said to have *failed*). At any time, each C-TÆMS agent can be executing at most one of its methods, and no method can be executed more than once.

A C-TÆMS model is a discrete stochastic model: methods have multiple possible outcomes. Outcomes dictate the *duration* of the method, its *quality*, and its *cost*. Quality and duration are constrained to be numbers not less than zero. Cost is not being used in the current work. Quality and cost are unitless, and there is no fixed scheme for combining them into utilities. For the initial COORDINATORS experiments, we treat quality as non-normalized utility (we will use the terms "utility" and "quality" pretty much interchangeably).

To determine the overall utility of a C-TÆMS execution trace, we must have a mechanism for computing the quality of tasks (composite actions) from the quality of their children. Every task in the hierarchy has associated with it a "quality accumulation function" (QAF) that describes how the quality of its children are aggregated up the hierarchy. The QAFs combine both logical constraints on subtask execution and how quality accumulates. For example, a :MIN QAF specifies that all subtasks must be executed and must achieve some non-zero quality in order for the task itself to achieve quality, and the quality it achieves is equal to the minimum achieved by its subtasks. The :SYNCSUM QAF is an even more interesting case. Designed to capture one form of synchronization across agents, a :SYNCSUM task achieves quality that is the sum of all of its subtasks that start at the same time the earliest subtask starts. Any subtasks that start after the first one(s) cannot contribute quality to the parent task.

The quality of a given execution of a C-TÆMS task network is the quality the execution assigns to the root node of the task network. C-TÆMS task networks are constrained to be trees along the subtask relationships, so there is a unique root whose quality is to be evaluated. The quality is the quality of that node at the end of the execution trace. C-TÆMS task networks are required to have a deadline on their root nodes, so the notion of the end of a trace is well-defined. One may be able to determine bounds on the final quality of a task network before the end of the trace, but it is not in general possible to determine the quality prior to the end, and it may not even be possible to compute useful bounds.

Traditional planning languages model interactions between agents and the state of their environment through preconditions and postconditions. In contrast, C-TÆMS does not model environmental state change at all: the only thing that changes state is the task network. Without a notion of environment state, in C-TÆMS task interactions are modeled by "non-local effect" (NLE) links indicating inter-node relationships such as enablement, disablement, facilitation, and hindrance. Complicating matters significantly is the fact that these NLEs may have associated *delays*, violating the conventional Markov assumption. We will discuss the implications of all of these in terms of developing Markov models of the COORDINATORS problem shortly.

Figure 1 illustrates a simple version of the two-agent hostage-rescue problem described earlier. The whole diagram shows a global "objective" view of the problem, capturing primitive methods that can be executed by different agents (A and B). The agents in a COORDINATORS problem are *not* given this view. Instead, each is given a (typically) incomplete "subjective" view corresponding to what that individual agent would be aware of in the overall problem. The subjective view specifies a subset of the overall C-TÆMS problem, corresponding to the parts of the problem that the local agent can directly contribute to (e.g., a method the agent can execute or can enable for another agent) or that the local agent is directly affected by (e.g., a task that another agent can execute to enable one of the local agent's tasks). In Figure 1, the unshaded boxes indicate the subjective view of agent-A, who can perform the primitive methods Move-into-Position-A and Engage-A. The "enable" link indicates a non-local effect dictating that the Move-into-Position-A method must be completed successfully before the agent can begin the Engage-A method. The diagram also illustrates that methods may have stochastic expected outcomes; for example, agent-B's Move-into-Position-B method has a 40% chance of taking 25 time units and a 60% chance of taking 35 time units. The :SYNCSUM QAF on the Engage task encourages the agents to perform their subtasks starting at the same time (to retain the element of surprise).

Markov Decision Processes and C-TÆMS

Given a (fixed) C-TÆMS task network and the fact that method outcomes are stochastic, we frame the COORDINATOR problem as building a *policy* that dictates how each agent in the network chooses methods at every point in time. In earlier work on TÆMS, the objective was to find a satisfactory balance among some combination of quality, cost, and duration (Wagner, Garvey, & Lesser 1998), and research focused on how to define a good balance among these competing objectives. C-TÆMS problems, by contrast, focus on the problem of optimizing the cooperative action scheduling of a multi-agent system under complex timing and interaction constraints. In this paper we will primarily be concerned with the local problem, of finding a policy for a single agent that determines how that agent will choose which

[1] The terminology is somewhat unfortunate, since conventional HTN planners refer to their composite actions as *methods* and their primitives as operators.

