Distributed Management of Flexible Times Schedules

Stephen F. Smith, Anthony Gallagher, Terry Zimmerman,
Laura Barbulescu, Zachary Rubinstein
The Robotics Institute, Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh PA 15024
{sfs,anthonyg,wizim,laurabar,zbr}@cs.cmu.edu

ABSTRACT

In this paper we consider the problem of managing and exploiting schedules in an uncertain and distributed environment. We assume a team of collaborative agents, each responsible for executing a portion of a globally pre-established schedule, but none possessing a global view of either the problem or solution. Each individual agent is aware of dependencies between its ~~scheduled~~ actions and those of other agents (providing a basis for online coordination), and each agent is also provided with a set of local contingency (fallback) options. The goal is to maximize the joint quality obtained from the actions executed by all agents, given that unexpected events will force changes to some prescribed actions and reduce the utility of ~~executing~~ others as execution unfolds. We describe an agent architecture for solving this problem that couples two basic mechanisms: (1) a "flexible times" representation of the agent's schedule (using a Simple Temporal Network (STN)), which hedges against temporal uncertainty by promoting execution from a set of feasible solutions, and (2) an incremental rescheduling procedure, which acts to revise the agent's schedule when execution is forced outside of this set of solutions or when execution events reduce the expected value of this feasible solution set. Two layers of coordination augment this core local problem-solving infrastructure. Basic coordination with other agents is achieved simply by communicating schedule changes to those agents with inter-dependent actions. Then, as time permits, the STN is used to drive an inter-agent option generation and query process, aimed at identifying opportunities for solution improvement through joint change. Using a simulator to model the uncertain execution environment, we compare the performance of our multi-agent system with an expected optimal (but non-scalable) centralized MDP solver ~~over a range of problem instances.~~

Keywords

Cooperative Distributed Problem Solving, Agent Architectures, Multi-Agent Planning,

1. INTRODUCTION

The practical constraints of many application environments require distributed management of executing plans and schedules. Such factors as geographical separation of executing agents, limitations on communication bandwidth, constraints relating to chain of command and the high tempo of execution dynamics may all preclude any single agent from obtaining a complete global view of the problem, and hence necessitate collaborative yet localized planning and scheduling decisions. In this paper, we consider the problem of managing and executing schedules in an uncertain and distributed environment as defined by the DARPA Coordinators program. We assume a team of collaborative agents, each responsible for executing a portion of a globally pre-established schedule, but none possessing a global view of either the problem or solution. The team goal is to maximize the total quality of all actions executed by all agents, given that unexpected events will force changes to pre-scheduled actions and ~~reduce~~ the utility of executing others as execution unfolds. ~~To provide a basis for distributed coordination, each agent is aware of dependencies between its scheduled actions and those of other agents, and each agent is also provided with a pre-computed set of local contingency (fallback) options.~~

We take an incremental flexible-times scheduling framework as ~~our starting point for specifying a multi-agent architecture for solving this problem.~~ In a flexible-times representation of an agent's schedule, the execution intervals associated with scheduled actions are not fixed, but instead are allowed to float within imposed time and action sequencing constraints. ~~A flexible times representation provides the basic advantage that it allows the explicit~~ use of slack as a hedge against simple forms of executional uncertainty (e.g., action durations), and its underlying implementation as a Simple Temporal Network (STN) model provides efficient updating and consistency enforcement mechanisms. The advantages of flexible times frameworks have ~~been demonstrated by previous work~~ in various centralized planning and scheduling contexts (e.g., [11, 7, 8, 9, 10]). However their use in distributed problem solving settings has been quite sparse ([6] is one exception), and prior approaches to multi-agent scheduling (e.g., [5, 12] have generally operated with fixed-times representations of agent schedules.

We define a local agent infra-structure centered around incremental management of a flexible times schedule, and use the STN-based representation to loosen the coupling be-

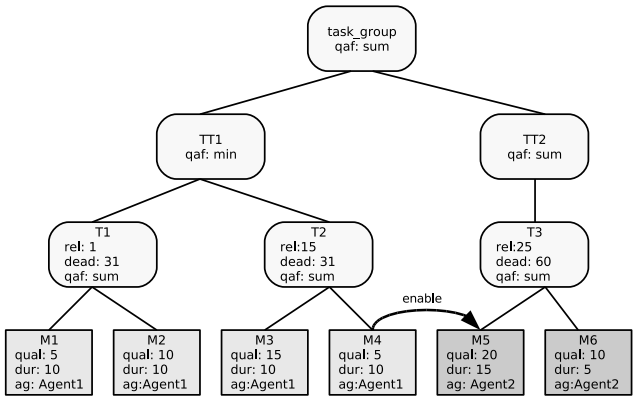

Figure 1: A two agent C_TAEMS problem.

