# The Soft Real-Time Agent Control Architecture *

Horling, Bryan and Lesser, Victor
University of Massachusetts
Department of Computer Science
Amherst, MA 01003
{bhorling, lesser}@cs.umass.edu

Vincent, Regis
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025
vincent@ai.sri.com

Wagner, Thomas
Honeywell Laboratories
Automated Reasoning Group
Minneapolis, MN 55418
wagner_tom@htc.honeywell.com

**Abstract**

Real-time control has become increasingly important as technologies are moved from the lab into real world situations. The complexity associated with these systems increases as control and autonomy are distributed, due to such issues as temporal and ordering constraints, shared resources, and the lack of a complete and consistent world view. In this paper we describe a soft real-time architecture designed to address these requirements, motivated by challenges encountered in a real-time distributed sensor allocation environment. The system features the ability to generate schedules respecting temporal, structural and resource constraints, to merge new goals with existing ones, and to detect and handle unexpected results from activities. We will cover a suite of technologies being employed, including quantitative task representation, alternative plan selection, partial-order scheduling, schedule consolidation and execution and conflict resolution in an uncertain environment. Technologies which facilitate on-line real-time control, including meta-level accounting, schedule caching and variable time granularities are also discussed.

## 1 Overview

In the field of multi-agent systems, much of the research and most of the discussion focuses on the dynamics and interactions between agents and agent groups. Just as important, however, is the design and behavior of the individual agents themselves. The efficiency of an agent's internal mechanics contribute to the foundation of the system as a whole, and the degree of flexibility these mechanics offer affect the agent's achievable level of sophistication, particularly in its interactions with other agents [19, 24]. We believe that a general control architecture, responsible for both the planning for the achievement of temporally constrained goals of varying worth, and the sequencing of actions local to the agent that have resource requirements, can provide a robust and reusable platform on which to build high level reasoning components. In this article, we will discuss the design and implementation of the Soft Real Time Architecture (SRTA), a generic planning, scheduling and execution subsystem designed to address these needs.

The SRTA architecture [33], a sophisticated agent control engine with relatively low overhead, provides several key features:

1. The ability to quickly generate plans and schedules for goals that are appropriate for the available resources and applicable constraints, such as deadlines and earliest start times.
2. The ability to merge new goals with existing ones, and multiplex their solution schedules.
3. The ability to efficiently handle deviations in expected plan behavior that arise out of variations in resource usage patterns and unexpected action characteristics.

The system is implemented as a set of interacting components and representations. A domain independent task description language is used to describe goals and their potential means of completion, which includes a quantitative characterization of the behavior of alternatives. A planning engine can determine the most appropriate means of satisfying such a goal within the set of known constraints and commitments. This permits the system to be able to adjust which goals it will achieve, and how well it will achieve these chosen goals based on the dynamics of the current situation. Scheduling services integrate these actions and their resource requirements with those of other goals being concurrently pursued, while a parallel execution module performs the actions as needed. Exception handling and conflict resolution services help repair and route information when unexpected events take place. Together, this system can assume responsibility for the majority of the goal-satisfaction process, which allows the high-level reasoning system to focus on goal

*Effort sponsored in part by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory Air Force Materiel Command, USAF, under agreements number F30602-99-2-0525 and DOD DABT63-99-1-0004. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. This material is also based upon work supported by the National Science Foundation under Grants No. IIS-9812755 and IIS-9988784. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. Furthermore, the views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), Air Force Research Laboratory or the U.S. Government.

selection, determining goal objectives and other potentially domain-dependent issues. For example, agents may elect to negotiate over an abstraction of their activities or resource allocations, and only locally translate those activities into a more precise form [25]. SRTA can then take this description and use it to both enforce the semantics of the commitments which were generated, and automatically attempt to resolve conflicts that were not addressed through the negotiation.

Based on this architecture, it should be clear that this research assumes sophisticated agents are best equipped to operate and address goals within a resource-bound, interdependent, mixed-task environment. In such a system, individual agents are responsible for effectively balancing the resources they choose to allocate to their multiple time and resource sensitive activities. A different approach addresses these issues through use of groups of simpler agents, which may individually act in response to single goals and only as a team address large-grained issues. In such an architecture, either the host operating system or increased communication must be used to resolve temporal or resource constraints, and yet more communication is required for the agents to effectively deliberate over potential alternative plans in context. Decomposing the problem space completely to "simple" agents does not address the problem or remove the information and decision requirements. We feel that such a design is over-decomposed, and would more effectively be addressed by more sophisticated agents capable of directly reasoning over and acting upon multiple concurrent issues, thereby saving both time and bandwidth.

In recent work on a distributed sensor network application[13], which will be discussed in more depth below, we have exploited SRTA to create a virtual organization of simple agents which is instantiated within a (typically smaller) collection of real agents. Conceptually, each "virtual" agent represents a goal, created as needed and dynamically assigned to a specific sophisticated "real" agent, based on information approximating the current resource usage of agents and the type of resources available at each agent. Each "real" agent then performed detailed planning/scheduling based on local resource availability and the priority of the goals it possessed, and multiplexed among the different goals that it was concurrently executing in order to meet soft real-time requirements. The use of sophisticated agents has also helped us to construct a resource negotiation protocol that operates on an abstract model of resources and does not need to resolve all conflicts to be successful [25]. In this case, the agent took on the responsibility of mapping the abstract resource allocation policy generated by the negotiation protocol into a detailed resource allocation schedule and where possible, resolved resource allocation conflicts at the abstract level through local shifting and modification of tasks.

SRTA operates as a functional unit within a Java-based agent, which itself it running on a conventional (i.e. not real-time) operating system. The SRTA controller is designed to be used in a layered architecture, occupying a position below the high-level reasoning component in an agent [39, 1] (see Figure 2). In this role, it will accept new goals, report the results of the activities used to satisfy those goals, and also serve as a knowledge source about the potential ability to schedule future activities by answering what-if style queries. Within this context, SRTA offers a range of features designed to provide support for operating in a distributed, intelligent environment. The goal description language supports quantitative, probabilistic models of activities, including non-local effects of actions and resources and a variety of ways to define how tasks decompose into subtasks. In particular, the uncertainty associated with activities can be directly modeled through the use of quantitative distributions describing the different outcomes a given action may produce. Commitments and constraints can be used to define relationships and interactions formed with other agents, as well as internally generated limits and deadlines. The planning process uses this information to generate a number of different plans, each with different characteristics, and ranked by their predicted utility. A plan is then used to produce a schedule of activities, which is combined with existing schedules to form a potentially parallel sequence of activities, which are partially ordered based on their interactions with both resources and one another. This sequence is used to perform the actions in time, using the identified preconditions to verify if actions can be performed, and invoking light-weight rescheduling if necessary. Finally, if conflicts arise, SRTA can make use of an extensible series of resolution techniques to correct the situation, in addition to passing the problem to higher level components which may be able to make a more informed decision.