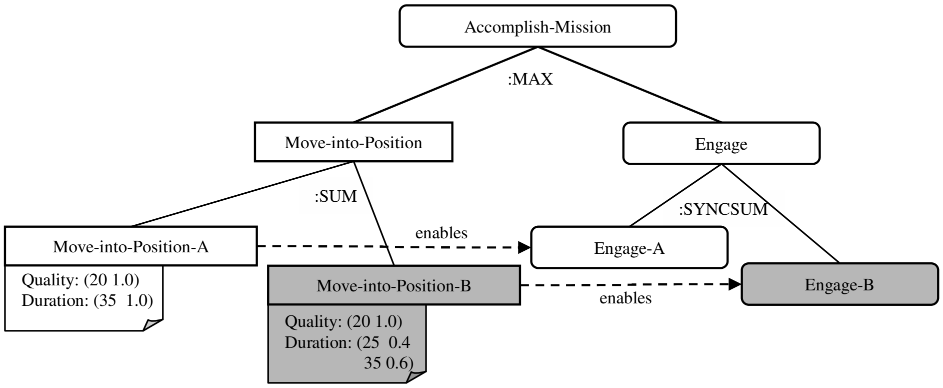

Figure 1: A simple C-TÆMS task network for two agents, illustrating some of the representation features. Some details have been omitted for brevity.

method to execute at each point in time.

We assume that the reader has a grasp of the basic definitions of Markov Decision Processes; we recommend Puterman's text (Puterman 1994) for more specifics. Briefly, an MDP is akin to a finite state machine, except that transitions are probabilistic, rather than deterministic or nondeterministic. Agents may also receive reward (which may be either positive or negative) for entering some states. Typically, this reward is additive over any trajectory through the state space (some adjustments are needed in the case of MDPs of infinite duration). The solution to an MDP is a *policy* — an assignment of action choice to every state in the MDP — that maximizes *expected utility*. MDPs' advantages are that they offer a sound theoretical basis for decision-making and action under uncertainty, and that there are relatively simple, polynomial algorithms for finding optimal policies.[2]

An agent's C-TÆMS task model specifies a *finite-horizon* MDP. The problems are finite-horizon because C-TÆMS problems have finite duration, with no looping or method retries. However, the MDP tends to be quite large for even modest-sized C-TÆMS problems because of the branching factor associated with uncertain outcomes, and because of the temporal component of the problem. For example, even a single applicable method with three possible durations and three possible quality levels gives us a branching factor of nine. In addition, time is a critical aspect of TÆMS problems: methods consume time and NLEs can have associated delays (so WAIT is often a useful action alternative). Furthermore, an agent can always abort a method that it is executing, and choose to start a different method. So the branching factor is never less than two at every time tick, in a full consideration of the problem. Multi-agent C-TÆMS MDPs are even worse.

The COORDINATORs problem differs from most problems treated as MDPs in three important ways. First, the problem is inherently distributed and multi-agent, so that in the objective view, multiple actions can be executed simultaneously. For this reason, if one were to formulate a centralized COORDINATORs problem directly as an MDP, the action space would have to be a tuple of assignments of actions to each agent. As one would expect, this causes an explosion in the state space of the problem. This problem we address by having each agent develop its own policy (although it may negotiate with other agents). A second difference is that the actions in the COORDINATORs domain are temporally extended, rather than atomic. Such "durative" actions can be accommodated, but only at the cost of a further explosion in the state space. Finally, the C-TÆMS planning problem is not straightforwardly Markovian. For example, the delays associated with NLEs such as "enables" links require the MDP state to hold some execution outcome history. In the following sections we will describe how we tackle the latter two issues, unrolling the C-TÆMS task model into an MDP whose state has been augmented to handle non-Markov constructs, and how we cope with the large resulting state space. We do not have space here to discuss how we decompose the multi-agent problem into multiple, interacting, single-agent problems (see (Musliner *et al.* 2006)).