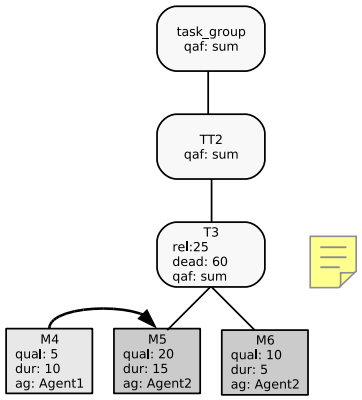

Figure 2: Subjective view for Agent 2.

tween executor and scheduler threads, ~~to retain a basic ability to absorb unexpected executional delays (or speedups), and to provide a basic criterion for detecting the need for schedule change.~~ Local change is accomplished by an incremental scheduler, designed to maximize quality while attempting to minimize schedule change. To this schedule management infra-structure, we add two mechanisms for multi-agent coordination. Basic coordination with other agents is achieved by simple communication of local schedule changes to other agents with inter-dependent actions. Layered over this is a non-local option generation and evaluation process, aimed at identification of opportunities for global improvement through joint changes to the schedules of multiple agents. This latter process uses analysis of detected conflicts in the STN as a basis for generating options.

The remainder of the paper is organized as follows. We begin by briefly summarizing the general distributed scheduling problem of interest in our work. Next, we introduce the agent architecture we have developed to solve this problem and sketch its operation. In the following sections, we describe the components of the architecture in more detail, considering in turn issues relating to executing agent schedules, incrementally revising ~~agent~~ schedules and coordinating schedule changes among multiple agents. We then give some experimental results to indicate current system performance. Finally we conclude with a brief discussion of current research plans.

2. THE COORDINATORS PROBLEM

As indicated above the distributed schedule management problem that we address in this paper is that put forth by the DARPA Coordinators program. The Coordinators problem is concerned generally with the collaborative execution of a joint mission by a team of agents in a highly dynamic environment. A mission is formulated as a network of tasks, which are distributed among the agents by the MASS simulator such that no agent has a complete, "objective" view of the whole problem. Instead, each agent receives only a "subjective view" containing just the tasks for which it is responsible and the remote tasks that have interdependencies with these local tasks. A pre-computed initial schedule is also distributed to the agents, and each agent's schedule indicates which of its local ~~tasks~~ should be executed and when.

Each ~~task~~ has an associated quality value which accrues if it is successfully executed within its constraints, and the overall goal is to maximize the quality obtained during execution. As execution proceeds, agents must react to unexpected results generated by the simulator (e.g., ~~task~~ delays, failures) as well as changes to the mission (e.g., new ~~tasks~~, deadline changes), recognize when scheduled tasks are no longer feasible or desirable, ~~and coordinate~~ with each other to take corrective, quality-maximizing rescheduling actions that keep execution of the overall mission moving forward.

Problems are formally specified using a version of the *TAEMS* language (Task Analysis, Environment Modeling and Simulation) [4] called *C_TAEMS* [1]. Within *C_TAEMS*, ~~tasks~~ are represented hierarchically, as shown in the example in Figure 1. At the highest, most abstract level, the root of the tree is a special task called the task group. On successive levels, tasks constitute aggregate activities, which can be decomposed into sets of subtasks and/or primitive actions, termed "methods." Methods appear at the leaf level of *C_TAEMS* task structures and are those that are directly executable in the world. Each declared method m can only be executed by a specified agent (denoted by $ag : AgentN$ in Figure 1) and each agent can be executing at most one method at any given time (i.e. agents are unit-capacity resources). Method durations and quality are typically specified as discrete probability distributions, and hence known with certainty only after they have been executed.[1] It is also possible for a method to fail unexpectedly in execution, in which case the reported quality is zero.

For each task, a quality accumulation function *qaf* is defined, which specifies when and how a task accumulates quality as its subtasks (methods) are executed. For example, a task with a *min* qaf will accrue the quality of its child with lowest quality *if* all its children execute and accumulate positive quality. Tasks with *sum* or *max* qafs acquire quality as soon as one child executes with positive quality; as their qaf names suggest, their respective values ultimately will be the total or maximum quality of all children that executed ~~respectively~~. A *sync-sum* task will accrue quality only for those children that commence execution concurrently with

[1]For simplicity, Figures 1 and 2 show only fixed values for method quality and duration.

the first child that executes, while an *exactly-one* task accrues quality only if precisely one of its children executes.

Inter-dependencies between tasks/methods in the problem are modeled via non-local effects (*nles*). Two types of nles can be specified: "hard" and "soft." Hard nles express causal preconditions: for example, the *enables* nle in Figure 1 stipulates that the target method $M5$ can not be executed until the source $M4$ accumulates quality. Soft nles, which include *facilitates* and *hinders*, are not required constraints; however, when they are in play, they amplify (or dampen) the quality and duration of the target task.

Any given task or method a can also be constrained by an earliest start time and a deadline, specifying the window in which a can be feasibly executed. a may also inherit these constraints from ancestor tasks at any higher level in the task structure, and its effective execution window will be defined by the tightest of these constraints.

