Multi-Agent Management of Joint Schedules

Stephen F. Smith[1], **Anthony Gallagher**[1], **Terry Zimmerman**[1], **Laura Barbulescu**[1] and **Zachary Rubinstein**[2]

[1] The Robotics Institute
Carnegie Mellon University
Pittsburgh PA 15213
{sfs,anthonyg,wizim,laurabar}@cs.cmu.edu

[2] Department of Computer Science
University of New Hampshire
Durham, NH 03824-3591
zack@cs.unh.edu

Abstract

In this paper, we describe an incremental scheduling framework designed to support joint management of inter-dependent schedules by multiple executing agents. We assume an uncertain execution environment and a distributed representation of the overall problem and schedule such that no single agent has a complete view. Hence as unexpected execution events force changes to individual agent schedules, agents must recognize inter-dependencies and coordinate to revise schedules in a way that continues to maximize the quality of their joint activities. Our approach combines an underlying flexible-times representation of the schedule, which exploits temporal flexibility to absorb executional uncertainty and insulate dependencies across agent schedules, with an incremental approach to schedule revision, which promotes solution stability and tends to minimize the ripple effect of change across agent schedules. We summarize basic mechanisms for constructing and revising schedules, for integrating scheduling with execution and for generating options to focus coordination with other agents. Our framework is being developed and applied within the DARPA Coordinators program.

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, and the high tempo of execution dynamics will frequently preclude any single agent from obtaining a complete global view of the problem, and hence necessitate collaborative yet localized planning and scheduling decisions. In such circumstances, individual planning/scheduling agents must be capable of (1) reconciling updates of execution results and state with expectations contained in the current schedule and recognizing the need for plan/schedule change; (2) generating local options for unilateral response to unexpected events for purposes of keeping execution going while better joint actions are being pursued; (3) selectively generating non-local options that offer the possibility of a higher utility response but require some amount of change to inter-dependent aspects of other agents' plans/schedules; and (4) responding to queries initiated by other agents in an attempt to explore the viability of generated non-local options and to implement a coordinated multi-agent response.

In this paper, we present a scheduling framework that promotes joint management of inter-dependent schedules by multiple agents in an uncertain execution environment, under the assumption that no single agent has global visibility of the entire schedule. We focus on a specific type of multi-agent scheduling problem in which actions that are executed within their constraints accrue quality and the overall objective is to maximize quality.

We take as our starting point a flexible-times representation of a given agent's schedule, where the execution intervals associated with scheduled actions are not fixed, but are allowed to float within imposed time and action sequencing constraints. This flexible times representation of the schedule and its underlying implementation as a simple temporal network (STN) model, offers several advantages. First, it allows the explicit use of slack as a hedge against simple executional uncertainty (e.g., action durations). We define a schedule execution policy that allows immediate execution of any action within its latest start time window and whose enabling activities have been executed. This allows the agent's executor thread to proceed with looser coupling to the scheduler thread. Second, it provides a convenient basis for managing "current time" as execution proceeds and confirming the continuing viability of scheduled actions. Following suggestions of previous work (Hunsberger 2002), we specify an STN-based procedure for aligning the agent's schedule with updated values of "current time" and detecting situations where one or more actions in the schedule can no longer be feasibly executed as planned. Third, a flexible times representation provides a natural criterion for detecting the need to trigger a rescheduling process, i.e., a violation of one or more constraints in the current schedule. In a fixed-times schedule, conflict detection simply implies that the start and end times projected from the execution do not precisely match those in the current schedule. In contrast with this, the detection of a conflict (or negative cycle) in the underlying STN network implies that all available slack has been eliminated and that change is required to the sequence of actions that is currently scheduled. We introduce techniques for recovering from detected STN conflicts and taking appropriate rescheduling actions. Finally, an STN-based representation can provide principled guidance for proposing non-local rescheduling options. Building from previous research (Smith, Hildum, & Crimm 2005), we develop a new *conflict-directed* approach to identifying good opportunities for coordinated, multi-agent schedule change.

In its most basic form, an agent in our framework consists of the following components: (1) a scheduler, which operates on an STN representation of the agent's portion of the current problem and solution and incrementally manipulates this solution as the dynamics of execution dictate, (2) a distributed state mechanism (DSM), which accepts new schedules published by the scheduler and broadcasts selected elements to remote agents with dependent decisions, (3) an executor, which also receives updated schedules from the DSM, and repeatedly initiates scheduled actions as they become eligible for execution, and (4) a negotiator, which requests non-local options from the scheduler and attempts to implement them through explicit coordination actions. In collaboration with SRI International, we have developed an initial implementation of this agent architecture as part of the DARPA Coordinators program. In this paper, we take a scheduler-centric view of this agent architecture, and describe how the scheduler's interactions support and focus the multi-agent schedule execution and management process.

