Exploiting C-TÆMS Models for Policy Search

Bradley J. Clement and Steven R. Schaffer
Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive, M/S 126-347
Pasadena, CA 91109-8099
bclement@jpl.nasa.gov, srschaff@aig.jpl.nasa.gov

Abstract

We describe our experience developing software for finding centralized multiagent optimal policies for problems specified in the C-TÆMS modeling language, which differs from more classical state-based modeling languages in specifying task relationships explicitly. This software has been used by the DARPA Coordinators program for evaluating other real-time, decentralized algorithms that do not guarantee optimal solutions. The search algorithm is AO*, but we focus discussion on our customization of the search for C-TÆMS, how we addressed uncertain duration, and the resulting challenges in trying to scale to larger problems. We will describe different techniques for reducing the number of states created and how they succeeded or failed.

Introduction

A common approach to planning under uncertainty is to search for policies represented as Markov Decision Processes (MDPs). However, scalability can be a problem since the state space often grows more than exponentially with the problem size. Planning for multiple agents is more complex in that finding optimal policies can grow more than exponentially with respect to the size of this already exponentially growing state space.

In order to simplify the complexity of optimal policy search, a common technique is to only instantiate the state space that is reachable from a known initial state (Barto, Bradtke, & Singh 1995; Hansen & Zilberstein 2001; Mausam & Weld 2005; Musliner *et al.* 2006; Wu & Durfee 2007). Much MDP planning research is based on either no language in particular (just the abstract MDP formulation) or on languages deriving from that of STRIPS (Stanford Research Institute Problem Solver), where state variables are used to describe and determine legal task interactions (Fikes & Nilsson 1971). This paper will describe some techniques specific to the C-TÆMS modeling language (Boddy *et al.* 2005), based on the TÆMS (Task Analysis, Environment Modeling, and Simulation) framework (Decker & Lesser 1993).

The C-TÆMS modeling language is similar to a Hierarchical Task Network (Erol, Hendler, & Nau 1994), but instead of using state representations, C-TÆMS models task relationships explicitly and quantitatively. The language allows probability distributions over discrete sets of action durations, quality outcomes, and relationship effects. C-TÆMS also explicitly represents concurrently executing agents.

Other recent work also discusses how to minimize policy space exploration for C-TÆMS problems (Musliner *et al.* 2006; Wu & Durfee 2007). This paper addresses the problem of finding centralized optimal solutions offline, while others describe single agent optimal algorithms used to find decentralized solutions in real time without optimality guarantees. Although not searching for policies, a constraint programming approach has been shown to optimally solve large centralized C-TÆMS problems without uncertainty (van Hoeve *et al.* 2007).

We will describe techniques that were intended to improve problem solving performance by exploiting domain knowledge captured by the C-TÆMS problem description. Some of the techniques were useful, some were not, and others were for certain kinds of problems. These techniques aimed to

- reduce the number of states explored and their actions,
- cache and reuse state evaluation computations, and
- streamline the creation of individual states.

Some of these techniques only gave a small factor of improvement on the same problems while others enabled the software to scale to larger problems. The tightly coupled problems of the DARPA Coordinators program that meet the edge of what our implementation solves involve between 6 and 14 agents, a total of 24 to 36 methods, each with 1 to 3 duration and 1-3 quality possibilities. These problems typically created a policy space of 1 to 4 million states consuming 1 to 5 GB of RAM in 10 to 45 minutes. The decentralized algorithms solve much larger problems (Musliner *et al.* 2006; Maheswaran *et al.* 2008; Smith *et al.* 2007). Although these algorithms do not guarantee optimality, for many problem sets, they all performed within a couple of percent of optimal. We personally have had much trouble trying to solve these problems by hand, needing to consult the optimal policy and still taking hours.