An important aspect of most real-world systems is their ability to handle real-time constraints. This is not to say that they must be fast or agile (although it helps), but that they should be aware of deadlines which exist in their environment, and how to operate such that those deadlines are reasoned about and respected as much as possible. This notion of real-time is weaker than its relative, strict real-time, who's adherents attempt to rigorously quantify and formally bound their systems' execution characteristics. Instead, systems working in soft real-time operate on tasks which may still have value for some period after their deadlines have passed [31], and missing the deadline of a task does not lead to disastrous external consequences. Our research addresses a derivative of this concept, where systems are expected to be statistically fast enough to achieve their objectives, without providing formal performance guarantees. This allows it to successfully address domains with highly uncertain execution characteristics and the potential for unexpected events, neither of which are well suited for a hard real-time approach. As its name implies, SRTA operates in soft real-time, using time constraints specified during the goal formulation and scheduling processes, and acting to meet those deadlines whenever necessary. In this system, we have sacrificed the ability to provide formal performance guarantees in order to address more complex and uncertain problem domains. As will be shown shortly, this technology has been used to successfully operate in a real-time distributed environment.

To operate in soft-real time, an agent must know when actions should be performed, how to schedule its activities and commitments such that they can be performed or satisfied, and have the necessary resources on hand to complete them. Our solution to this problem addresses two fronts. The first is to implement the technologies needed to directly reason about real-time. As mentioned above, we begin by modeling the quantitative and relational characteristics of the goals and activities the agent may perform, which can be done a priori and accessed as plan library or through a runtime learning process[17]. This information is represented, along with other goal achievement and alternative plan information, in a TÆMS task structure [4, 12] (discussed in more detail in section 3.1). In addition to modeling primitive actions, it is also possible to model and schedule some meta-level activities, such as negotiation. This permits a costing-out of the activity's characteristics, allowing the agent to, for instance, directly reason about what sort of negotiation is appropriate for the given context. A planning component, the Design-to-Criteria scheduler (DTC) [35, 37], uses these TÆMS task structures, along with the quantitative knowledge of action interdependence and deadlines, to select the most appropriate plan given current environmental conditions. This plan is used by the Partial Order Scheduler process to determine when individual actions should be performed, either sequentially or in parallel, within the given precedence and runtime resource constraints. In general, we feel that real-time can be addressed through the interactions of a series of components, operating at different granularities, speed and satisficing (approximate) behaviors.

The second part of our solution attempts to optimize the running time of our technologies, to make it easier to meet deadlines. The partial order schedule provides an inherently flexible representation. As resources and time permit, elements in the schedule can be quickly delayed, reordered or parallelized. New goals can also be incorporated piecemeal, rather than requiring a computationally expensive process involving re-analysis of the entire schedule. Together, these characteristics reduce the need for constant re-planning, in addition to making the scheduling process itself less resource-intensive. Learning plays an important role in the long-term viability of an agent running in real time, taking advantage of the repetitive nature of its activities. Schedules may be learned and cached, eliminating the need to re-invoke the DTC process when similar task structures are produced, and the execution history of individual actions may be used to more accurately predict their future performance. A similar technique could be used to track the requisite actions and time needed to devote to particular goals. Because the planning and execution processes are distinct, a feedback loop was added to provide the planner with information describing which actions may potentially run in parallel in a given environmental or resource context. This effectively reduces the time it takes to perform a sequence of actions, which permits the planner to explore and suggest more sophisticated plans.

This article will proceed by discussing the problem domain which motivated much of this system. Functional details of the architecture will be covered, along with further discussion of the various optimizations that have been added. Experimental traces are also included demonstrating some of the features that are described. We will conclude with an overview of related research and a discussion of overall conclusions, including future directions.

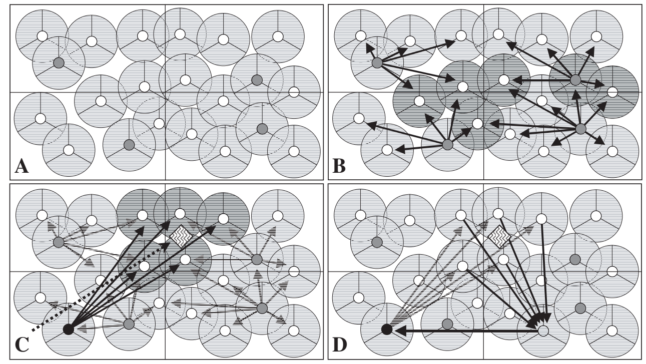



Figure 1: High-level distributed sensor allocation architecture. A) shows the initial sensor layout, decomposition and allocation of sector managers. B) shows the dissemination of scanning tasks. The new track manager in C) can be seen coordinating with sensors to track a target, while the resulting data is propagated in D) for processing.

## 2 Problem Domain

A distributed resource allocation domain which motivated much of this work [13] will be used throughout this article to ground the topics which are discussed and formulate examples. This section will briefly describe the environment and the particular challenges it offers. Components of the SRTA architecture have also been used successfully in several other domains, such as intelligent information gathering [22], intelligent home control [20], and supply chain [10].

The distributed resource environment consists of several sensor nodes arranged in a region of finite area, as can be seen in Figure 1A. Each sensor node is autonomous, capable of communication, computation and observation through the attached sensor. We assume a one-to-one correspondence between each sensor node and an agent, which serves locally as the operator of that sensor. The high level goal of a given scenario in this domain is to track one or more target objects moving through the environment. This is achieved by having multiple sensors triangulate the positions of the targets in such a way that the calculated points can be used to form estimated movement tracks. The sensors themselves have limited data acquisition capabilities, in terms of where they can focus their attention, how quickly that focus can be switched and the quality / duration tradeoff of its various measurement techniques. The attention of a sensor, or more specifically the allocation of a sensor's time to a particular tracking task, therefore forms an important, constrained resource which must be managed effectively to succeed.