The remainder of the paper is organized as follows. First we introduce the *C_TAEMS* scheduling problem that our scheduler is currently designed to address, and its Hierarchical Task Network (HTN) style representation of problems and initial schedules. Next, we summarize the single agent scheduler, first discussing its core STN infrastructure and then its incremental, quality-driven scheduling algorithm. We then discuss the scheduler's interaction and operation with the executor, emphasizing the loosely-coupled but nonetheless closed-loop approach to schedule management, and the STN-based mechanisms that make this possible. Next, we turn attention to the issue of coordinating local rescheduling actions with other agents. We consider techniques for generating non-local options and give examples of how they can be used. Before closing, we discuss other related work in multi-agent schedule management. Finally, we sketch our current research plans.

The Coordinators Problem

The Coordinators problem specifies a group of agents that have to collaborate to execute a network of tasks in a highly dynamic environment. The tasks are distributed among the agents such that no agent has a complete, "objective" view of the whole problem. Instead, each agent has a "subjective view" of the problem, containing only the tasks for which it is responsible and the remote tasks that have interdependencies with the local tasks. The agents need to be able to coordinate their actions and quickly react to changes in the environment, while keeping their state consistent with the execution state and responding to changes with actions that ensure continuity in execution. A solution to the problem is a schedule that specifies for each agent which tasks to execute and when. This schedule continuously changes based on execution updates; the problem specifies an initial schedule for each agent. The execution of each task has a quality value assigned to it. The problem objective is to maximize the quality of the solution.

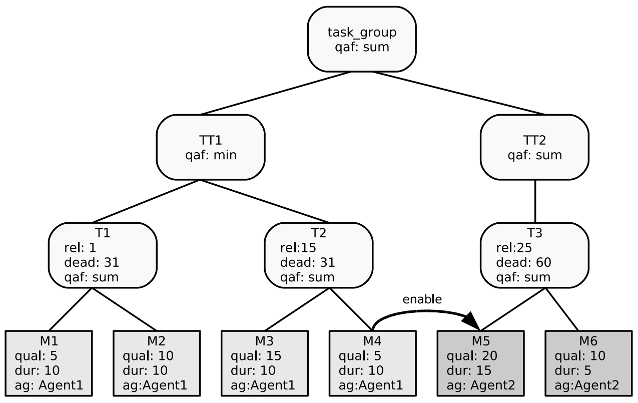

Figure 1: Problem represented in C_TAEMS (objective view)

The problems are formally specified using a version of the *TAEMS* language (Task Analysis, Environment Modeling and Simulation) (Horling *et al.* 1999) called *C_TAEMS* (Boddy *et al.* 2005). The activities to be executed by an agent are represented as a hierarchy of tasks (tree) that is intended to model various levels of abstraction corresponding to the tasks. This tree structure is similar to hierarchical task network (HTN) planning representations; in fact, *TAEMS* is often used as the annotation language on top of HTN plans.

As indicated in Figure 1, at the highest, most abstract level, the root of the tree is a special task called the task group. On subsequent levels, the tasks constitute aggregate activities, which can be decomposed on the lower levels into other tasks and/or real-world activities to be executed, termed 'methods'. These methods appear at the leaf level in Figure 1. Each declared method a can only be executed by a specified agent (denoted by $ag(a)$ in Figure 1) and each agent can be executing at most one method at any given time (i.e. agents are unit-capacity resources). Methods acquire quality once they are executed and the actual value is based on an assigned discrete probability distribution.[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 corresponding methods are executed. For example, a task with a *min* qaf will accrue the quality of its child with lowest quality *if* all children accumulate positive quality when executed. 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 sum or the maximum quality of all children that executed.

Activity interactions in the problem are modeled via non-local effects (*nles*). Two types of nles can be specified:

[1]Typically, a Coordinators problem has specified distributions for method quality and duration, and only after execution are the actual values for a method made known. For simplicity, Figures 1 and 2 show only fixed values for methods.