We implemented these strategies in a centralized planner that the DARPA Coordinators Program has used to compute optimal solutions as a benchmark to compare other distributed planning algorithms that do not make optimality guarantees. There are other algorithms that use MDP

representations for TÆMS and C-TÆMS problems and that have similarities to ours (Wagner, Raja, & Lesser 2006; Musliner *et al.* 2006; Wu & Durfee 2007). As we describe in AO* Policy Search section, our algorithm mainly differs from others in centralizing search, in handling concurrent action and uncertain duration, and in how it exploits the structure of the C-TÆMS modeling language.

We will first describe how we represent C-TÆMS problems as MDPs with states. For context, we explain our use of an AO* strategy for policy search, iteratively computing a policy while expanding the state space. We then present techniques that exploit domain knowledge in the problem description and a preliminary evaluation of their effectiveness.

MDP Representation of C-TÆMS

Instead of describing C-TÆMS in detail as is done elsewhere (Boddy *et al.* 2005), we will describe the basic modeling features and how we translate them into states of an MDP. Some of the semantics described are vague but we aim to describe them in sufficient detail to understand the results in variations of the planning algorithm.

C-TÆMS modeling features

A C-TÆMS problem is a tree of activities with executable *methods* as leaves. The methods are executed once by a specified agent and have a probability distribution over labeled *outcomes*, each of which having discrete distributions over duration and quality outcomes. Other activities are *tasks* that are each composed of a group of subactivities (tasks and/or methods).[1]

A task's quality is determined by a *quality accumulation function* (QAF) of the qualities of the subactivities. The problem is to maximize the quality accumulated by the root task. Since our algorithm is offline, our goal is to compute the optimal policy, a complete contingency plan. For example, a *max* QAF accumulates the maximum of the qualities of the subactivities, suggests that the quality of only one will affect the overall quality, so it may not make sense to execute more than one of them, depending on the quality distributions. C-TÆMS also defines *min*, *sum*, *synchronized-sum*, *sum-and*, and *exactly-one*. Without getting into details, the last two approximate the usual AND/OR semantics of hierarchical plans (Erol, Hendler, & Nau 1994), and synchronized-sum differs from sum in returning zero quality if not all tasks start simultaneously.

Another feature of the language is the representation of *non-local effects* (NLEs), effects of one activity's outcome on another's. An activity can *enable* or *disable* another when it accumulates quality $>$ zero. An activity A *facilitates* or *hinders* activity B by scaling or shrinking (respectively) B's quality linearly with A's quality by a factor chosen from a discrete distribution. The same two NLEs can similarly shrink or scale (respectively) the duration of a target activity. An NLE can also specify a *delay* in its effect. For example, if an NLE specifies that A disables B with a delay of 10 seconds, B can successfully execute if it starts less than 9 seconds after A first accumulates quality.

[1] A more recent version of the C-TÆMS specification includes preconditions and postconditions for methods (like STRIPS) that are used for specifying location state and repeatable "template" methods that are available to a specified set of agents and used for movement between locations. The scope of this paper does not include these language elements.

Representing MDP states and actions

An MDP commonly represents a planning problem as a mapping of states and possible actions to possible outcome states (with respective probability and reward/quality). We represent state for C-TÆMS problems with the following values.

- method state (for each method)
  - phase of execution $\in$ {pending, active, completed, failed, aborted, abandoned}
  - start time
  - outcome
  - duration
  - quality
- time

From this information the accumulated quality for all of the tasks can be computed. The different phases of a method capture not only the basics for accurate simulation (pending, active, and completed), but some for readability/debugging (failed, aborted) and one implied intention (abandoned) that we will later show can be helpful in eliminating suboptimal action choices and avoiding repeated exploration of equivalent action sequences. By including time in the definition, a state can never be revisited after an action is taken since all subsequent states will have a later time value. This restricts the network formed by the state-action-state mapping to a directed acyclic graph.

The possible actions for the agents are simply to start or abort execution of a method. Each agent can only execute one method at a time.

This representation is sufficient for handling uncertain duration, as is a similar representation for a STRIPS descendant (Mausam & Weld 2005). Other translations of C-TÆMS to MDP are done from a single agent's perspective as part of decentralized problem solving and manage interdependencies (NLEs) between agents instead of centralizing the analysis of possible concurrent joint actions (Wagner, Raja, & Lesser 2006; Musliner *et al.* 2006; Wu & Durfee 2007).