The real-time requirement of this environment is derived from the triangulation process. Under ideal conditions, three or more sensors will perform measurements at the same instant in time[1]. Individually, each sensor can only determine the target's distance and velocity relative to itself. Because each node will have seen the target at the same position, however, these gathered data can then be fused to triangulate the target's actual location. In practice, exact synchronization to an arbitrarily high resolution of time is not possible, due to the uncertainty in sensor performance and clock synchronization. A reasonable strategy then is to have the sensors perform measurements within some relatively small window of time, which will yield positive results as long as the target is near the same location for each measurement. Thus, the viable length of this window is inversely proportional to the speed of the target (in our scenarios we use a window length of one second for a target moving roughly one foot per second).

Part of the resource allocation task revolves around how each node's sensor capabilities are assigned to various objectives. A tradeoff exists, for instance, between scanning for new targets in the environment by sensing in the greatest possible area, and the directed tracking of existing ones. The potential for multiple targets means that a given sensor may be able to obtain data from different sources, but because the sensor measurements cannot distinguish between targets the sensor itself can be used to gather data from only one at a time. This means both that the sensor array as a whole must be allocated appropriately to maximize their usefulness, and that individual measurements must be handled and interpreted carefully to avoid fusing data from disparate targets, which would lead to a highly inaccurate result.

Competing with the sensor measurement activity are a number of other local goals, including sector management (Figure 1A), target discovery scanning (1B), measurement tasks for other targets (1C), and data processing (1D). We don't see these as separate agents or threads, but rather as different objectives/goals that an agent is multiplexing. Note in 1C the sensor performing the track negotiation is one that previously received a scanning task. Meta-level functionality such as negotiation, planning and scheduling also contend for local resources. To operate effectively, while still meeting the deadlines posed above, the agent must be capable of reasoning about and acting upon the importance of each of these activities.

In summary, our real-time needs for this application require us to synchronize several measurements on distributed sensors with a granularity of one second. A missed deadline may prevent the data from being fused, or the resulting triangulation may be inaccurate - but no catastrophic failure will occur. This provides individual agents with some minimal leeway to occasionally decommit from deadlines, or to miss them by small amounts of time, without failing to achieve the overall goal. At the same time, there is a great deal of uncertainty in when new tasks will arrive, and how long individual actions will take, so a strict timing policy is too restrictive. Thus, our notion of real-time here is relatively soft, enabling the agents to operate effectively despite uncertainty over the behavioral characteristics of computations and their resource requirements.

Further details on this domain and the multi-agent architecture designed to address it can be found in [13].

# 3 Soft Real-Time Control Architecture

Our previous agent control architecture, used exclusively in controlled time environments, was fairly large grained. As goals were addressed by the problem solving component, they would be used to generate task structures to be analyzed by the Design-To-Criteria (DTC) scheduler. The resulting linear schedule would then be directly used for execution by the agent. Task structures created to address new goals would be merged with existing task structures, creating a monolithic view of all the agent's goals. This combined view would then be passed again to DTC for a complete re-planning and re-scheduling. Execution failure would also lead to a complete re-planning and rescheduling. This technique leads to "optimal" plans and schedules at each point if meta-level overheads are not included. As will be discussed in section 3.2, however, the combinatorics associated with such large structures can get quite high. This made agents ponderous when working with frequent goal insertion or handling exceptions, because of the need to constantly perform the relatively expensive DTC process. In a real-time environment, characterized by a lot of uncertainty in the timing of actions and the arrival of new tasks, where the agent must constantly reevaluate their execution schedule in the face of varied action characteristics, this sort of control architecture was impractical.

In the SRTA architecture, we have attempted to make the scheduling and planning process more incremental and compartmentalized. New goals can be added piecemeal to the execution schedule, without the need to re-plan all the agent's activities, and exceptions can be typically be handled through changes to only a small subset of the schedule. Figure 2 shows the new agent control architecture we have developed to meet our soft real-time needs. We will first present an overview of how it functions, and cover the implementation in more detail in later sections. In this architecture, goals can arrive at any time, in response to environmental change, local planning, or because of requests from another agents. The goal is used by the problem solving component to generate a TÆMS task structure, which quantitatively describes the alternative ways that goal may be achieved. The TÆMS structure can be generated in a variety of ways; in our case we use a TÆMS "template" library, which we use to dynamically

[1] If the tracking of the vehicle in previous time frames was very accurate relative to where the vehicle actually was, only two sensors would be needed for triangulation (where uncertainty between multiple candidate points is resolved by using the track information). However, the uncertainty of the prior track, coupled with the potential for poor quality measurements leads us to use more sensors where possible.

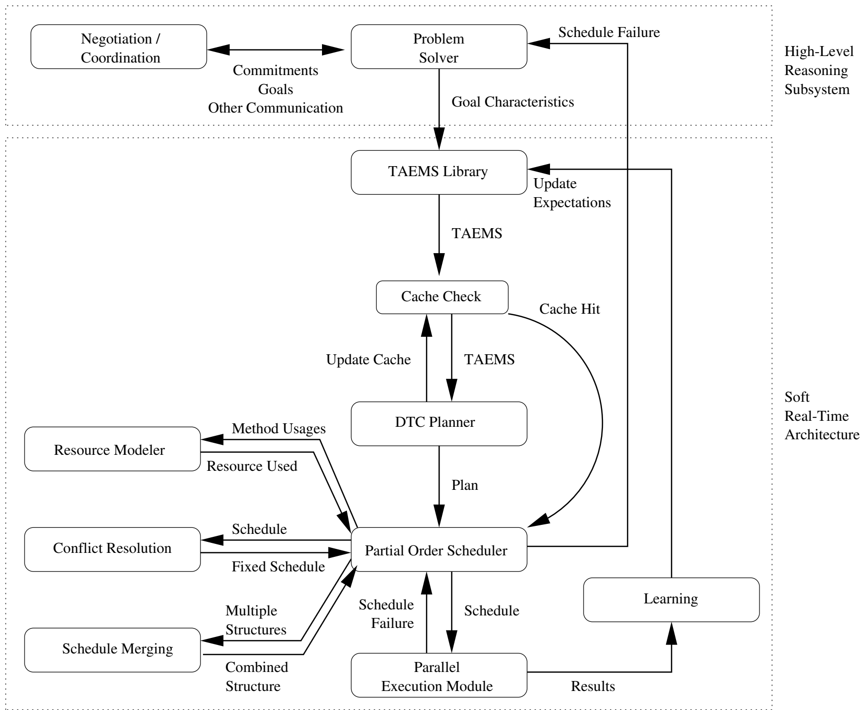



Figure 2: High-level agent control architecture.

instantiate and characterize structures to meet current conditions. Other options include generating the structure directly in code [22], or making use of an approximate base structure and then employing learning techniques to refine it over time [17].