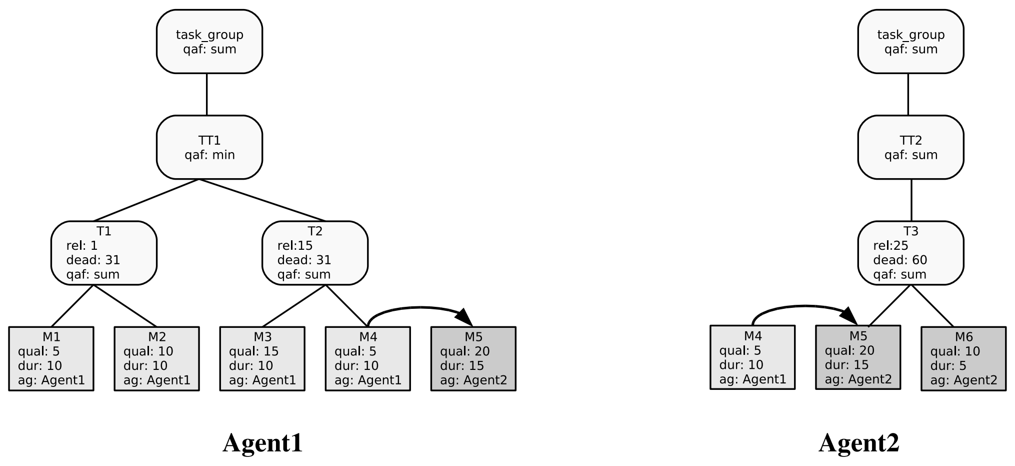

Figure 2: Subjective views for the two agents in the problem.

"hard" and "soft". The "hard" nles are used to model 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. The "soft" nles don't need to be enforced, however, when they are, they amplify (or dampen) the quality of the recipient task.

Both for tasks and for methods, release times and deadlines can be defined. Consider an activity a (either a task or a method). We denote the release time $rel(a)$ and the deadline $dead(a)$; these times define the window within which a must occur. Activities may also inherit these times from ancestor tasks at any higher level. As time progresses and/or the problem evolves, the effective time window of a is likely to become tighter than the initial time window specified by $rel(a)$ and $dead(a)$. The boundaries of the effective time window of a are defined by its earliest start time, $est(a)$, and its latest finish time, $lft(a)$. We also define the latest start time of an activity $lst(a) = lft(a) - dur(a)$ and the earliest finish time as $eft(a) = est(a) + dur(a)$, where $dur(a)$ is the duration of the method.

The Scheduler

The design of our agent scheduler builds on our previous experience with *incremental*, *flexible-times* scheduling techniques (Smith, Becker, & Kramer 2004; Smith, Hildum, & Crimm 2005; Gallagher, Zimmerman, & Smith 2006 to appear). 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, where the selection of precise method start times and finish times is delayed until execution time, adds additional leverage in an uncertain, multi-agent execution environment. On one hand, slack can be used as a hedge against uncertain method execution times. On the other, it provides a basis for softening the impact of interdependencies 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.

STN Infrastructure

To maintain the range of admissible values for the start and end times of various methods in a given agent's schedule, we encode all problem and scheduling constraints impacting these times in an underlying Simple Temporal Network (STN)(Dechter, Meiri, & Pearl 1991). A STN represents temporal constraints as a graph $G < N, E >$, where nodes in N represent time points, and edges in E are distances (labeled as $<$ lower bound, upper bound $>$ pairs) between the time points in N. A special time point, called *calendar zero* grounds the network and has the value 0. The network maintains lower and upper bounds on the time points by propagating the bounds on the distances of the edges. 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. Scheduling decisions generally correspond to the introduction of new constraints into the network (e.g., sequencing two methods assigned to the same agent) or the adjustment of existing constraints (e.g., refining the duration of an activity, changing the earliest start time). In either case, constraint propagation updates the bounds of affected nodes and checks for *cycles* in the resulting network. The lack of any such *cycle* ensures continued temporal feasibility of the plan. Otherwise a *conflict* has been detected, and backtracking (or some amount of constraint relaxation) is necessary. Our implementation utilizes an incremental bounds propagation and conflict checking algorithm based on (Cestar & Oddi 1996).

Generating and Maintaining High-Quality Schedules

The scheduler consists of two key components: a *quality propagator* and an *activity allocator* that work in a tightly integrated loop. The quality propagator parses 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 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.