Figure 1 shows the complete "objective" view of a simple 2 agent problem. Figure 2 shows the subjective view available to agent 2 for the same problem. In what follows, we will sometimes use the term activity to refer generically to both task and method nodes.

3. OVERVIEW OF APPROACH

Our solution framework combines two basic principles for coping with the problem of managing multi-agent schedules in an uncertain and time stressed execution environment. First is the use of a STN-based flexible times representation of solution constraints, which allows execution to be driven ~~by from~~ a "set" of schedules rather than a single point solution. This provides a basic hedge against temporal uncertainty and can be used to modulate the need for solution revision. The second principle is to first respond locally to exceptional events, and then, as time permits, explore non-local options (i.e., options involving change by 2 or more agents) for global solution improvement. This provides a ~~basic~~ means for keeping pace with execution, and for tying the amount of effort spent in more global multi-agent solution improvement to the time available. Both local and non-local problem solving time is further minimized by the use of a core incremental scheduling procedure.

Our solution framework is made concrete in the agent architecture depicted in Figure 3. In its most basic form, an agent comprises four principal components - an Executor, a Scheduler, a Distributed State Manager (DSM), and an Options Manager - all of which share a common model of the current problem and solution state that couples a domain-level representation of the subjective c_taems task structure to an underlying STN. At any point during operation, the currently installed schedule dictates the timing and sequence of domain-level ~~actions~~ that will be initiated by the agent. The Executor, running in its own thread, continually monitors the enabling conditions of various pending activities, and activates the next pending activity as soon as all of its causal and temporal constraints are satisfied.

When execution results are received back from the environment and/or changes to assumed external constraints are received from other agents, the agent's model of current state

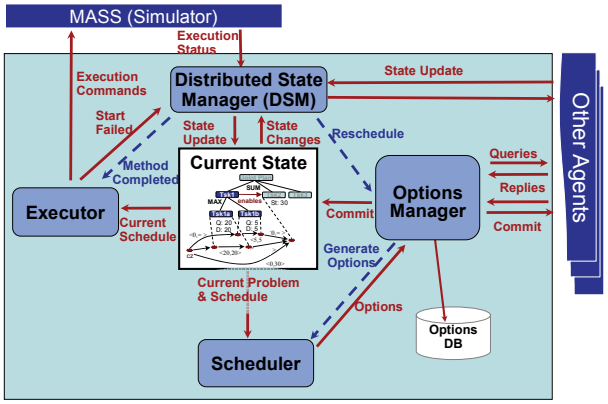

Figure 3: Agent Architecture.

is updated. In cases where this update leads to inconsistency in the STN or it is otherwise recognized that the current local schedule might now be improved, the Scheduler, running on a separate thread, ~~will be~~ invoked to revise the current solution and install a new schedule. Whenever local schedule constraints change either in response to a current state update or through manipulation by the Scheduler, the DSM is invoked to communicate these changes to interested agents (i.e., those agents that share dependencies and have overlapping subjective views).

After responding locally to a given state update and communicating consequences, the agent will use any remaining computation time to explore possibilities for improvement through joint change. The Option Manager utilizes the Scheduler (in this case in hypothetical mode) to generate one or more non-local options, i.e., identifying changes to the schedule of one or more other agents that will enable the local agent to raise the quality of its schedule. These options are formulated and communicated as queries to the appropriate remote agents, who in turn hypothetically evaluate the impact of proposed changes from their local perspective. In those cases where global improvement is verified, joint changes are committed to.

In the following sections we consider the mechanics of these components in more detail.

4. THE SCHEDULER

As indicated above, our agent scheduler operates incrementally. Incremental scheduling frameworks are ideally suited for domains requiring tight scheduler-execution coupling: rather than recomputing a new schedule in response to every change, they respond quickly to execution events by localizing changes and making adjustments to the current schedule to accommodate the event. ~~Schedule~~ stability is maintained, providing better support for the continuity in execution. This latter property is also advantageous in multi-agent settings, since solution stability tends to minimize the ripple across different agents' schedules.

The coupling of incremental scheduling with flexible times scheduling adds additional leverage in an uncertain, multi-

agent execution environment. As mentioned earlier, slack can be used as a hedge against uncertain method execution times. It also provides a basis for softening the impact of inter-dependencies across agents.

In this section, we summarize the core scheduler that we have developed to solve the Coordinators problem. In subsequent sections we discuss its use in managing execution and coordinating with other agents.