Unrolling C-TÆMS task nets

For any agent, the C-TÆMS task network implicitly defines a possible state space and a transition function that maps individual states to possible future states. Our system reasons about C-TÆMS task networks by "unrolling" them into MDPs that make this implicit state space explicit. In this section, we explain how we have derived a state representation for C-TÆMS that satisfies the Markov constraint. We then explain how the transition function is defined and describe our unrolling algorithm.

State space To meet the constraints of having a well-formed MDP, we must incorporate in each state enough information to make the state transition function be a function of only the current state and the chosen action. We must also incorporate into each state enough information that we can evaluate the quality/utility function for the task network.

To compute the final score of any C-TÆMS task network over an execution trace, it is sufficient to know the quality,

[2] But polynomial in the size of the state space, which is typically exponential in the size of the input!

start, and completion time of every method the agent has executed in the trace. The quality of every task node can be computed from the quality of its children, so there is no need to store quality information about internal nodes.[3] We need to know the start and completion times of each method to determine whether or not the method has satisfied its time constraints (and hence determine its quality). In practice, it suffices to record the completion time and quality of each executed method; the effect of the release time can be captured in the quality, since a method started before its release time will achieve zero quality.

Because methods have duration and release time constraints, we must record "wall-clock time" in the state of the agent. That is, in order to know the possible outcomes of executing a method (recall that the outcome is characterized by duration and quality), we must know when the action is starting. When determining the outcome distribution for a method M, we must also know the quality of every node n for which there exists an NLE, $n \rightarrow M$, and we must know the quality of n at the time $now - \text{delay}(n \rightarrow M)$.[4] The need to reason about NLEs is another reason we must record the completion times of all methods. Because of the delays in the NLE effects, we must be able to compute a quality for the tail nodes at different times in the trace.[5]

Given the above features of C-TÆMS, we can define the state of a C-TÆMS agent as a tuple $\langle t, M \rangle$, where t is the current time, and M is a set of method outcomes. If $\mathcal{M}$ is the set of methods a TÆMS agent can execute, we can assign to $\mathcal{M}$ an arbitrary numbering $1...n$, for $n = |\mathcal{M}|$. Then M is a set of tuples $\langle i, \sigma(i), \delta(i), q(i) \rangle$: the index of the method, its start time, duration, and quality.[6] This information is sufficient (but not always all necessary) to give a state space that has the Markov property.

Transition function The C-TÆMS task network, with its duration and quality distributions, defines a transition function. For example, if an agent executes a method i starting at time t, yielding a duration $\delta(i)$ and a quality $q(i)$, that is a state transition as follows:

$$\langle t, M \rangle \rightarrow \langle t + \delta(i), M \cup \{\langle i, t, \delta(i), q(i) \rangle\} \rangle$$

So the C-TÆMS task model defines a state transition distribution of the form $P(s' \mid m, s)$ for methods m and states s and s'. Note that we do *not* generate states at every tick in the system clock. Instead, we only generate states for "interesting" times, when methods start or finish.

In addition to allowing agents to carry out C-TÆMS methods, we also allow them to execute "wait" pseudo-actions. We allow agents to choose to wait until the next time at which an action becomes enabled (i.e., the next release time). That is, we allow the agent to perform wait actions of the form $\text{wait}(n)$ that cause a transition as follows:

$$\langle t, M \rangle \rightarrow \langle t + n, M \rangle$$

The wait actions are deterministic.

```
let $openlist$ = $\emptyset$
while $openlist$ do
  let* $s$ = pop($openlist$)
       $ms$ = applicable-methods($s$)
    for $m$ $\in$ $ms$ do
      ;; transitions, ts, is a set of
      ;; <probability, successor> pairs $\langle p, s' \rangle$
      let $ts$ = succs($s$,$m$)
          for $\langle p, s' \rangle$ $\in$ $ts$ do
            unless old-state($s'$) do
                add $s'$ to $openlist$
            add $\langle p, s' \rangle$ to transitions($s$)
```

Figure 2: Unrolling algorithm

Unrolling the state space Our techniques "unroll" the state space for the MDP from its initial state ($\langle 0, \emptyset \rangle$) forward.[7] From the initial state, the algorithms identify every possible method that could be executed, and for each method every possible combination of duration-and-quality outcomes, generating a new state for each of these possible method-duration-quality outcomes. Each state is then further expanded into each possible successor state, and so on. For states where no methods can apply, a "wait-method" is generated that leads to a later state where some non-wait method has been enabled (or the scenario has ended). We give a simplified pseudocode for this algorithm in Figure 2. The unrolling process ends at leaf states whose time index is the end of scenario. The code for performing this unrolling was adapted from previous state-space unrollers developed at SIFT for applications in optimal cockpit task allocation (Miller, Goldman, & Funk 2003).