AO* Policy Search

The planning algorithm we use is based on an A* algorithm for AND/OR graphs, AO* (Ginsberg 1987). It is a forward expansion version of Looping AO* (Hansen & Zilberstein 2001) without looping (cycling of states during execution). It is forward expansion in that it starts with a known initial state and only creates/expands new states that are reachable from already existing states through actions applicable in those existing states. Revisiting a state during execution (looping) does not occur because the state includes time.

Thus, the expansion of the state-action space forms a directed acyclic graph, and it is unnecessary to discount quality accumulation (i.e. discount reward as for value iteration) in order to guarantee convergence in policy evaluation and search. The forward expansion is an exploration of possible simulations of alternative contingency plans.

The algorithm iteratively

1. picks the most likely, best-so-far unexpanded (*frontier*) state in the current policy,
2. expands just one level of the state's joint actions and their outcome states while computing an admissible overestimate of eventual expected quality for each of the outcome states,
3. identifies the policy action from the state as that with the best expected quality outcomes, and
4. propagates the frontier state's better estimate back to preceding states, recursively choosing possible new policy actions along the way back to the initial state

until all policy actions are expanded.

Evaluating the next state to expand in step 1 is accomplished by a simple A* search from the initial state along the not fully explored best actions, choosing at each step to follow the outcome that (from the initial state) is most likely to occur until a frontier state is found. This search, in the worst case, explores the entire policy, which is $O(o^{am})$ for a agents, o outcomes per method, and m methods per agent since there are $O(o^a)$ outcome branches for a chosen policy joint action at each of $O(m)$ steps of a fully concurrent schedule.

The idea is to expand the best guess at the policy, so that states unique to suboptimal policies are never instantiated. The best case is $\Omega(oam)$ when there is no backtracking in the policy and no concurrent execution among the agents. Trading space for computation time, the reference to the most likely frontier state could be cached with each state along with the likelihood, and the values could be updated along with the policy actions in step 4, adding only a constant factor slowdown to 4 and a constant factor increase in memory. Caching the search in this way eliminates the A* search, reducing step 1 to a constant lookup, thus providing an exponential boost in computational speed. We have not implemented this since we have been focused on scalability, which has always been limited by memory.

Step 2 expands the state on the frontier picked in step 1, and gives an admissible overestimate of the expected quality for each of the actions' outcome states as they are instantiated. This provides a more accurate estimate of the quality of each action, which in step 3 can lead to picking a new policy action with a now higher estimated expected quality. In turn, the expanded state's estimate is the same as its policy action and so in step 4 the more accurate estimate is propagated to prior states to make their estimates more precise.

Step 4 is computed by walking back from the frontier toward the initial state along all action-outcome paths between, updating the stored quality estimate and policy action for each visited prior state. Multiple paths occur when different actions (possibly from different states) lead to equivalent outcome states that can be "merged" as a single instantiation to save memory. The implementation of these updates should not be done as a simple recursion:

```
function badUpdate( expandedState ) {
  updateExpectedQualityFromOutcomes( expandedState )
  for each priorState of expandedState
    badUpdate( priorState )
}
```

This can unnecessarily induce an exponentially growing number of updates for each earlier prior state since they can be updated for each time one of their outcome states is updated. Instead, each prior state should wait until all of its outcome states have been updated so that it is only updated once. Our implementation uses a priority queue of states to be updated, reverse sorted by time.

```
function goodUpdate( expandedState ) {
  queue.insert( expandedState )
  while ( queue not empty ) {
    state = queue.pop()
    updateExpectedQualityFromOutcomes( state )
    for each priorState of state
      queue.insertAfterLaterStates( priorState )
  }
}
```

If all outcome states are unique in the policy, step 4 only requires updating one prior state for each step from the initial state in it's unique schedule. For the worst case fully concurrent policy size $p = O(o^{am})$, step 4 requires a total of mp state updates for the generation of the policy.