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 attempts to install the *contributors* of the taskgroup identified by the quality propagator on the agent's timeline. An overview of the allocator's algorithm is shown in Figure 3. The *contributors* list is processed by the *Filter* routine. This routine uses a quality-centric heuristic to sort the methods in decreasing quality order. In addition, methods associated with a *min* task are grouped together, due to the requirement that a *min* task only accumulates quality if all its children are scheduled.

The allocator iteratively removes the first method m_{new} from the agenda and attempts to install it. The *Install* routine starts by ensuring that all the activities that enable m_{new} have been scheduled, and attempts to install any enabler that is not. If any of the enabler activities fails to install, the routine fails. Next, the *enables* constraints linking the enabler activities to m_{new} are activated. The STN rejects any infeasible *enabler* constraint by indicating a *conflict*. If a conflict is detected, the routine uninstalls any enabler activity it had scheduled and returns failure. Finally, a place on the agent's timeline must be found for the method: *Install* focuses on the portion of the agent's timeline within m_{new}'s time window and attempts to insert m_{new} by placing it between two currently scheduled methods. At the STN level, m_{new}'s insertion breaks the sequencing constraint between the two methods currently on the timeline, and inserts two new sequencing constraints that chain m_{new} to these two methods. If the insertion of both sequencing constraints succeeds, a place on the timeline for m_{new} has been found, and the routine returns success. Otherwise, if the STN returns a cycle, we re-chain the two original methods and try to insert m_{new} elsewhere. If we don't succeed in inserting m_{new} anywhere within its time window, the routine returns failure.

If a method fails to install, any currently scheduled methods that are not part of the new agenda are uninstalled by the *DeallocateNonContributors* routine.

```
Allocate($Taskgroup$)
1. $SContr \leftarrow$ Filter(Contributors($Taskgroup$))
2. while $SContr$
3.    $Method \leftarrow$ Pop($SContr$)
4.    $Success \leftarrow$ Install($Method$)
5.    if Not $Success$
6.       $SContr \leftarrow$ Filter(Contributors($Taskgroup$))
7.       DeallocateNonContributors($Taskgroup$)
8.    end-if
9. end-while
end
```

Figure 3: Algorithm for Installing Taskgroup's Contributors

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 in order to handle execution within a scheduler is a dynamic model for current time. We employ a model proposed by (Hunsberger 2002) 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 an update to current time, the link between calendar-zero and current-time is updated to reflect this new time, and as a consequence this 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 prescribed set of 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*, a 'what-if' mode of invoking the scheduler which is initiated by the negotiator in pursuit of 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 and then globally higher quality schedules, entailing inter-agent coordination are pursued by the negotiator as time permits. As such, the default scheduling mode for updates is to seek the highest quality local option according to its search strategy, instantiate it as its current schedule, and notify the executor of the revision.

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 and the agent can proceed to communicate the time at which it has accumulated quality to any remote agents linked to the method via an *nle*.

However if the method completes with a duration shorter than expected (i.e., finishes earlier than expected), 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 alternatively, 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 that can be received by the scheduler include:

- *method start* - If a method sent for execution is tarted within its $[est, lst]$ window, the scheduler response is to mark it as 'in-process'. A method cannot start earlier than when it is transmitted by the executor but due to vagaries of the simulation system, and indeed in a real-world situation, 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 scheduler shortens the scheduled duration 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. After the current-time link in the STN is updated (as described above), if a conflict results it's indicative of an unanticipated state. In this case, the scheduler proceeds as if execution is consistent with its expectations, subject to possible later updates.

Responding to Model Updates

The scheduler 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 a Coordinators problem run 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; only if posting of the revised constraints leads to an STN conflict is a rescheduling response is required. In the case of non-temporal model changes, alternatively, rescheduling action is currently always initiated.

Coordinating with Other Agents

Unexpected execution results and/or model changes will frequently necessitate joint action by multiple agents. Consider the example of an *enable* nle between two methods, where different agents are responsible for scheduling the source and target of the nle, respectively. Without a local incentive to schedule the source, the target will never get scheduled. Moreover, even if the source does get scheduled, the agent owning the target will not schedule it unless it is notified that its activity is now enabled. Finally, if an execution bottleneck delays the scheduled start of the source, it may no longer be possible to feasibly execute the target. Communication and coordination between the agents involved is fundamental to maximizing the quality gain in all such circumstances. In this section, we summarize the mechanisms provided by the scheduler to support multi-agent coordination.