The Design-To-Criteria component, used in the original controller described earlier, retains a critical role in SRTA. Where before it was responsible for both selecting an appropriate plan of activities and producing a schedule of actions for monolithic structures, SRTA generally exploits only its planning capabilities for discrete structures. Using the TÆMS structure mentioned above, along with criteria such as potential deadlines, minimum quality, and external commitments, DTC selects an appropriate plan.

The resulting plan is used to build a partially ordered schedule, which will use structure details of the TÆMS structure to determine precedence constraints and search for actions which can be performed in parallel. Several components are used during this final scheduling phase. A resource modeling component is used during this analysis to ensure that resource constraints are also respected. A conflict resolution module reasons about mutually-exclusive tasks and commitments, determining the best way to handle conflicts. Finally, a schedule merging module allows the partial order scheduler to incorporate the actions derived from the new goal with existing schedules. Failures in this process are reported to the problem solver, which is expected to handle them (by, for instance, relaxing constraints such as the goal completion criteria or delaying its deadline, completing a substitute goal with different characteristics, or decommiting from a lower priority goal or the goal causing the failure).

Once the schedule has been created, an execution module is responsible for initiating the various actions in the schedule. It also keeps track of execution performance and the state of actions' preconditions, potentially re-invoking the partial order scheduler when failed expectations require it. As will be shown later, the partial order scheduler can use a fast action shifting mechanism to resolve such failures with minimal overhead where possible. A learning component also monitors execution performance, which is able to update the TÆMS template library when new trends are observed.

Except where noted, the system described in this paper is a functional, existing, research-grade artifact. It is written in Java, with the exception of DTC which was implemented in C++ and is accessed through a Java native interface. As alluded to above, SRTA is a collection of interconnected components, where each component represents an encapsulated technique with a well-defined boundary and purpose. They are currently written and distributed as part of the JAF agent framework [11]. These ten or so components and their supporting classes comprise roughly 50,000 lines of Java code, while the DTC planner consists of around 40,000 lines of C++. The execution characteristics of the engine as a whole depend on the frequency and complexity of goals it is asked to achieve. On average we observe cycle times of between 50 and 100 milliseconds on 400 Mhz x86-based systems, where a cycle represents a pass through the SRTA engine analyzing current goals and executing methods, although this can jump to a half-second or more if a particularly complex situation must be analyzed. Because the system runs on conventional operating systems with no level of

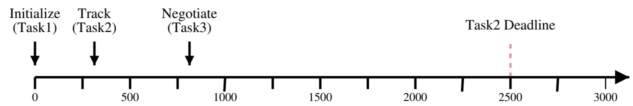



Figure 3: The timeline of events for the running scenario in the paper. Shown are the arrival times for the goals shown in Figures 4 and 7, along with the negotiated deadline for Task 2.

service guarantee, competing external processes may add an additional level of performance uncertainty.

To better explain our architecture's functionality, we will work through a example in the next several sections, using simplified versions of task structures in the actual sensor network application. The initial timeline for this example can be seen in Figure 3. At time 0 the agent recognizes its first goal - to initialize itself. After starting the execution of the first schedule it will receive another goal to track a target and sent the results before time 2500. Later, a third goal, to negotiate for delegating tracking responsibility, is received. We will show how these different goals may be achieved, and their constraints and interdependencies respected.

### 3.1 TÆMS Generation

Before progressing, we must provide some background on our task description language, TÆMS. TÆMS, the Task Analysis, Environmental Modeling and Simulation language, is used to quantitatively describe the alternative ways a goal can be achieved [4, 12]. A TÆMS task structure is essentially an annotated task decomposition tree. The highest level nodes in the tree, called task groups, represent goals that an agent may try to achieve. The goal of the structure shown in Figure 4 is `Task2`. Below a task group there will be a set of tasks and methods which describe how that task group may be performed, including sequencing information over subtasks, data flow relationships and mandatory versus optional tasks. Tasks represent sub-goals, which can be further decomposed in the same manner. `Task2`, for instance, can be performed by completing subtasks `Set-Parameters`, `Track`, and `Send-Results`.

Methods, on the other hand, are terminal, and represent the primitive actions an agent can perform. Methods are quantitatively described, with probabilistic distributions of their expected quality, cost and duration. These quantitative descriptions are themselves grouped together as outcomes, which abstractly represent the different ways in which an action can conclude. At runtime, it is the responsibility of the code executing the primitive action corresponding to the method to indicate the relevant outcome that was experienced, the quality it produced and the cost incurred. `Set-Parameters`, then, is described with two potential outcomes, `Must-Update-Parameters` and `Already-Set-Correctly`, each with its relative probability and description of expected duration. The quality of the two outcomes are the same, but the former outcome, which will happen 80% of the time, has a duration twice as long as the latter, which only occurs 20% of the time according to the model.

The quality accumulation functions (QAF) below a task describes how the quality of its subtasks is combined to calculate the task's quality. For example, the *min* QAF below `Task2` specifies that the quality of `Task2` will be the minimum quality of all its subtasks - so all the subtasks must be successfully performed for the `Task2` task to succeed. On the other hand, the *max* below `Track` says that its quality will be the maximum of any of its subtasks - the agent has a choice of one or more alternatives to complete `Track` (complete descriptions of these and other QAFs can be seen in [12]).

Interactions between methods, tasks, and affected resources are also quantitatively described as interrelationships. The *enables* interrelationships in Figure 4 represent precedence relationships, which in this case say that `Set-Parameters`, `Track`, and `Send-Results` must be performed in-order. An analogous *disables* interrelationship exists, as well as the softer relations *facilitates* and *hinders*. These latter two are particularly interesting because they permit the further modeling of choice - the agent might choose to perform a facilitating method prior to its target to obtain an increase in the latter's quality, or ignore the method to save time.

`lock2` and `release2` are resource interrelationships, describing, in this case, the *consumes* and *produces* effects method `Send-Results` has on the resource `RF`. These indicate that when the method is activated, it will consume or produce some quantity of that resource. The resource effect is further described through the *limits* interrelationship, which defines the changes in the method's execution characteristics when that resource is over-consumed or over-produced. The resource itself is also modeled, including its bounds and current value (as shown below the `RF` triangle), and whether it is consumable or not (e.g. printer paper is consumable, where the printer itself is not). In the model shown in Figure 4, these resource interrelationships are being used to describe a simple locking system, where when a method has "consumed" the RF resource, it has obtained an exclusive lock on it such that other RF-using activities cannot operate concurrently.