4.1 STN Solution Representation

To maintain the range of admissible values for the start and end times of various methods in a given agent's schedule, all problem and scheduling constraints impacting these times are encoded in an underlying Simple Temporal Network (STN)[3]. An STN represents temporal constraints as a graph $G < N, E >$, where nodes in N represent the set of time points of interest, and edges in E are distances between pairs of time points in N. A special time point, called *calendar zero* grounds the network and has the value 0. Constraints on activities (e.g. release time, due time, duration) and relationships between activities (e.g. parent-child relation, enables) are uniformly represented as temporal constraints (i.e., edges) between relevant start and finish time points. An agent's schedule is designated as a total ordering of selected methods by posting precedence constraints between the end and start points of each ordered pair. As new methods are inserted into a schedule or external state updates require adjustments to existing constraints (e.g., substitution of an actual duration constraint, tightening of a deadline), the network propagates constraints and maintains lower and upper bounds on all time points in the network. This is accomplished efficiently via the use of a standard all-pairs shortest path algorithm; in our implementation, we take advantage of an incremental procedure based on [2]. As bounds are updated, a consistency check is made for the presence of *negative cycles*, and the absence of any such cycle ensures the continued temporal feasibility of the network (and hence the schedule). Otherwise a *conflict* has been detected, and some amount of constraint retraction is necessary to restore feasibility.

4.2 Maintaining High-Quality Schedules

The scheduler consists of two basic components: a *quality propagator* and an *activity allocator* that work in a tightly integrated loop. The quality propagator analyzes the activity hierarchy and collects a set of methods that (if scheduled) would maximize the quality of the agent's local problem. The methods are collected without regard for resource contention; in essence, the quality propagator optimally solves a relaxed problem where agents are capable of performing an infinite number of activities at once. The allocator selects methods from this list and attempts to install them in the agent's schedule. Failure to do so reinvokes the quality propagator with the problematic activity excluded.

The Quality Propagator - The quality propagator performs the following actions on the *C_TAEMS* task structure:

- Computes the quality of all activities in the task structure: The quality $qual(m)$ of a method m is computed from the probability distribution of the execution outcomes. The quality $qual(t)$ of a task t is computed by applying its *qaf* to the assessed quality of its children.
- Generates a list of *contributors* for each task: methods that, if scheduled, will maximize the quality obtained by the task.
- Generates a list of *activators* for each task: methods that, if scheduled, are sufficient to qualify the task as scheduled. Methods in the activators list are chosen to minimize demands on the agent's timeline without regard to quality.

The first time the quality propagator is invoked, the qualities of all tasks and methods are calculated and the initial lists of contributors and activators are determined. Subsequent calls to the propagator occur as the allocator installs methods on the agent's timeline: failure of the allocator to install a method causes the propagator to recompute a new list of contributors and activators.

The Activity Allocator - The activity allocator seeks to install the *contributors* of the taskgroup identified by the quality propagator onto the agent's timeline. Any currently scheduled methods that do *not* appear in the contributors list are first unscheduled and removed from the timeline. The *contributors* are then preprocessed using a quality-centric heuristic to create an agenda sorted in decreasing quality order. In addition, methods associated with ~~a~~ "and" task (i.e., *min, sumand*) are grouped consecutively within the agenda. Since an "and" task accumulates quality only if all its children are scheduled, this biases the scheduling process towards failing early (and regenerating contributors) when the methods chosen for the ~~*min*~~ cannot together be allocated.

The allocator iteratively ~~removes~~ the first method m_{new} from the agenda and attempts to install it. This entails first checking that all activities that enable m_{new} have been scheduled, while attempting to install any enabler that is not. If any of the enabler activities fails to install, the allocation pass fails. When successful, the *enables* constraints linking the enabler activities to m_{new} are activated. The STN rejects an infeasible *enabler* constraint by returning a *conflict*. In this event any enabler activities it has scheduled are uninstalled and the allocator returns failure. Once scheduling of enablers is ensured, a feasible slot on the agent's timeline within m_{new}'s time window is sought and the allocator attempts to insert m_{new} between two currently scheduled methods. At the STN level, m_{new}'s insertion breaks the sequencing constraint between the two extant timeline methods and attempts to insert two new sequencing constraints that chain m_{new} to these methods. If these insertions succeed, the routine returns success, otherwise the two extant timeline methods are relinked and allocation attempts the next possible slot for m_{new} insertion.

5. THE DYNAMICS OF EXECUTION

Maintaining a *flexible-times* schedule enables us to use a *conflict-driven* approach to schedule repair: Rather than reacting to every event in the execution that may impact the existing schedule by computing an updated solution, the STN can absorb any change that does not cause a *conflict*.

Consequently, computation (producing a new schedule) and communication costs (informing other agents of changes that affect them) are minimized.

One basic mechanism needed to model execution in the STN is a dynamic model for current time. We employ a model proposed by [6] that establishes a 'current-time' time point and includes a link between it and the calendar-zero time point. As each method is scheduled, a simple precedence constraint between the current-time time point and the method is established. When the scheduler receives a current time update, the link between calendar-zero and current-time is modifed to reflect this new time, and the constraint propagates to all scheduled methods.