Implicit Coordination

A basic means of coordination is provided within our assumed agent architecture by a Distributed State Mechanism (DSM), which is responsible to for communicating changes made to the model or schedule of a given agent to other "interested" agents. More specifically, a given agent's DSM pushes any changes that are made to a task or method in its subjective view to all other agents that also have that same task or method in their subjective views. A recipient agent treats 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 reacted to in precisely the same manner as are updates reflecting schedule execution results, potentially triggering the local scheduler if the need to reschedule is detected.

Generating Non-Local Options

As mentioned in the previous section, the scheduler'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 (re)scheduling actions that are consistent with all currently known constraints originating from other agents' schedules, and hence can be implemented without interaction with other agents. It may frequently be the case, however, that a larger-scoped change to the schedules of two or more agents can produce a higher-quality response. To support such coordinated action by two or more agents, the scheduler also provides basic mechanisms for generating non-local options.

Generally speaking, a non-local option produced by a given agent identifies a certain set of relaxations (to one or more constraints imposed by methods that are scheduled by one or more remote agents) that enables the generation of a higher quality local schedule. A non-local option can be used by a coordinating agent to formulate queries to other agents, in order to determine the impact of such relaxations from their local scheduling perspective. If the combined quality change found as a result of issuing relevant queries is a net gain, then this joint rescheduling action is subsequently committed to.

The scheduler currently provides two mechanisms for generating non-local options:

- *optimistic synchronization* - where search is directed at exploring the potential impact on quality of making optimistic assumptions about currently unscheduled, remote enablers (or targets), and
- *conflict-driven relaxation* - where analysis of STN conflicts associated with local unscheduled methods is used to identify and prioritize remote constraints to drop.

These two types of non-local options are discussed in more detail in the subsections below.

Optimistic Synchronization To produce this type of non-local options, the scheduler looks for unscheduled local methods that need to be enabled by remote activities before they can be scheduled. For each such local method, the scheduler attempts to schedule it, assuming the remote enablers are scheduled. If successful, the scheduler produces a non-local option, specifying the new schedule and the *must-schedule* activities, which are the activities that need to be scheduled by other agents.

Consider again the example in Figure 2. The maximum quality that Agent1 can contribute to the task group is 15 (by scheduling $M1$, $M2$ and $M3$). The maximum quality that Agent2 can contribute to the task group when $M5$ is not scheduled is 10; the total quality accumulated for the task group would then be $15 + 10 = 25$. If $M5$ becomes enabled, and both $M5$ and $M6$ are scheduled, the quality contributed by Agent2 to the task group would be 30. For this, $M4$ needs to be scheduled; because of the time window constraints, $M3$ would not be scheduled anymore and the new quality contributed by Agent1 to the task group would be at most 5. However, since $M5$ is enabled, the total quality for the task group node would be $5+30 = 35$. Here, in order for a global higher quality to be achieved (35 as opposed to 25), Agent1 has to produce a local schedule that has a lower quality value than the best it can obtain.

In optimistic synchronization, Agent2 will produce an option that designates:

- Both $M5$ and $M6$ are scheduled. The quality contributed by Agent2 to the task group is $20 + 10 = 30$.
- $M4$ is marked as a *must-schedule* for Agent1.

The coordination mechanism can then *query* Agent1 about the impact of scheduling $M4$. A query is handled by a

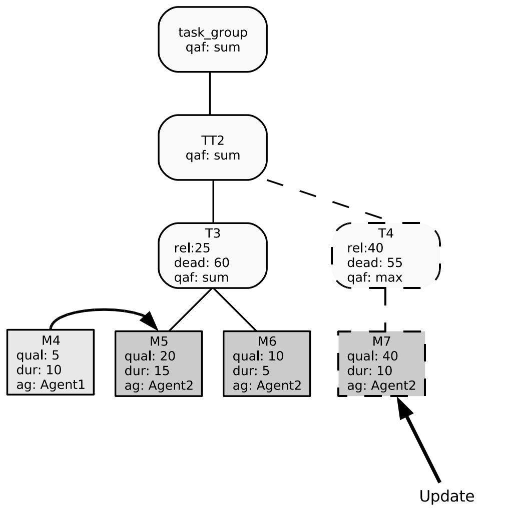

Figure 4: A high quality task is added to the task structure of Agent2. The new task and method are represented with dotted lines.