Note that in order to prevent an even worse state space explosion, we must not revisit states. The state space generated in unrolling is a DAG rather than a tree, so states may be regenerated. As shown in Figure 2, we check to see if a state has previously been generated before adding it to the openlist. The following section on recognizing equivalent states shows how we can extend this state space pruning to dramatically reduce the number of states enumerated.

Another issue is that the set of enabled methods is a complex function of the current state, influenced by temporal constraints, NLEs, etc. Not only must we reason about the state to determine what methods are enabled, we also aggressively prune dominated action choices so that we don't have to enumerate the resulting states. For example, once one task under a :MIN QAF has failed (gotten zero quality) then none of the :MIN node's remaining children should ever be tried, because they cannot increase the :MIN node's

[3] But see our discussion of state equivalence later in the document.

[4] This is actually a slight oversimplification. For some NLEs we need to know the exact quality of the tail node, for others we need only know whether that quality is non-zero.

[5] Note that the quality of nodes is a montonically non-decreasing function of time.

[6] In practice, the set M is most efficiently implemented as a vector.

[7] Note that we can just as easily start unrolling *in medias res*, by starting from a state in which the agent has already executed some actions.

quality.[8] Similarly, we prune methods if they are guaranteed to run longer than their deadlines, and we abort methods early if we can determine that they will run past their deadlines. For example, if a method has three durations, $d_1 \leq\ d_2 \leq\ d_3$, and its deadline is $D = t + d_1$, the agent can execute the method, but if it fails to get an outcome with d_1, we should halt execution at D, rather than pointlessly running longer.

Even with aggressive dominance-based pruning, we are unable to fully enumerate the state spaces of even single agent COORDINATORS MDPs. These MDPs would be challenging even in the best of circumstances, but in the COORDINATORS application, we must unroll the networks and solve them under time pressure. The two techniques most important to meeting these resource constraints are pruning equivalent states — a process that provides exponential reductions in state space — and heuristically-guided, partial task model unrolling.

Equivalence folding

The size of the state space of the C-TÆMS MDP grows exponentially as it progresses. Since the growth is exponential, every state we can remove from the state space, especially early in the problem, can provide an enormous reduction in overall state space size. Accordingly, we have been working to aggressively prune the state space by identifying and folding together equivalent states. Two states are equivalent if they are identical with respect to the actions available from the current time forward and the eventual quality that will result from any execution trace starting at the two states.

Recall that all questions about an execution trace can be answered if one knows the history of method executions in that trace. That is why the state of a C-TÆMS agent can be characterized by a pair of the current time and the set of method outcomes. If that is the case, then a simple equivalence-folding can proceed by merging together states that have the same set of method outcomes (recall that these may be captured by recording the completion time and quality of the methods), and the same time index.

It is important to remember that there are two aspects to the problem of finding equivalent states in this kind of application. One aspect is the problem of matching, or verifying equivalence. The definitions of equivalence that we have outlined above provide the means for developing matching routines. However, equivalence folding will be worse than useless without an efficient solution to the problem of retrieving candidates for matching. In particular, if too many candidates are retrieved then the cost of the matching checks will overwhelm the benefit of equivalence folding. Indeed, our first experiments with equivalence folding encountered this problem until we improved our indexing. For the first, simple equivalence matching, we were able to index the states by keeping a vector of hash-tables, one hash-table for each time index. However, in order to effectively hash the states, we needed to develop a custom hash-function able to indexing arrays of floating-point numbers without an excessive number of collisions.