A second ~~basic~~ issue concerns synchronization between the executor and the scheduler, as producer and consumer of the schedule running on different threads within a given agent. This coordination must be robust despite the fact that the executor needs to start methods for execution in real-time even while the scheduler may be reassessing the schedule to maximize quality, and/or transmitting a revised schedule. If the executor, for example, slates a method for execution based on current time while the scheduler is instantiating a revised schedule in which that method is no longer next-to-be-executed, an inconsistent state may arise within the agent architecture. This is addressed in part by introducing a "freeze window;" a ~~short specified~~ (and adjustable) time period beyond current time within which any activity slated as eligible to start in the current schedule cannot be rescheduled by the scheduler.

The scheduler is triggered in response to various environmental messages. There are two types of environmental message classes that we discuss here as "execution dynamics:" 1) feedback as a result of method execution - both the agent's own and that of other agents, and 2) changes in the C_TAEMS model corresponding to a set of simulator-directed evolutions of the problem and environment. Such messages are termed *updates* and are treated by the scheduler as directives to permanently modify parameters in its model. We discuss these update types in turn here and defer until later the discussion of *queries* to the scheduler, a 'what-if' mode initiated by a remote agent that is pursuing higher global quality.

Whether it is invoked via an update or a query, the scheduler's response is an *option*; essentially a complete schedule of activities the agent can execute along with associated quality metrics. We define a *local option* as a valid schedule for an agent's activities, which does not require change to any other agent's schedule. The overarching design for handling execution dynamics aims at anytime scheduling behavior in which a local option maximizing the local view of quality is returned quickly, possibly followed by globally higher quality schedules that entail inter-agent coordination if available scheduler cycles permit. As such, the default scheduling mode for updates is to seek the highest quality local option according to the scheduler's search strategy, instantiate the option as its current schedule, and notify the executor of the revision.

5.1 Responding to Activity Execution

As suggested earlier, a committed schedule consists of a sequence of methods, each with a designated $[est, lst]$ start time window (as provided by the underlying STN representation). The executor is free to execute a method any time within its start time window, once any additional enabling conditions have been confirmed. These scheduled start time windows are established using the expected duration of each scheduled method (derived from associated method duration distributions during schedule construction). Of course as execution unfolds, actual method durations may deviate from these expectations. In these cases, the flexibility retained in the schedule can be used to absorb some of this unpredictability and modulate invocation of a schedule revision process.

Consider the case of a *method completion* message, one of the environmental messages that could be communicated to the scheduler as an execution state update. If the completion time is coincident with the expected duration (i.e., it completes exactly as expected), then the scheduler's response is to simply mark it as 'completed' and the agent can proceed to communicate the time at which it has accumulated quality to any remote agents linked to this method.

However if the method completes with a duration shorter than expected a rescheduling action might be warranted. The posting of the actual duration in the STN introduces no potential for conflict in this case, either with the lsts of local or remote methods that depend on this method as an enabler, or to successively scheduled methods on the agent's timeline. However, it may present a possibility for exploiting the unanticipated scheduling slack. The flexible times representation afforded by the STN provides a quick means of assessing whether the next method on the timeline can begin immediate execution instead of waiting for its previously established *est* start time. If indeed ~~*est*~~ of the next scheduled method can "spring back" to *current-time* once the actual duration constraint is substituted for the expected duration constraint, then the schedule can be left intact and simply communicated back to the executor. If alternatively, other problem constraints prevent this relaxation of the *est*, then there is forced idle time that may be exploited by revising the schedule, and the scheduler is invoked (always respecting the freeze period).

If the method completes later than expected, then there is no need for rescheduling under flexible times scheduling unless 1) the method finishes later than the latest start time (lst) of the subsequent scheduled activity, or 2) it finishes later than its deadline. Thus we only invoke the scheduler if, upon posting the late finish in the STN, a constraint violation occurs. In the latter case no quality is accrued and rescheduling is mandated even if there are no conflicts with subsequent scheduled activities.

Other execution status updates the agent may receive include:

- *method start* - If a method sent for execution is started within its $[est, lst]$ window, the response is to mark it as 'executing'. A method cannot start earlier than when it is transmitted by the executor but it is possible for it to start later than requested. If the posted

start time causes an inconsistency in the STN (e.g. because the expected method duration can no longer be accommodated) the duration constriant in the STN is shortened based on the known distribution until either consistency is restored or rescheduling is mandated.

- *method failure* - Any method under execution may fail unexpectedly, garnering no quality for the agent. At this point rescheduling is mandated as the method may enable other activities or significantly impact quality in the absence of local repair. Again, the executor will proceed with execution of the next method if its start time arrives before the revised schedule is committed, and the scheduler accommodates this by respecting the freeze window.

- *current time advances* An update on 'current time' may arrive either alone or as part of any of the previously discussed updates. If, when updating the current-time link in the STN (as described above), a conflict results, the execution state is inconsistent with the schedule. In this case, the scheduler proceeds as if execution were consistent with its expectations, subject to possible later updates.