Together, these descriptions provide the foundation for the scheduling and planning processes to reason about the effects of selecting this method for execution, so a planner can choose correctly when the agent is willing to trade off uncertainty against quality or some other metric.

The problem solver is responsible for translating its high-level goals into TÆMS, which serves as a more detailed representation usable by other parts of the agent. This could be done by building TÆMS structures in the source code, but this tends to be impractical if the agent must define multiple complex or heterogeneous structures. On the other hand, the problem solver could read static structures from a plan library, selecting the one designed to address the particular goal in question. This works well, except it lacks the flexibility to easily handle the minor variations in structure needed when environmental conditions shift. We developed a hybrid scheme, which uses a library of TÆMS templates, which are dynamically

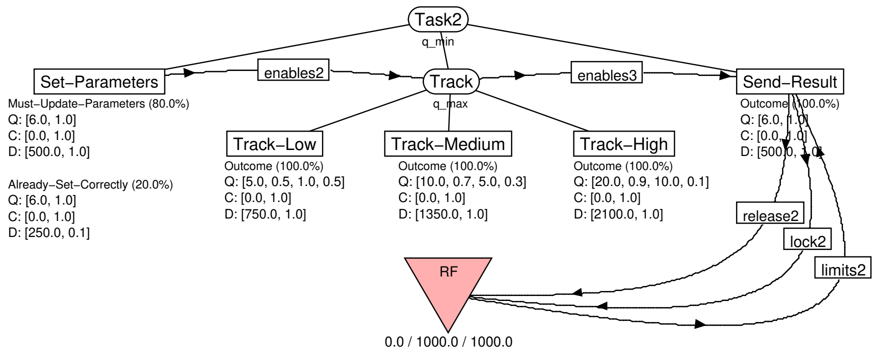



Figure 4: An example TÆMS task structure for tracking. The expected execution characteristics are shown below each method, and the `Send-Results` method in this figure has a deadline of 2500.

instantiated at runtime, taking into account the agent's current working conditions. In this way we can handle such things as varying execution performance, negotiation partners and commitment details. A small example of this is shown in table 1, which compares a template specification to a sample of the TÆMS code it might produce. In figure 4, the `Track-Medium` method must include timing and commitment information if it is being performed in response to a negotiated commitment. Similarly, if the learning component determined that `Track-Medium` was taking longer than expected, this information can be fed into the template to reflect that change. The template shown in table 1 shows how such information can be used to dynamically specify a task structure. The commitment in the figure accepts information describing the remote agent and its desired start time, deadline and minimum quality. The `Track-Medium` definition includes similar fields, and also allows the duration to be modified (in which case the default at the top will be overridden). More details on the textual specification of TÆMS structures can be found in [12].

At time 0 the agent will use its template library to generate the initialization structure seen in Figure 7A. In this structure, the agent must first `Init` and then `Calibrate` its sensor. Properties passed into the template specify the particular values used in `Init`, such as the sensor's desired gain settings or communications channel assignment, as well the number of measurements to be used during `Calibrate`. As specified by the *enables* interrelationship, `Init` must successfully complete before the agent can `Send-Message`, reporting its capabilities to its local manager. `Send-Message` also uses resource interrelationships to obtain an exclusive lock on the `RF` communication resource. Only one action at a time can use `RF` to send message, so all messaging methods have similar locking interrelationships. As we will see later, this indirect interaction between messaging methods creates interesting scheduling problems. `Task2` and `Task3`, shown in Figures 4 and 7B, respectively, are generated later in the run in a similar manner.

## 3.2 DTC Planner / Initial Scheduler

Design-to-Criteria (DTC) scheduling is the soft real-time process of evaluating different possible courses of action for an intelligent agent and choosing the course that best fits the agent's current circumstances. For example, in a situation where the RF resource is under a great deal of concurrent usage, the agent may be unable to send data using the traditional quick communications protocol and thus be forced to spend more time on a more reliable, but slower method to produce the same quality result (analogous to selecting between a UDP or TCP session). Or, in a different situation when both time and cost are constrained, the agent may have to sacrifice some degree of quality to meet its deadline or cost limitations. Design-to-Criteria is about evaluating an agent's problem solving options from an end-to-end view and determining which tasks the agent should perform, when to perform them, and how to go about performing them. Having this end-to-end view is crucial for evaluating the relative performance of alternative plans able to satisfy the goal.

As TÆMS task structures model a family of plans, the DTC scheduling problem has conceptually certain characteristics in common with planning and certain characteristics of more traditional scheduling problems, and it suffers from pronounced combinatorics on both fronts. The scheduler's function is to read as input a TÆMS task structure (or a set of task structures) and to 1) decide which set of tasks to perform, 2) decide sequencing constraints among the tasks, taking advantage of soft relationships where possible, 3) to perform the first two functions so as to address hard constraints, e.g., deadlines on tasks, and to balance the soft design/goal criteria specified by the designer, to do this computation in soft real-time so that it can be used online.

One would expect any reasonable planning process to enforce so-called "hard" constraints - ones which must be satisfied for a goal to be achieved or a commitment satisfied. It is DTC's additional ability to reason about weaker, optional interactions which sets it apart. The *sum* QAF in TÆMS , for instance, defines a task who's quality is determined by the sum of all its subtasks' qualities. In a time critical situation, DTC may opt for a shorter, but lower quality plan which only calls for one

| TÆMS Template | Resulting Definition |
|---|---|

**TÆMS Template**

```
#if (#ndef $TM_DUR)
   #define TM_DUR = 750.0 1.0
#endif

(spec_method
   (label Track-Medium)
   (agent $AGENT)
   (supertasks Track)
#if (#def($EST) == true)
   (earliest_start_time $EST)
#endif
#if (#def($DEADLINE) == true)
   (deadline $DEADLINE)
#endif
   (outcomes
      (Outcome
         (density 1.0)
         (quality_distribution 5.0 0.5 1.0 0.5)
         (duration_distribution $TM_DUR)
         (cost_distribution 0.0 1.0)
      )
   )
)

#if (#def($COMMITID) == true)
(spec_commitment
   (label commitment-$COMMITID)
   (type deadline)
   (from_agent $AGENT)
   (to_agent $TOAGENT)
   (task Track)
#if (#def($MINQ) == true)
   (minimum_quality $MINQ)
#endif
#if (#def($EST) == true)
   (earliest_start_time $EST)
#endif
#if (#def($DEADLINE) == true)
   (deadline $DEADLINE)
#endif
)
#endif
```