This first, simple approach to equivalence folding did, indeed, yield substantial speedups in the unrolling process. However, these speedups were rapidly swallowed up by large-scale problems with hundreds of C-TÆMS methods. In search of further improvements, we looked for ways to discard some methods from the agent's history, reducing the amount of state and thus permitting more states to be folded together. That is, we use a form of abstraction specifically tailored to C-TÆMS in order to map states to abstractions that omit irrelevant history, and we fold together states that are equivalent under this abstraction.

Our approach to exploiting the task network structure to abstract the states relies on two features of C-TÆMS QAFs: First, if we can determine the final quality of a task t, then we do not need to know about the quality of t's children to determine the final quality of the execution trace. Second, by static inspection of the task network, we can determine a point in time by which we will know the final quality of every node in the task network; the node deadlines give us that information. For any node n, we define the "latest useful time" at which we need to know its status *in order to compute the quality of its parent* as $\mathrm{lut}_{tree}(n)$. To emphasize that we do not use an exact value of $\mathrm{lut}_{tree}(n)$, but only an easily-computable upper-bound, we will use $\widehat{\mathrm{lut}}_{tree}(n)$.

Unfortunately, the C-TÆMS QAFs and the rules for computing quality up the task network are not sufficient to give us a simple, sound abstraction relationship. The abstraction is complicated by the presence of NLEs and their delays. That is, we cannot "forget" the quality over time of a given node until the time when we can determine that this information is no longer needed to compute the status of the NLEs out of that node. We refer to this quantity as $\mathrm{lut}_{NLE}(n)$. As with $\mathrm{lut}_{tree}(n)$, we can compute an approximation, $\widehat{\mathrm{lut}}_{NLE}(n)$ using information that is statically available in the task network. Essentially, for n this number is

$$\widehat{\mathrm{lut}}_{NLE}(n) = \max_{n' \mid n \overset{\mathrm{NLE}}{\rightarrow} n'} \mathrm{deadline}(n')$$

Armed with these facts, we can define the latest useful time for information about a node, n, as follows:

$$\widehat{\mathrm{lut}}(n) = \max(\widehat{\mathrm{lut}}_{tree}(n), \widehat{\mathrm{lut}}_{NLE}(n))$$

We may now use $\widehat{\mathrm{lut}}(n)$ to define an abstracted equivalence relationship between two states, s and s'. To a first approximation, this relation is as follows:

- s and s' are equivalent if $\mathrm{time}(s) = \mathrm{time}(s')$ and they are *method-equivalent* and they are *task equivalent*;
- s and s' are method-equivalent if, for all $m \in methods$, either $\widehat{\mathrm{lut}}(m) < \mathrm{time}(s)$ or $\text{end-time}(m, s) = \text{end-time}(m, s')$ and $\mathrm{quality}(m, s) = \mathrm{quality}(m, s')$.[9]
- s and s' are task-equivalent if, for all $t \in tasks$, either $\widehat{\mathrm{lut}}(t) < \mathrm{time}(s)$ or $\mathrm{quality}(t, s) = \mathrm{quality}(t, s')$.

[8] Actually, if a child node can enable some other local or nonlocal method it may still have utility. This sort of effect makes it quite challenging to accurately assess dominance.

[9] We have suppressed some complications here.

	No overlap		Moderate overlap	
	New	Original	New	Original
States	13.3K	49.1K	86.4K	356K
Terminal states	12	2187	13	15.3K
MDP Unroll time (secs)	16.5	93.5	72	3995
Find policy time (secs)	0.24	0.86	417	(killed)
Reward	12.2	12.2	12.2	12.2

Table 1: Effects of improved equivalence folding as a function of problem flexibility

Some remarks on the implementation:

- Since the $\widehat{\mathrm{lut}}(n)$ information is static, it can be computed at task model load time and then reused.
- We may represent the "task quality state" using a vector of quality values, one for each node. The quality of a task will remain in the vector until after its LUT, at which time its parent's final quality will be known and its own quality can be forgotten.
- We can represent LUT information for "forgetting" states using bit masks for the quality state vectors, and these bit masks can be lazily computed and reused. The bit masks provide an efficient implementation of the abstraction.
- "Quality states" may be efficiently compared using bitwise operations.
- We have developed an effective special-purpose hash function for these vectors.