5.2 Responding to Model Updates

The agent can also dynamically receive changes to the agent's underlying C_TAEMS model. Dynamic revisions in the outcome distributions for methods already in an agent's subjective view may impact the assessed quality and/or duration values that shaped the current schedule. Similarly, dynamic revisions in the designated release times and deadlines for methods and tasks already in an agent's subjective view can invalidate an extant schedule or present opportunities to boost quality. It is also possible during execution to receive updates in which new methods and possibly entire task structures are given to the agent for inclusion in its subjective view. Model changes that involve temporal constraints are handled in much the same fashion as described for method starts and completions, i.e, rescheduling is required only when the posting of the revised constraints leads to an STN conflict. In the case of non-temporal model changes, ~~alternatively,~~ rescheduling action is currently always initiated.

6. COORDINATING WITH OTHER AGENTS

Having responded locally to an unexpected execution result or model change, it is necessary to communicate the consequences to agents with inter-dependent actions so that they can align their decisions accordingly. Responses that look good locally may have a sub-optimal global effect once alignments are made, and hence agents must have the ability to seek mutually beneficial joint schedule changes. In this section we summarize the coordination mechanisms provided in the agent architecture to address these issues.

6.1 Communicating Non-Local Constraints

A basic means of coordination with other agents is provided by the Distributed State Mechanism (DSM), which is responsible for communicating changes made to the model or schedule of a given agent to other "interested" agents. More specifically, ~~the DSM of a given agent~~ acts to push any changes made to time bounds, quality, or status of a local task/method to all the other agents that have that same task/method as a remote node in their subjective views. A recipient agent treats any communicated changes as ~~additional forms of updates, in this case, modifying~~ the current constraints associated with non-local (but inter-dependent) tasks or methods. These changes are handled identically to updates reflecting schedule execution results, potentially triggering the local scheduler if the need to reschedule is detected.

6.2 Generating Non-Local Options

As mentioned in the previous section, the agent's first response to any given query or update (either from execution or from another agent) is to generate one or more local options. Such options represent local schedule changes that are consistent with all currently known constraints originating from other agents' schedules, and hence can be implemented without interaction with other agents. It many cases, however, a larger-scoped change to the schedules of two or more agents can produce a higher-quality response.

Exploration of opportunities for such coordinated action by two or more agents is the responsibility of the Options Manager. Running in lower priority mode than the Executor and Scheduler, the Options Manager initiates a non-local option generation and evaluation process in response to any local schedule change made by the agent if computation time constraints permits. Generally speaking, a non-local option identifies certain relaxations (to one or more constraints imposed by methods that are scheduled by one or more remote agents) that enable the generation of a higher quality local schedule. When found, a non-local option is used by a coordinating agent to formulate queries to any other involved agents in order to determine the impact of such constraint relaxations on their local schedules. If the combined quality change reported back from a set of one or more relevant queries is a net gain, then the issuing agent signals to the other involved agents to commit to this joint set of schedule changes. The Option Manager currently employs two basic search strategies for generating non-local options, each exploiting the local scheduler in hypothetical mode.

Optimistic Synchronization - Optimistic synchronization is a non-local option generation strategy where search is used to explore the impact on quality if optimistic assumptions ~~about~~ currently unscheduled remote enablers ~~are made.~~ More specifically, the strategy looks for "would be" contributor methods that are currently unscheduled due to the fact that one or more remote enabling (source) tasks/methods are not currently scheduled. For each such local method, the set of remote enablers are hypothetically activated, and the scheduler attempts to construct a new local schedule under these optimistic assumptions. If successful, a non-local option is generated, specifying the value of the new, higher quality local schedule, the temporal constraints on the local target activity, and the set of *must-schedule* enabler activities that must be scheduled by remote agents in order to achieve this local quality. The needed queries requesting the quality impact of scheduling these activities are then formulated and sent to the relevant remote agents.

To illustrate, consider again the example in Figure 2. The

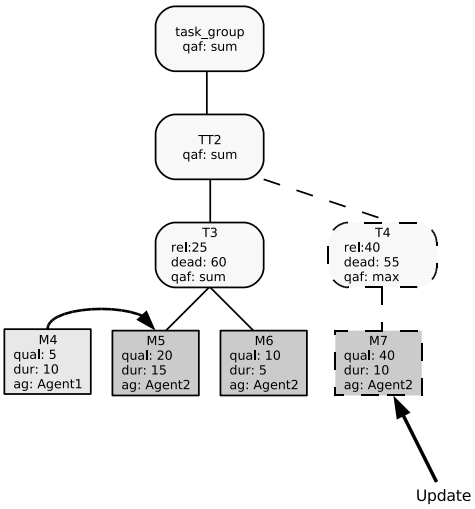

Figure 4: A high quality task is added to the task structure of Agent2.