It turns out that for any policy, at worst, each state expansion requires on average a number of state updates equal to the length of a schedule (policy graph) even if multiple paths from the initial state must be updated when merged outcome states are updated. Intuitively, if you take the worst case policy tree with no merges and merge two states at the same level, an update to the merged state requires updates to the same two chains of prior states that lead to the two separate states. Even though one of the outcome trees leading out of the merged states disappears, the number of updates for those states do not change. Thus, in effect no remaining state from the merge is updated any more or less than it was before, so the merge does not affect the number of updates for a state expansion, but only the number of states overall.

The AO* search algorithm terminates with the exact expected quality assigned to the initial state and with all policy actions proven to have greater or equal quality than other choices which must have occurred when all frontier states are terminal. States are deemed terminal when there are no remaining methods to execute. So, the algorithm keeps an anytime policy for the states it has explored and the policy is already computed upon completion of state generation. The main computational overhead for keeping a policy during expansion is really in selecting the next state to expand (for which we discussed a remedy) and in estimating the expected quality of the policy from each state as it is created.

A similar AO* algorithm, the informed unroller (IU), is applied to the decentralized, real-time C-TÆMS problem from a single agent's point of view (Musliner *et al.* 2006; Wu & Durfee 2007). Instead of updating the policy after each state expansion, IU "unrolls" the frontier of the best-so-far policy until the size grows by some pre-specified factor, choosing most reachable states first. Then, IU updates the policy based on estimated quality, determines the new

frontier of the policy, and repeats the unrolling. So, this approach differs most with step 1 of our algorithm. Our algorithm pays a higher cost in performing the search for the next best state to expand after each expansion because IU sorts the policy frontier $O(nlogn)$ for n frontier states (or n policy states) while we potentially explore the entire policy for each frontier state, $O(n^2)$. Our proposed remedy would eliminate this cost and require $O(logn)$ to update a heap for the next best candidate state to expand. The advantage of our approach over IU is that it will expand fewer total states to find the optimal solution (on average) since it is careful to only expand best-policy-so-far actions on each step rather than correcting after some number of steps have been taken, as IU does.

Instead of evaluating the most reachable state first, LAO* has been implemented to perform a depth-first search to expand the policy, irrespective of the likelihood of reaching the states, eliminating any overhead for prioritizing states. Hansen and Zilberstein state that this equal treatment of less likely states is an advantage for faster convergence in a comparison with RTDP (Barto, Bradtke, & Singh 1995) on a car racing problem (Hansen & Zilberstein 2001). In our future work, we hope to compare the depth-first expansion to the most-likely first heuristic for C-TÆMS problems. Otherwise, our algorithm differs from LAO* only in our customization to C-TÆMS and ignorance of looping (since there is none when including time as part of the state).

Minimizing State Creation Time

Our first implementation typically expanded about a million states in two minutes, exhausting 14GB of RAM, so we have since tried to be smarter about selectively exploring actions and states to avoid running out of memory. Here we describe how we streamlined the generation of actions and their outcome states that we believe greatly contributed to the speed of the implementation.

The basic idea is simple. We make a copy of the state being expanded and iteratively make small changes to it, enumerating all actions and all of their outcomes. A common way to generate combinations is to enumerate them in the same way counting is accomplished by incrementing the digits of a number. The least significant digit is always incremented through its values (0-9) and the other digits are only incremented when the next lesser significant digit "rolls over" from 9 back to 0. So, in order to generate the next state, we only need to "increment" the copy.

The variables describing the state are treated like digits. When starting to expand a state, we generate actions by enumerating/incrementing combinations of phase changes for starting and aborting methods. So, a "least significant method" always iterates like a binary digit between starting and not starting or aborting and not aborting. For each cycle through the pair of phase changes, the phase of the next incomplete method is incremented. Complete methods are skipped since there are no action choices.