**Resulting Definition**

```
(spec_method
   (label Track-Medium)
   (agent Agent_A)
   (supertasks Track)
   (earliest_start_time 500)
   (deadline 2000)
   (outcomes
      (Outcome
         (density 1.0)
         (quality_distribution 5.0 0.5 1.0 0.5)
         (duration_distribution 750.0 1.0)
         (cost_distribution 0.0 1.0)
      )
   )
)

(spec_commitment
   (label commitment-1)
   (type deadline)
   (from_agent Agent_A)
   (to_agent Agent_B)
   (task Track)
   (earliest_start_time 500)
   (deadline 2000)
)
```

Table 1: The pre-TÆMS template specification for a portion of the tracking task, and the resulting structure generated after values have been specified. When the template was instantiated, the variables `AGENT`, `EST`, `DEADLINE`, `COMMITID` and `TOAGENT` were specified, while `TM_DUR` and `MINQ` were left undefined.

of these subtasks to be executed. In more relaxed conditions, more may be added to the plan. Similarly, soft interrelationships such as *facilitates* or *hinders* may be respected or not, depending on their specific quantitative effects and the current planning context. DTC's behavior is governed through the use of a criteria description, which is provided to it along with each TÆMS structure. This criteria specifies, for example, the desired balance between plan quality and duration, or what level of uncertainty is tolerable. From a user's perspective, these characteristics can be modeled with a set of sliders, as shown in Figure 5, each of which define a particular attribute of the criteria[34].

Meeting these objectives is a non-trivial problem. In general, the upper-bound on the number of possible schedules for a TÆMS task structure containing $n$ actions is given in Equation 1. Clearly, for any significant task structure the brute-strength approach of generating all possible schedules is infeasible – offline or online. This expression contains complexity from two main sources. On the "planning" side, the scheduler must consider the (unordered) $O(2^n)$ different alternative different ways to go about achieving the top level task (for a task structure with $n$ actions). On the "scheduling" side, the scheduler must consider the $m!$ different possible orderings of each alternative, where $m$ is the number of actions in the alternative. Despite the fact that DTC is not used for the actual scheduling of activities in SRTA, the scheduling analysis is still necessary when quantitatively comparing candidate plans because an end-to-end view is required to calculate the properties of a proposed plan.

$$\sum_{i=0}^{n} \binom{n}{i} i! \tag{1}$$

The types of constraints that may be present in TÆMS and the existence of interactions between tasks (and the different $QAFs$ that define how to achieve tasks), prevent a simple, optimal solution approach. DTC copes with the high-order combinatorics using a battery of techniques. Space precludes detailed discussion of these, however, they are documented in [35]. From a very high level, the scheduler uses goal directed focusing, approximation, scheduling heuristics, and schedule improvement/repair

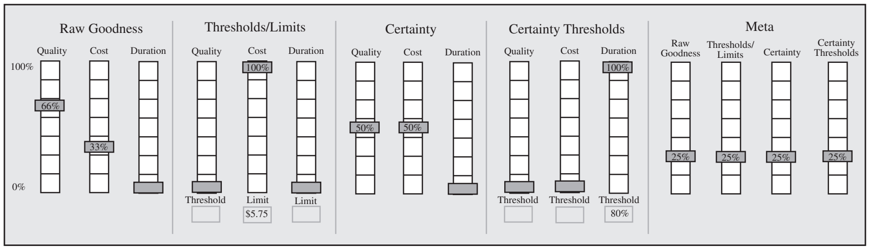



Figure 5: The "slider" model for specifying and interpreting the range of criteria DTC uses to weight its plan generation and selection process.

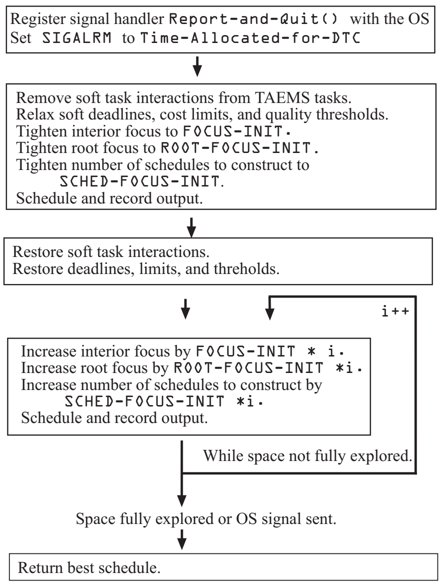



Figure 6: Real-Time Control for DTC

heuristics [44, 30] to reduce the combinatorics to polynomial levels in the worst case.

The Design-to-Criteria scheduling process falls into the general area of flexible computation, [14] but differs from most flexible computation approaches in its use of multiple actions to achieve flexibility (one exception is [15]) in contrast to *anytime algorithms* [3, 29, 42]. We have found the lack of restriction on the properties of primitive actions to be an important feature for application in large numbers of domains. Another major difference is that in DTC we not only propagate uncertainty [43], but we can work to reduce it when in the criteria for achieving a goal designates this characteristic as important.

Until recently, DTC supplied online scheduling and planning services to other components by being "fast enough" for the activities being scheduled. For example, in the BIG information gathering agent [22], scheduling/planning accounted for less than 1% of the agent's execution time - but this was in situations where problem solving episodes were in the scale of minutes. In tighter real-time situations, being fast enough may not be sufficient, as discussed in [37]. The current generation scheduler supports hard real-time deadlines governing its execution time at the grainsize afforded by the Unix/Linux operating system. The control algorithm used by the scheduler is shown in Figure 6. To meet hard deadlines on the amount of time the scheduler can take to plan/schedule, it first relaxes constraints that are likely to produce worst-case behavior and schedules. It then records the most highly rated schedule, restores a portion of the constraints, and schedules again. This schedule is also recorded. The scheduler then lessens its degree of focusing, enabling it to explore a larger percentage of the schedule solution space, and reschedules. The resulting schedule is recorded, the degree of focusing is decreased again, and the scheduler again reschedules. This process continues until the hard-deadline is met or the scheduler explores the entire scheduling space. If the hard deadline occurs before the scheduler is able to produce a single viable schedule, no schedule is returned to the client. For a specific application, we can thus set a time limit for DTC to operate within. However, this capability also allows for the more interesting possibility of a meta-level control component which adapts scheduling duration over time[28, 27].