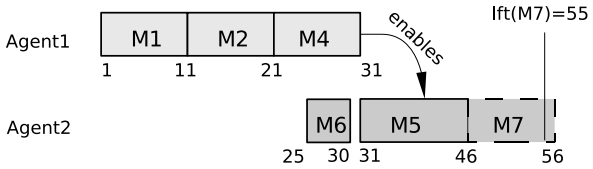

Figure 5: If $M4$, $M5$ and $M7$ are scheduled, a conflict is detected by the STN.

maximum quality that Agent1 can contribute to the task group is 15 (by scheduling $M1$, $M2$ and $M3$). Assume that this is Agent1's current schedule. Given this state, the maximum quality that Agent2 can contribute to the task group is 10, and the total taskgroup quality would then be $15 + 10 = 25$. Using optimistic synchronization, Agent2 will generate a non-local option that indicates that if $M5$ becomes enabled, both $M5$ and $M6$ would be scheduled, and the quality contributed by Agent2 to the task group would become 30. Agent2 sends a *must schedule M4* query to Agent1. Because of the time window constraints, Agent1 must remove $M3$ from its schedule to get $M4$ on, resulting in a new lower quality schedule of 5. However, when Agent2 receives this option response from Agent1, it determines that the total quality accumulated for the task group would be $5 + 30 = 35$, a net gain of 10. Hence, Agent 2 signals to Agent1 to commit to this non-local option.

Conflict-Driven Relaxation - A second strategy for generating non-local options, referred to as Conflict-Directed Relaxation, utilizes analysis of STN conflicts to identify and prioritize external constraints to relax in the event that a particular method that would increase local quality is found to be unschedulable. Recall that if a method cannot be feasibly inserted into the schedule, an attempt to do so will generate a negative cycle. Given this cycle, the mechanism proceeds in three steps. First, the constraints involved in the cycle are collected. Second, by virtue of the connections in the STN to the domain-level C_TAEMS model, this set is filtered to identify the subset associated with remote nodes. Third, constraints in this subset are selectively retracted to determine if STN consistency is restored. If successful, a non-local option is generated indicating which remote constraint(s) must be relaxed and by how much to allow installation of the new, higher quality local schedule.

To illustrate this strategy, consider ~~Figure 5~~ where Agent1 has $M1$, $M2$ and $M4$ on its timeline, and therefore $est(M4) = 21$. Agent2 has $M5$ and $M6$ on its timeline, with $est(M5) = 31$ ($M6$ could be scheduled before or after $M5$). Suppose that Agent2 receives a new task $M7$ with deadline 55 (see Figure 4). If Agent2 could schedule $M7$, the ~~quality contributed by Agent2 to the task group~~ would be 70. However, an attempt to schedule $M7$ together with $M5$ and $M6$ leads to a conflict, since the $est(M7) = 46$, $dur(M7) = 10$ and $lft(M7) = 55$ (see Figure 5). Conflict-directed relaxation by Agent 2 suggests relaxing the $lft(M4)$ by 1 tick to 30, and this query is communicated to Agent 1. In fact, by retracting either method $M1$ or $M2$ from the schedule this relaxation can be accommodated with no quality loss to Agent1 (due to the *min* qaf). Upon communication of this fact Agent 2 signals to commit.

7. EXPERIMENTAL RESULTS

An initial version of the agent described in this paper was developed in collaboration with XYZ Corporation and subjected to the independently conducted Coordinators programmatic evaluation. This evaluation involved over 2000 problem instances randomly generated by a scenario generator that was configured to produce scenarios of varying durations within six experiment classes. These classes, summarized in Table 1, were designed to evaluate key aspects of a set of Coordinators distributed scheduling agents, such as their ability to handle unexpected execution results, chains of nle's involving multiple agents, and effective scheduling of new activities that arise unexpectedly at some point during the problem run. Year 1 evaluation problems were constrained to be small enough (3 -10 agents, 50 - 100 methods) such that comparison against an optimal centralized solver was feasible. The evaluation team employed an MDP-based solver capable of unrolling the entire search space for these problems, choosing for an agent at each execution decision point the activity most likely to produce maximum global quality. This established a challenging benchmark for the distributed agent systems to compare against. The hardware configuration used by the evaluators instantiated and ran one agent per machine, dedicating a separate machine to the MASS simulator.

As reported in Table 1 the year 1 prototype agent clearly compares favorably to ~~benchmark~~ on all classes, coming within 2% of the MDP optimal averaged over the entire set of 2190 problems. These results are particularly notable given that each agent's STN-based scheduler does very little reasoning over the success probability of the activity sequences it selects to execute. Only simple tactics were adopted to *explicitly* address such uncertainty, such as the use of expected durations and quality for activities and a policy of excluding from consideration those activities with failure likelihood of >75%. The very respectable agent performance can be at least partially credited to the fact that