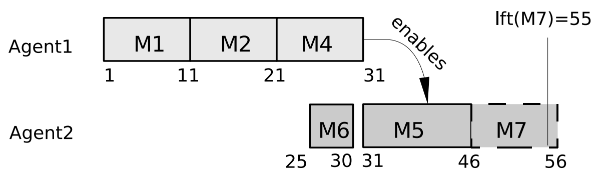

Figure 5: If $M4$, $M5$ and $M7$ are scheduled, a conflict is detected by the STN. $M7$ can not fit into the schedule, it exceeds its deadline.

recipient agent's scheduler in the same way as an update, except that changes are made temporarily and then rolled back. The coordination mechanism can infer that the overall quality if this non-local option were pursued would go down by 10 for Agent1 but up by 20 for Agent2; so overall the quality would improve by 10.

Conflict-Driven Relaxation When generating this type of non-local option, the scheduler assumes that constraints associated with remote methods can be relaxed in order to schedule a currently unscheduled local method. An option specifies a schedule and the constraints that need to be relaxed by other agents.

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 during execution, a model change update adds a new task to the task structure of Agent2 (see Figure 4). If Agent2 could schedule $M7$, the quality contributed by Agent2 to the task group would be 70. However, if $M5$, $M6$ and $M7$ were scheduled, $est(M7) = 46$ (see Figure 5). The deadline for $M7$ is $lft(M7) = 55$ and its duration is $dur(M7) = 10$. So, with the current schedules for Agent1 and Agent2, $M7$ can not be scheduled ($lft(M7) - est(M7) > dur(M7)$).

The basic steps in conflict-driven relaxation are:

- Schedule a currently unscheduled local method to generate a cycle in the STN. Agent2 attempts to schedule $M7$ by posting a sequencing constraint between $M5$ and $M7$. The STN detects a conflict (negative cycle); the methods involved in the conflict set are $M4$, $M5$ and $M7$.
- Filter conflict set for remotely imposed constraints in the manner successfully employed in (Smith, Hildum, & Crimm 2005). In our example, such constraints would be related to $M4$ – in particular, the earliest finish time for $M4$ is violated (needs to be 30 instead of 31).
- Selectively relax and recheck STN consistency. Making $eft(M4) = 30$ would bring back the consistency in the STN.

The coordination mechanism can then query Agent1 about the possibility of scheduling $M4$ such that it finishes by time 30. Agent1 will generate an option showing either $M1$ or $M2$ unscheduled. Note that the schedule quality for Agent1 will not be affected for this particular example (because $TT1$ has a *min* qaf). However, in general the quality of the schedule that has to accommodate the constraint relaxation might decrease. The negotiator will then have to reason about the combined effect of the schedule quality increases and possible decreases.

Related Work

Previous research by the scheduling community has demonstrated the advantages of using STNs to keep a *flexible-times* schedule. Several applications using STN-based schedulers working in a tight-loop with the executor module have appeared. (Muscettola *et al.* 1998) present the Remote Agent framework to control NASA rovers. (Lemai & Ingrand 2004) outline the IxTeT-eXeC continuous planner for online applications. This planner specializes in domains with independent subsystems. (Ruml & Fromherz 2004) describe a planner/scheduler for high-speed manufacturing. To handle execution uncertainty (Morris & Muscettola 2000) replace the STN with a 'family' of STNs, where each 'family' member encodes different alternative constraints from the potential outcome distributions of the activities. (Wehowsky 2003) augments the STN to encode potential choices of activities in the STN graph; he uses this type of STN in the context of a continuous planner in a multi-robot application. (Shu, Effinger, & Williams 2005) present a novel STN-based solution that allows the introduction of several (potentially inconsistent) constraints into the network and implement an iterative algorithm for reverting the STN back to a consistent state. We are currently studying the potential use of this new technology in our application. To the best of our knowledge, all of these STN-based applications in the literature present centralized solutions. Our approach extends the current state-of-the-art by producing the first distributed STN-based scheduling application that works in a tight loop with execution.