As with most hard real-time applications, there is a minimum temporal grainsize below which no solutions will be produced. With TÆMS scheduling, the minimum temporal floor is defined by the characteristics of the problem instance, e.g., number and types of interdependencies, constraint tightness, existence of alternative solution methods, classes of QAFs, etc. Predictability [31] in a hard real-time sense is thus still lacking. In general, the issue returns to the grainsize of the problem. For some applications, a hard scheduling deadline of one second is reasonable, whereas for others, twenty seconds may be required to produce a viable result. In the distributed sensor application, the scheduler grainsize is too great, particularly when rescheduling occurs frequently, as discussed below. Thus, additional, secondary measures were needed to decrease the frequency and duration of DTC's scheduling sessions. These tactics included using a caching system (see section 4.2) and reducing complexity by planning over individual goals whose schedules are later merged, rather than directly over a single aggregate goal.

Returning to our example, DTC is used to select the most appropriate set of actions from the initialization task structure. In this case, it has only one valid plan: `Init`, `Calibrate`, and `Send-Message`. A more interesting task structure is seen in `Task2` from figure 4, which has a set of alternative methods under the task `Track`. A deadline is associated with `Send-Result`, corresponding to the desired synchronization time specified by the agent managing the tracking process. In this case, DTC must determine which set of methods is likely to obtain the most quality, while still respecting that deadline. Because TÆMS models duration uncertainty, the issue of whether or not a task will miss its deadline involves probabilities rather than simple discrete points. The techniques used to reason about the probability of missing a hard deadline are presented in [37]. It selects for execution the plan `Set-Parameters`, `Track-Medium`, and `Send-Results`. After they are selected, the plans will be used by the partial order scheduler to evaluate precedence and resource constraints, which determine when individual methods will be performed.

### 3.3 Partial Order Scheduler

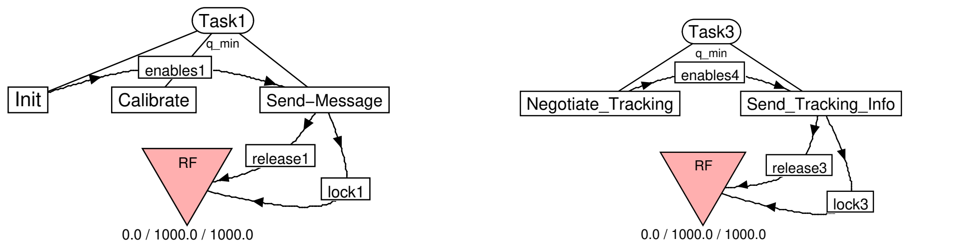
DTC was designed for use in both single agents and agents situated in multi-agent environments. Thus, it makes no assumption about its ability to communicate with other agents or to "force" coordination between agents. This design approach, however, leads to complications when working in a real-time, multi-agent environment where distributed resource coordination is an issue. When resources can be used by multiple agents at the same time, DTC lacks the ability to request communication for the development of a resource usage model. This is the task of another control component that forms scheduling constraints based on an understanding of resource usage. In most applications, these constraints are formed by rescheduling to analyze the implications of particular commitments. In the real-time sensor application, the rescheduling overhead is too expensive for forming these types of relationships. The solution we have adopted is to use a subset of DTC's functionality, and then offload the distributed resource and fine grained scheduling analysis to a different component - the partial order scheduler. Specifically, DTC is used in this architecture to reason about tradeoffs between alternative plans, respect ordering relationships in the structure, evaluate the feasibility of soft interactions, and ensure that hard duration, quality and cost constraints are met.

DTC presents the partial order scheduler with a linear schedule meeting the requested deadline. Timing details, with the exception of hard deadlines generated by commitments to other agents and overall goal deadlines, are ignored in the schedule, which is essentially used as a plan. The partial order scheduler uses this to build a partially ordered schedule, which includes descriptions of the interrelationships between the scheduled actions in addition to their desired execution times. This partially ordered schedule explicitly represents precedence relationships between methods, constraints and deadlines. This information arises from commitments, resource and method interrelationships, and the QAFs assigned to tasks and is represented as a precedence graph. This graph can then be used both to determine which activities may potentially be run concurrently, because they have no precedence relation between them or they do not have interfering resource usages[2], and where particular actions may be placed in the execution timeline. Of particular significance, this latter functionality allows the scheduler to quickly reassess scheduled actions in context, so that some forms of rescheduling can be performed with very low overhead when unexpected events require it. Much of this information can be directly determined from the TÆMS task structure.

Consider the tracking task structure shown in Figure 4. *Enables* interrelationships between the tasks and methods indicate a strict ordering is necessary for the three activities to succeed. In addition (although not shown in the figure), a deadline constraint exists for `Send-Result`, which must be completed by time 2500. Next look at the initialization structure in Figure 7A. While an *enables* interrelationship orders `Init` and `Send-Message`, it does not affect the `Calibrate` method. Internally, the partial order scheduler will use this information to construct a precedence graph. In this example, the graph will first be used to determine that `Calibrate` may be run in parallel with the other two methods in its structure. Later, when `Task2` arrives, the updated graph can be used to find an appropriate starting time for `Set-Parameters` which still respects the deadline of `Send-Result`, which relies upon it.

An example precedence graph modeling the complete set of our running activities can be seen in Figure 8. Two types of precedence interrelationships are shown, static and dynamic. Static relations are those which can be derived directly from the structure. This includes constraints from interrelationships and QAFs. Dynamic relations are those which are deduced from the execution context, such as when methods are scheduled or what resources are available. This includes elements such as deadline or earliest start time constraints from commitments, and the effects of resource usage. For instance, note the deadline (2500) constraint on `Send-Result`, which causes deadline constraints to be propagated and recognized for `Track-Medium` and `Set-Parameters`. The RF resource usage, shown abstractly at the bottom of the graph, is generated using a resource modeler that is covered in greater detail in the next section. The mutual relationships between those methods demonstrates the idea that regardless of their actual ordering, the methods still maintain a resource precondition with one another. Additionally, such resource-based constraints are only discovered as part of the scheduling process, so some interactions may not be found until the scheduler actually searches the space of solutions where they are manifested. In this example, the constraint over the RF resource between `Send-Message` and `Send-Result` has not

[2]Note that a method's expected duration does not imply use of a (bounded) CPU during that time. Early implementations of TÆMS assumed that a method's duration implied the complete use of the local processor during that period. We now allow methods which are otherwise independent to run concurrently. The original, single-processor behavior can be modeled through the use of a shared processor resource, which is reasoned about like any other resource.




A) Initialization task structure. B) Tracking goal negotiation task structure.