Our preliminary experiments indicate substantial speedups from using this new, abstract equivalence relationship. Table 1 shows the improvement that was achieved on two versions of a representative example of a COORDINATORs scenario, which is small enough for generation of a *complete* policy (considering all relevant action choices, across the entire planning horizon). This scenario involved four separate windows of activity, in each of which there are tasks that must be successfully accomplished for the agent to achieve a positive reward. The agent has a total of 7 possible actions it can take at any given time step, though for any given time step, several of them will be ruled out as rational choices by temporal constraints. The two sets of results entitled "No overlap" and "Moderate overlap" were generated to test the behavior of the new equivalence reasoning in the presence of increased method choice, achieved by relaxing the temporal constraints on the windows of activity such that they go from disjoint, to overlapping by approximately 20% of the width of each window. In each case, we report the number of states generated in a complete unrolling of the state space, subject to folding together states recognized as equivalent, the number of terminal states (distinguishable states at the end of the scenario), the time to unroll the state space, and the time required for Bellman backup to generate a policy for that state space. The "Reward" entry is present to ensure that the stronger form of equivalence reasoning is not altering the problem being solved (e.g. by combining states that are not truly equivalent).

The results show a pronounced improvement for the new equivalence reasoning over the original formulation, which increases with an increasing freedom of action choice (and the concomitant increase in the size of the policy). The reduction in the number of terminal states in the unrolled state space generated is particularly striking, and relevant in that it directly affects the complexity of finding a policy.

We have also demonstrated that the new equivalence reasoning can provide a superlinear advantage over the old version of equivalence (which was itself very much better than an naive approach using no equivalence reasoning), in terms of how solution times scale with problem size. The results in Table 2 shows the difference in how old and new equivalence scale with problem size. The problem template used for these experiments consists of a variable number of windows, each containing multiple methods that the agent may execute. The reason for the difference in growth rates between the two approaches is that new equivalence does not need to track *how* a given level of quality was achieved for a task whose execution time is past, just what the quality achieved was. While this particular problem template was chosen to illuminate a particular structure, we expect real scenarios to have this character to at least some degree: it will frequently not matter exactly how a given result was achieved, just what it was, that was achieved.

So far, we have only done what one might refer to as "static" equivalence folding; equivalence folding using information that is independent of the actual evolution of an execution trace. However, there is additional, dynamic information that could be employed for further abstraction. For example, once we know that one child of a MIN task has failed, then we know that this task will have a final quality of zero, and that we do not need any information about other children of the task in order to compute the quality of the task or its parents.[10] It is not clear to us, however, to what extent we can actually get a net benefit by employing this information — computing this kind of abstraction may be more expensive than it is worth.

To date, we have only worked with exact state equivalence. Folding together states that are only approximately equivalent could yield even greater savings. We hope to expand to do this in the future. Unfortunately, the problem of identifying good approximations for equivalence is considerably complicated by the fact that the C-TÆMS QAFs are so expressively powerful. It is a trivial task to construct a C-TÆMS task network whose quality is only bounded from 0 to n for an arbitrarily large n until the final time step, when all quality is gained or lost. Determining what quality outcomes are effectively collapsible will involve quite complex heuristic reasoning.

"Informed Unrolling"

Even with the optimizations mentioned above, the state space of even a single COORDINATOR agent's MDP may be too large for full enumeration. Since full enumeration of even single-agent C-TÆMS MDPs is impractical, we have

[10] But there may still be NLEs out of the children that have to be reckoned with.

	Number of windows in scenario											
	1		2		3		4		5		6	
	New	Orig.	New	Orig.	New	Orig.	New	Orig.	New	Orig.	New	Orig.
States	8	13	114	817	120	4087	126	20437	132	102187	138	killed
Terminal states	1	5	1	70	1	350	1	1750	1	8750	1	killed
MDP Unroll time (msec)	10	10	60	200	60	1470	30	7560	40	200K	40	killed
Find policy time (msec)	5	5	7	23	8	56	3	362	4	10K	4	killed
Reward	8.0	8.0	8.0	8.0	8.0	8.0	8.0	8.0	8.0	8.0	8.0	killed

Table 2: Effects of improved equivalence folding on scaling

developed a technique for heuristic enumeration of a subspace of the full MDP. Our *informed unroller* (IU) algorithm prioritizes the openlist of states waiting to be unrolled based on an estimate of the likelihood that the state would be encountered when executing the optimal policy from the initial state. The intent is to guide the unrolling algorithm to explore the most-probable states first.