Before incrementing to the next action combination, outcomes of the combined (joint) action are iterated as less significant variables. So, only when all outcomes are enumerated does the state "roll over" to the next joint action. So, both the actions and their outcomes are enumerated by incrementing the values of following variables (digits) ordered by decreasing significance corresponding to increasing detail:

- agent
  - method
    - phase
    - outcome (including NLEs)
      - duration
      - quality

In addition to the basic state, there is other derived state information that is preserved and reused for subsequent states. For example, the next possible start or abort time for a method does not need to be recomputed.

Avoiding Redundant Policy Space Exploration

Because the policy space (state-action graph) being explored grows so intractably, theoretically any method that can soundly eliminate instantiation of an action or state can potentially exponentially reduce the memory (and time) used to find the policy. Thus, for scalability, obvious suboptimal actions should be avoided. For example, never start a method

- for an agent that is already executing another,
- before its release time,
- after it can possibly meet its deadline,
- when disabled, or
- when not enabled.

Another way to conserve memory is to expand outcomes as a discrete event simulation instead of at every clock tick. We do this by computing the time of each outcome state as the minimum of possible method start times, abort times, and completion times. A challenge in the simulation of uncertain duration is that the probability distributions of duration and quality change over time. If a method does not terminate at one of its possible durations, then other durations become more likely, changing the relative likelihood of outcomes, to which quality distributions are tied. To minimize this computation during expansion, we precompute each method's distributions in a 2D array indexed by the duration that the method has been executing so far and the time left before the method's deadline.

A brute force technique for exploring the policy space would be to consider starting any combination of actions at every time point. However, if there are no deadlines, then there would be an infinite number of policies to explore (e.g., start method A at time t=1, start A at t=2, at t=3, ...). Exploring this infinite space is unnecessary since the expected quality of executing a method may not depend on when it is executed (so just explore starting A at time=1 if all other time points can give no better result). So the question then is how to minimize the start times tried without compromising optimality.

Selectively starting methods

Our basic rule of thumb to avoid redundant schedule exploration is that every agent must be executing a method unless all remaining methods are NLE targets, have not yet been released, or can no longer meet their deadlines.[2] The candidate times we consider for starting method are

- the release time,
- when the agent finishes executing another method,
- when it is enabled or facilitated (after the delay), and
- one time unit after it would be disable or hinder another.

For the last one, we cannot simply consider starting a task just before it would be disabled/hindered. Using this strategy, if the disabler's only opportunities to start would end the activity more than the NLE delay before the disabled activity's release time, then no schedules including both activities would ever be explored, compromising optimality.

Instead, the end of the source method must be timed to avoid disabling/hindering the target activity. This is especially difficult given that the search is a forward expansion/simulation in time, and the target's start time will be selected later. One option we considered was to wait until the target method started and then go back in time and repair the policy where necessary to include all options for timing the source method. However, this would be very complicated, especially when branches were deleted when pre-determined to be suboptimal.

Instead, we pre-compute the candidate start times for the source of a disable/hinder NLE as part of a temporal network of method start and end times. A depth first search generates all possible start times for the source based on all of its possible end times, which are based on all possible candidate start times of the NLE target, which may depend on start/end times of other methods. The resulting network records the chain(s) of justifications for the candidate time that must be reflected in the expanding state before choosing to start the NLE source. Note that the depth first search can itself be intractable, indicating in many cases that so is the overall problem.

Selectively aborting methods

The more obvious candidate abort times include the possible end times of the method. These would be the logical times to decide to abort in order for the agent to perform another method. The less obvious abort times are the possible completion times of other agents' methods that may be indirectly related. For example, suppose agents A and B each have a choice of two methods (because of deadline constraints), and A's method a2 enables B's method b2. If b1 is aborted to start b2, then a1 may need to be aborted to start a2 so that b2 can be enabled in time. So, a1's abort time is connected to b1's abort times. Beyond NLEs we have not determined when another agent's possible completion time could be reason to abort another agent's method, so we have to assume that any method could affect any other.

[2]We treat methods as inheriting NLEs, release times, and deadlines of their parent tasks.