The multi-agent/multi-robot systems community has produced a variety of techniques for coordinating the activities of a group of agents. However, a great emphasis has been placed on deciding how to divide the tasks among the agents with relatively simple techniques used for task scheduling. TAEMS (which C_TAEMS extends) was designed as a planner-independent task modeling framework to encode agents' goals in a Hierarchical Task Network (HTN). TAEMS highlights the challenges and opportunities for more sophisticated distributed task scheduling approaches, and has been used in a variety of domains. Design-to-time scheduling (Garvey & Lesser 1995) and design-to-criteria scheduling (Wagner & Lesser 2000) have been designed to produce coordinated agent schedules based on TAEMS task structures. Both techniques use sophisticated heuristics to drive the scheduling process. The resulting schedules are *fixed-times*. To our knowledge, only a few previous approaches have examined the use of *flexible-times* STN-based schedules in a multi-agent setting. (Hunsberger 2002) presents one such approach, where an initial global STN is decomposed into several 'temporally decoupled' STNs that can then be provided to the agents in the system.

Status and Directions

The scheduling framework summarized in this paper has been embedded in SRI International's cMatrix agent architecture, and is currently undergoing a year 1 evaluation test that entails closed-loop multi-agent management of schedules across a range of C_TAEMS problem instances (running with the MASS C_TAEMS simulator). Whereas our year 1 effort has focused on getting foundational scheduling and coordination components in place, our major focus in the coming year will be solidifying and and extending the multi-agent coordination capabilities. Numerous opportunities exist for more extensively exploiting the STN conflict detection capabilities of the scheduler to provide a wider range of options to the negotiation component. When the agent has spare cycles to devote to exploration of the scheduling search space, the possibilities for anytime streaming of higher quality or less disruptive options based on increasing depths of local search may prove valuable. Finally, we intend to place greater emphasis on schedule stability and explore the applicability of schedule partitioning concepts like temporal decoupling (Hunsberger 2002).

Acknowledgements

The design of the mechanism used to manage current time in the STN has benefited from conversations with Luke Hunsberger and we are currently collaborating with him to exploit his concept of temporal decoupling in the Coordinators multi-agent scheduling context. The overall agent architecture within which our schedule management framework has been embedded was designed and developed by Andrew Agno, Roger Mailler and Regis Vincent at SRI International. This paper is based on work supported by the Department of Defense Advance Research Projects Agency (DARPA) under Contract # FA8750-05-C-0033. Any opinions, findings and conclusions, or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of DARPA.

References

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

Cestar, A., and Oddi, A. 1996. Gaining efficiency and flexibility in the simple temporal problem. In *Proc. 3rd Int. Workshop on Temporal Representation and Reasoning*.

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

Gallagher, A.; Zimmerman, T.; and Smith, S. 2006 (to appear). Incremental scheduling to maximize quality in a dynamic environment. In *Proc. 2006 International Conference on Automated Planning and Scheduling*.

Garvey, A., and Lesser, V. 1995. Design-to-time scheduling and anytime algorithms. In *Proc. IJCAI-95 Workshop on Anytime Algorithms and Deliberation Scheduling*.

Horling, B.; Lesser, V.; Vincent, R.; Wagner, T.; Raja, A.; Zhang, S.; Decker, K.; and Garvey, A. 1999. The taems white paper.

Hunsberger, L. 2002. Algorithms for a temporal decoupling problem in multi-agent planning. In *Proc. 18th National Conf. on AI (AAAI-02)*.

Lemai, S., and Ingrand, F. 2004. Interleaving temporal planning and execution in robotics domains. In *Proc. 19th National Conf. on AI (AAAI-04)*.

Morris, P., and Muscettola, N. 2000. Execution of temporal plans with uncertainty. In *Proc. 17th National Conf. on AI*.

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

Ruml, W., and Fromherz, M. 2004. On-line planning and scheduling in a high-speed manufacturing domain. In *Proc. ICAPS Workshop on Integrating Planning into Scheduling*.

Shu, I.; Effinger, R.; and Williams, B. 2005. Enabling fast flexible planning through incremental temporal reasoning with conflict extraction. In *Proc.2005 International Conf. on Automated Planning and Scheduling*, 252–261.

Smith, S.; Becker, M.; and Kramer, L. 2004. Continuous management of airlift and tanker resources: A constraint-based approach. *Mathematical and Computer Modeling*, 39(6–8):581–598.

Smith, S.; Hildum, D.; and Crimm, D. 2005. Comirem: An intelligent form for scheduling. *IEEE Intelligent Systems* 20(2).

Wagner, T., and Lesser, V. 2000. Design-to-criteria scheduling: Real-time agent control. In *Proc. AAAI Spring Symposium on Real-Time Autonomous Systems*, 89–96.

Wehowsky, A. 2003. Safe distributed coordination of heterogeneous robots through dynamic simple temporal networks. Technical report, Massachusetts Institute of Technology.