One cannot determine the probability of reaching a state without considering the policy followed by the agent. Therefore, the IU intersperses policy-formulation (using the Bellman backup algorithm) with unrolling. This means that we must be able to find an (approximately) optimal policy for partial MDP state spaces, which means we must have a heuristic to use to assign a quality estimate to leaf nodes in our search that do not represent complete execution traces.

We have developed a suite of alternative heuristics for estimating intermediate state quality, since the problem of finding a good heuristic is quite difficult. Currently the most consistent performance is achieved by a heuristic that emphasizes unrolling the highest (estimated) probability states first. However, computing the heuristic function (which involves a full Bellman backup) and sorting the openlist (which may be quite large) is an expensive operation. Therefore we constrain the openlist sorting activity in two ways. First, the sorting only occurs when the size of the openlist doubles,[11] and second, once the sorting function takes more than a certain maximum allocated time (e.g., one second), it is never performed again. This has the net effect of sorting the openlist more often early in the search, when focus is particularly important, and less often or never as the space is unrolled farther and probability information becomes both less discriminatory (because the probability mass is distributed over a very large set of reachable edge nodes) and focus becomes less critical (because the agent can refine its model/policy during execution time). We are currently revising our implementation of Bellman backup to permit partial results to be reused across restarts, which will enable us to use our heuristic more extensively.

The informed unroller work is at an early stage, but early results from the evaluation against a complete solution are promising. For example, in Figure 3 we show a comparison of the performance of the informed unroller against the complete unrolling process. In these small test problems, the informed unroller is able to find a high-quality policy quickly and to return increasingly effective policies given more time. This allows the IU-agent to flexibly trade off the quality and timeliness of its policies. The current version of the IU does not support repeated, incremental unrolling of the state space during execution. However, we are actively working to build a new version, and integrate it into our COORDINATORS agent.

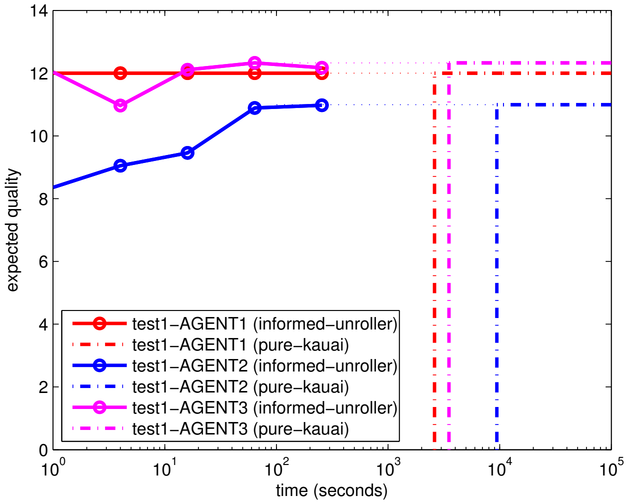

Figure 3: The Informed Unroller can find near-optimal policies much faster than building the complete MDP.

The IU approach is related to the "approximate dynamic programming" algorithms discussed in the control theory and operations research literature (Bertsekas 2005). These approaches derive approximate solutions to MDP-type problems by estimating, in various ways, the "cost to go" in leaf nodes of a limited-horizon portion of the full state space. While our exploration of the literature is not yet complete, initially we believe that a key difference in our IU approach is the notion of time-dependent horizon control and unrolling-guidance (vs. just estimation of leaf-node reward for policy derivation).

The IU method is a special case of the find-and-revise algorithm schema (Bonet & Geffner 2006) (which is a generalization of algorithms such as LAO^* (Hansen & Zilberstein 2001)). $LDFS$-family algorithms use knowledge of the initial state(s) and heuristics to generate a state subspace from which a policy can be abstracted. A find-and-revise algorithm finds a state in the network for which the current value estimate is inaccurate, and revises the value for that state (e.g., by generating successors, and propagating the value

[11] This threshold could, of course, be tailored.

functions backwards in standard MDP fashion).