Problem Class	Class Description	Agent Quality
OD (390 probs)	'Only Dynamics'. No NLEs. Actual task duration & quality vary according to distribution.	0.979
INT (360 probs)	'Interdependent'. Frequent & random NLEs (esp. facilitates)	1.000
CHAINS (360 probs)	Activities chained together via sequences of enables NLEs (1-4 chains/prob)	0.995
TT (360 probs)	'Temporal Tightness'. Release - Deadline windows preclude preferred high quality (longest duration) tasks from all being scheduled.	0.949
SYNC (360 probs)	Problems contain range of different Sync_sum tasks	.971
NTA (360 probs)	'New Task Arrival'. cTaems model is augmented with new tasks dynamically during run.	.990
OVERALL (2190)	AVE:	.981

Table 1: Performance of year 1 agent over Coordinators evaluation. 'Agent Quality' is % of 'optimal'

the flexible times representation employed by the scheduler affords it an important buffer against the uncertainty of execution and exogenous events.

The agent turns in its lowest performance on the TT (Temporal Tightness) experiment classes, and an examination of the agent trace logs reveals possible reasons. In about half of the TT problems the year 1 agent under-performs on, the specified time windows within which an agent's activities must be scheduled are so tight that any scheduled activity which executes with a longer duration than the expected value, causes a deadline failure. This constitutes a case where more sophisticated reasoning over success probability would benefit this agent. The other half of under-performing TT problems involve activities that depend on facilitation relationships in order to fit in their time windows (recall that facilitation increases quality and decreases duration). The limited facilitates reasoning performed by the year 1 scheduler sometimes causes failures ~~to install~~ a heavily facilitated initial schedule. Even when such activities are successfully installed they tend to be prone to deadline failures -If a source-side activity(s) either fails or exceeds its expected duration the resulting longer duration of the target activity can violate it's time window deadline.

8. STATUS AND DIRECTIONS

Our current research efforts are aimed at extending the capabilities of the Year 1 agent and scaling up to significantly larger problems. Year 2 programmatic evaluation goals call for effectively solving problems on the order of 100 agents and 10,000 methods, placing much higher computational demands on all of the agent's components. We have recently completed a reimplemention of the prototype agent designed to address some ~~basic~~ recognized performance issues. In addition to verifying that the performance on Year 1 problems is matched or exceeded, we have recently run some successful tests with the agent on a few 70 agent problems.

To fully address various scale up issues, we are investigating a number of more advanced coordination mechanisms. To provide more global perspective to local scheduling decisions, we are introducing mechanisms for computing, communicating and using estimates of the non-local *impact* of remote nodes. To better address the problem of establishing inter-agent synchronization points, we expanding the use of task owners and qaf-specifc protocols as a means for directing coordination activity. Finally, we plan to explore the use of more advanced STN-driven coordination mechanisms, including the use of temporal decoupling [6] to insulate the actions of inter-dependent agents and the introduction of probability sensitive contingency schedules.

9. REFERENCES

[1] M. Boddy, B. Horling, J. Phelps, R. Goldman, R. Vincent, A. Long, and B. Kohout. C_taems language specification v. 1.06, October 2005.

[2] A. Cesta and A. Oddi. Gaining efficiency and flexibility in the simple temporal problem. In *Proc. 3rd Int. Workshop on Temporal Representation and Reasoning*, Key West FL, May 1996.

[3] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61–95, May 1991.

[4] K. Decker. TÆMS: A framework for environment centered analysis & design of coordination mechanisms. In G. O'Hare and N. Jennings, editors, *Foundations of Distributed Artificial Intelligence*, chapter 16, pages 429–448. Wiley Inter-Science, 1996.

[5] A. Garvey and V. Lesser. Design-to-time scheduling and anytime algorithms. In *IJCAI Workshop on Anytime Algorithms and Deliberation Scheduling*, August 1995.

[6] L. Hunsberger. Algorithms for a temporal decoupling problem in multi-agent planning. In *Proceedings 18th National Conference on AI*, 2002.

[7] S. Lemai and F. Ingrand. Interleaving temporal planning and execution in robotics domains. In *Proc. 19th National Conference on AI*, 2004.

[8] N. Muscettola, P. P. Nayak, B. Pell, and B. C. Williams. Remote agent: To boldly go where no AI system has gone before. *Artificial Intelligence*, 103(1–2):5–47, 1998.

[9] W. Ruml and M. Fromherz. On-line planning and scheduling in a high-speed manufacturing domain. In *Proceedings of the ICAPS-04 Workshop on Integrating Planning into Scheduling*, 2004.

[10] I. Shu, R. Effinger, and B. Williams. Enabling fast flexible planning through incremental temporal reasoning with conflict extraction. In *ICAPS*, 2005.

[11] S. Smith and C. Cheng. Slack-based heuristics for constraint satisfaction scheduling. In *Proc. 12th National Conference on AI*, Wash DC, July 1993.

[12] T. Wagner and V. Lesser. Design-to-criteria scheduling: Real-time agent control. In *Proceedings of AAAI 2000 Spring Symposium on Real-Time Autonomous Systems*, Stanford, CA, March 2000.