Merging Equivalent States

Typically, MDP solutions only instantiate an outcome state once, tying all equivalent outcomes to the same state and only needing to instantiate the actions and outcomes of that state once. In our first implementation we did not bother doing this. Instead we just kept the duplicate outcome states. When later implementing the merge, we found that it slowed down problem solving by a factor of 2 or more and typically reduced the number of states expanded by no less than half. For problems that ran up against our memory limits, where only half could be solved, about 4 out of 5 of those could be solved more quickly without bothering to merge states.

This slowdown comes from having to store and lookup these states in a container.[3] One way we tried to improve merging was to loosen the definition of equality of states. A simple, effective tactic was to consider a method's state equivalent to another if neither would ever accumulate quality, that is if they failed (received a zero quality outcome or failed to meet its deadline), were aborted, or were abandoned.

A strategy that did not help was to ignore time when there were no more pending methods or the current time was ordered the same with respect to the release times of the pending methods. Because most of the computation time is spent in generating states and looking them up, the extra time for doing this simple check actually slows down performance by an order of magnitude and did not significantly reduce the number of states expanded.

Estimating Expected Quality

Part of the struggle of policy search is proving that one action is at least as good as all others. An A* search computes cost-so-far (function $g()$) and a heuristic overestimate of additional cost (function $h()$) for each search state, choosing to always explore the one with the least $f() = g() + h()$. A heuristic is better if the overestimate is smaller because it will expand no more search states than a greater (less accurate) overestimate. In addition, if one frontier state's upper bound $f()$ is less than another state's lower bound estimate $g()$, then not only is it not worth exploring, its memory can be reclaimed.

Since our optimal policy search is maximizing *expected* quality, instead of settling for bounds on eventual quality, we can more accurately use bounds on *expected* quality. Instead of settling for current quality as a lower bound, we can compute a more accurate lower bound on expected quality by including the expected quality of already executing methods that we know we will not abort. We have not yet tried to include quality from still pending methods. This is more difficult since is not enough to know just that the basic constraints of timing and enabling are guaranteed, but also that there is no opportunity cost to other methods whose timing or quality could be affected indirectly. Even then, it is necessary to avoid abandoning the method, so that quality

[3]We used a Standard Template Library map, which is implemented as a binary red-black tree. We intend to try other potentially quicker data structures such as a hash or trie, but haven't since our focus has been on the memory limit.

is not counted where it may not be accumulated, leading to improper policy action choices.

One of our bugs in computing expected values for bounding the AO* search was in computing quality for tasks through quality accumulation functions (QAFs). We were computing task expected quality bounds on expected quality bounds of subactivities. However, this is incorrect for min/max QAFs. For example, if a task is a max over two methods, each of which has quality distribution of (1.0, p=0.5; 2.0, p=0.5), the expected quality is 1.5 for each method, but the task's expected quality is not simply the max of 1.5 and 1.5. The task's quality is 2.0 for 3 of the 4 outcome combinations and 1.0 in the one case where both methods have qualities of 1.0, so the expected quality of the task is actually 1.75.

So, in order to compute bounds for the root task for the overall state, the QAFs must, in general, be evaluated on the quality distributions of its children. Calculating QAFs over distributions has slowed performance by a factor of 2, but the better expected quality estimates cut the number of states being expanded roughly in half.

Joint Action Explosion

One technique that did not help improve performance involved partially expanding a subset of possible joint actions when there were too many to all fit in memory. The idea was that even if the set of joint actions were intractable, the optimal policy may not, and we might at least get good bounds on the optimal expected quality. This expansion is achieved by only expanding actions of one agent from a state and not expanding outcomes until the last agent's actions are expanded. Thus, some combinations of actions can be explored without losing track of which ones had not yet been explored.

While some expansion was possible (where before it was not), the problems of this larger size that we tried were not solvable, and we have been unable to get useful bounds on expected quality for them (e.g. quality [1.0, 100.0]). The partial expansion was not helpful for the smaller problems either, roughly doubling the time to solve them.