Our technique differs from the general case, and its instances, in substantial ways. LAO^* generates a state subspace from which the optimal policy can be provably derived. The IU, on the other hand, executes online, and might lack enough time to enumerate such a state subspace even if it knew exactly which states to include. The IU is an anytime algorithm, unlike LAO^*, which runs offline. For this reason, the IU makes no claims about policy optimality; indeed, it is not even guaranteed to generate a closed policy.

The general find-and-revise algorithm family can provide guarantees weaker than those of LAO^*, but those guarantees rely on having an admissible heuristic value function for states that have not been fully explored. We have discussed earlier why an admissible heuristic is difficult to come by for the COORDINATORS domain. Any truly admissible heuristic is likely to be so vacuous as to cause search to founder. Furthermore, even if we had an admissible heuristic, it is not at all clear that the IU should use it. An admissible heuristic will tend to push the policy expansion to explore states where it is **possible** that the optimum will be found, in order that we not miss the optimum. However, the IU is operating in a time-pressured domain. So we should not be encouraging the system to move towards promising unexplored areas — that will tend to leave the agent with a policy that is broad but shallow, and virtually guarantee that it will "fall off policy" during execution. Instead of admissibility, we must find a heuristic function that will cause the agent to tend to build policies that trade off considerations of optimal choice against completeness/robustness of the policy. It is possible that this heuristic should be time-dependent — as the agent runs out of time for policy development, the IU's heuristic should focus more on robustness and less on optimality.

Conclusion and Future Directions

For MDP-based techniques to reach their full potential, AI must develop techniques for translating its complex representational schemes into what is, after all, only an enhanced finite state machine. Past successes in other AI fields, such as the advances offered by compiling PDDL planning representations in SAT problems, show that such translation methods can enable substantial practical advances. In this paper we have described techniques for compiling such an expressive representation, C-TÆMS, into MDPs. We have shown how to augment the state space to handle non-Markovian constructs, and provided techniques for handling the state space explosion that results, including techniques for aggressively folding together equivalent states, and heuristic state space enumeration. These techniques exploit particular features of the underlying task model and of the translation process. We have used earlier forms of these techniques to handle another multi-agent task model, and we believe that they may be adapted for many other domains.

Acknowledgments

This material is based upon work supported by the DARPA/IPTO COORDINATORS program and the Air Force Research Laboratory under Contract No. FA8750–05–C–0030. Any opinions, findings and conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA, the U.S. Government, or the Air Force Research Laboratory.

References

Bertsekas, D. P. 2005. Dynamic programming and suboptimal control: A survey from ADP to MPC. In *Proc. Conf. on Decision and Control*.

Boddy, M.; Horling, B.; Phelps, J.; Goldman, R. P.; and Vincent, R. 2005. C-TÆMS language specification. Unpublished; available from this paper's authors.

Bonet, B., and Geffner, H. 2006. Learning depth-first search: A unified approach to heuristic search in deterministic and non-deterministic settings, and its application to MDPs. In Long, D.; Smith, S. F.; Borrajo, D.; and McCluskey, L., eds., *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling*, 142–151.

Hansen, E. A., and Zilberstein, S. 2001. LAO: a heuristic search algorithm that finds solutions with loops. *Artificial Intelligence* 129(1-2):35–62.

Horling, B.; Lesser, V.; Vincent, R.; Wagner, T.; Raja, A.; Zhang, S.; Decker, K.; and Garvey, A. 1999. The TAEMS white paper. Technical report, University of Massachussetts, Amherst, Computer Science Department.

Miller, C. A.; Goldman, R. P.; and Funk, H. B. 2003. A Markov decision process approach to human/machine function allocation in optionally piloted vehicles. In *Proceedings of FORUM 59, the Annual Meeting of the American Helicopter Society*.

Musliner, D. J.; Durfee, E. H.; Wu, J.; Dolgov, D. A.; Goldman, R. P.; and Boddy, M. S. 2006. Coordinated plan management using multiagent mdps. In *Working Notes of the AAAI Spring Symposium on Distributed Plan and Schedule Management*, number SS-06-04 in AAAI Technical Report.

Puterman, M. 1994. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons.

Wagner, T. A.; Garvey, A. J.; and Lesser, V. R. 1998. Criteria Directed Task Scheduling. *Journal for Approximate Reasoning* 19:91–118.