Conclusion

We have described our experience in terms of challenges and strategies in trying to scale centralized, multiagent optimal policy search for C-TÆMS problems with uncertainty in action quality and duration. Partly due to an efficient state expansion, the search exhausted memory quickly, so search speed rarely inhibited scalability. Thus, our main focus has been to minimize the number of states expanded. We found that it is difficult to avoid redundant and obviously suboptimal policy exploration, mentioning some strategies that we have not tried but think may be fruitful. Many of the techniques we tried to minimize exploration of the state space had limited or no success.

Here's a summary of techniques that did and did not help in scaling to larger problems.

Techniques that helped scaling:

- efficient enumeration/creation of actions and states,
- selective start and abort times,
- more precise expected quality estimates (trading time for space), and
- instantiating duplicates of equivalent state to avoid the overhead of a lookup container.

Techniques that did not help scaling:

- merging equivalent outcome states to avoid expanding duplicates (same as last bullet above),
- using more inclusive equivalence definitions, and
- partially expanding actions to avoid the intractability of joint actions.

Although the literature on general MDP research is useful, we find that the structure provided by a modeling language can be exploited in many ways, of which we have only been able to scratch the surface. In the future, we hope to make better connections between the problems and strategies found with C-TÆMS and those of other STRIPS based modeling languages.

Acknowledgments

Alan Garvey greatly contributed to our research and development by providing a large number and variety of problem sets and finding boundary problems that we could and could not solve. The research described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with DARPA.

References

Barto, A. G.; Bradtke, S. J.; and Singh, S. P. 1995. Learning to act using real-time dynamic programming. *Artificial Intelligence* 72(1-2):81–138.

Boddy, M.; Horling, B.; Phelps, J.; Goldman, R.; Vincent, R.; Long, A.; Kohout, R.; and Maheswaran, R. 2005. C_TAEMS language specification. available from authors.

Decker, K., and Lesser, V. R. 1993. Quantitative Modeling of Complex Environments. *International Journal of Intelligent Systems in Accounting, Finance and Management. Special Issue on Mathematical and Computational Models and Characteristics of Agent Behaviour.* 2:215–234.

Erol, K.; Hendler, J.; and Nau, D. 1994. Semantics for hierarchical task-network planning. Technical Report CS-TR-3239, University of Maryland.

Fikes, R., and Nilsson, N. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2:189–208.

Ginsberg, M. L., ed. 1987. *Readings in nonmonotonic reasoning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

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

Maheswaran, R. T.; Szekely, P.; Becker, M.; Fitzpatrick, S.; Gati, G.; Jin, J.; Neches, R.; Noori, N.; Rogers, C.;

Sanchez, R.; Smyth, K.; and VanBuskirk, C. 2008. Predictability & criticality metrics for coordination in complex environments. In *7th International Joint Conference on Autonomous Agents and Multiagent Systems*.

Mausam, and Weld, D. S. 2005. Concurrent probabilistic temporal planning. 120–129.

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*.

Smith, S.; Gallagher, A.; Zimmerman, T.; Barbulescu, L.; ; and Rubinstein, Z. 2007. Distributed management of flexible times schedules. In *Proceedings of the International Joint Conference on Autonomous Agents and MultiAgent Systems*, 1–8. New York, NY, USA: ACM.

van Hoeve, W. J.; Gomes, C. P.; Selman, B.; and Lombardi, M. 2007. Optimal multi-agent scheduling with constraint programming. In *Proceedings of the Innovative Applications of Artificial Intelligence*. AAAI Press.

Wagner, T. A.; Raja, A.; and Lesser, V. R. 2006. Modeling uncertainty and its implications to sophisticated control in TÆMS agents. *Journal of Autonomous Agents and Multi-Agent Systems* 13(3):235–292.

Wu, J., and Durfee, E. H. 2007. Solving large TÆMS problems efficiently by selective exploration and decomposition. In *Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems*, 1–8. New York, NY, USA: ACM.