Figure 7: Two TÆMS task structures, abstractions of those used in our agents.

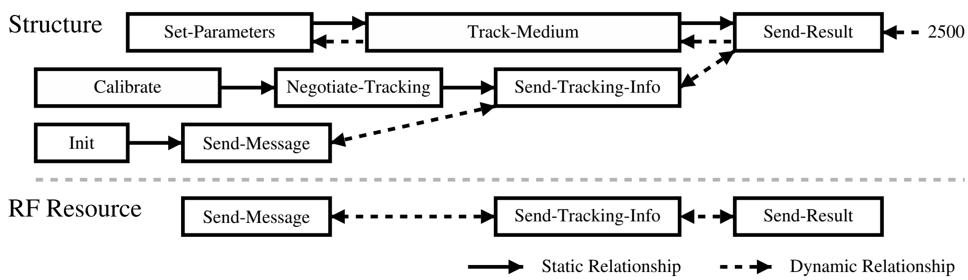



Figure 8: A partially ordered precedence graph modeling the running example on the task structures shown in Figures 4 and 7. Both static (i.e. those derived from interrelationships, QAFs, etc.) and dynamic (from resources, deadlines, etc.) constraints are shown.

been found for this reason.

From the partial order scheduler's point of view, this model is used to detect and cache resource relationships that can be used to facilitate the scheduling process just like any other ordering relationship. In such cases, a relationship can be detected when an attempt to schedule concurrent resource usages fails. This relationship can then be used in the future to reduce the resource usage search space which must be explored.

While the partial order scheduler may directly reason about direct precedence rules as outlined above, a more robust analysis is needed to identify indirect interactions which occur through common resource usage. Because of uncertain and probabilistic interactions between resources and actions, both locally and those to be performed by other agents, a thorough temporal model is needed to correctly determine acceptable times and limits for resource usage.

### 3.4 Resource Modeler

The responsibility of binding (conceptually allocating) resources to specific activities, belongs to a separate component called the resource modeler. The partial order scheduler does this by first producing a description of how a given method is expected to use resources, if at all. This description includes such things as the length of the usage, the quantity that will be consumed or produced, and whether or not the usage will be done throughout the method's execution or just at its start or completion. The scheduler then gives this description to the resource modeler, along with constraints on the method's start and finish time, and asks it to find a point in time when the necessary resources are available.

As with most elements in TÆMS the resource usage is probabilistically described, so the scheduler must also provide a minimum desired chance of success to the modeler. At any potential insertion point, the modeler computes the aggregate affects of the new resource usage, along with all prior usages up to the last known actual value of the resource. The expected usage for a given time slot can become quite uncertain, as the probabilistic usages are carried through from each prior slot. If the probability of success for this aggregate usage lies above the range specified by the scheduler, then the resource modeler assumes the usage is viable at that point. Since a given usage may actually take place over a range of time, this check is performed for all other points in that range as well. If all points meet the success requirement, the resource modeler will return the valid point in time. After this, the scheduler will insert the usage into the model, which will then be taken into account in subsequent searches. If a particular point in time is found to be incompatible, the resource modeler continues its search by looking at the next "interesting" point on its timeline - the next point at which a resource modification event occurs. The search process becomes much more efficient by moving directly from one potential time point to the next, instead of checking all points in between, making the search-time scale with the number of usage events rather than the span of time which they cover. Caching of prior results, especially the results of the aggregate usage computation, is also used to speed up the search process.

Consider our running example involving the three tasks shown in Figures 4 and 7. `Task1` arrives first, followed by `Task2` around time 260, and `Task3` is recognized at time 750. Each of these tasks includes methods which make use of the `RF` resource, using the pseudo-locking scheme described earlier, and as such their respective execution times will interact with one another. We will consider first the simple case where each affected method will consume all of the `RF` resource when it starts, and produces that same amount when it completes, with a probability of 1. The start times and finish times will be deterministic single values. In this case, the first scheduled method, `Send-Message`, will consume the `RF` resource at time 260, the `Send-Tracking-Info` will consume it at 1375, and `Send-Result` will continue that same level of consumption starting from 2000 until 2500. The combined resource model for this usage pattern can be seen in Figure 9A. This graph shows what the probability of a given resource level will be at a given time. For instance, at time 2000, there is a probability of 100% that the resource will be at level 0, while a 0% probability it will be at either +1000 or -1000. This is consistent with our description, which stated that while the method is running, the resource should be completely consumed.

A more interesting usage pattern will occur when uncertainty is introduced into the schedule. Consider the same set of three methods, with the same level of `RF` consumption. The difference in this case is that each method will have both an uncertain start time and uncertain duration, so that where the start time for `Send-Message` was [260, 1.0], it will now be [210, 0.25, 260, 0.5, 310, 0.5]; a 25% chance it will start early at time 210, a 50% chance of starting at the correct start time of 260, and a 25% chance it will start late at 310. Their durations are modified similarly. Figure 9B shows what the complete `RF` resource model would look like after this modified `Send-Message` usage is added. Note how the expected resource levels at the beginning are less precise, ranging from time 210 to 310, and how this uncertainty in combination with the modified duration produces an even less certain finish time, ranging from 660 to 860. A similar pattern is seen later in the time line when the usage from `Send-Result` is added in Figure 9C. When the usage from `Send-Tracking-Info` is added, however, something different occurs. The interaction between the uncertain finish time of `Send-Tracking-Info` and the start time of `Send-Result` results in a non-zero probability that the resource level might exceed its lower bound. Specifically, Figure 9D shows that around time 2000 there is a 24% chance that the level of `RF` resource might reach -1000, which is below its lower bound of zero, and would cause one of the methods to fail. This situation can be avoided by shifting one or both of the methods' execution times, so given this condition, the scheduling component could use this information to reason about the tradeoff between success probability and any temporal constraints associated with the activities.

The type of information shown in Figure 9 is used by the resource modeler to search for appropriate places where new resource usages may be inserted. In general, the scheduling process will provide a set of resource usage descriptions extracted from methods it is attempting to schedule, which may affect multiple different resources at different times, along with start and finish time bounds and a minimum desired probability of success, and the resource modeler will return the first possible match if one is found. These constraints are then used along with direct structural precedence rules and the existing schedule to lay down a final schedule.

### 3.5 Schedule Merging

Once potential interactions, through interrelationships, deadlines or resource uses are determined, the partial order scheduler can evaluate what the best order of execution is. Wherever possible, actions are parallelized to maximize the flexibility of the agent, as was introduced in section 3.3. In such cases, methods running concurrently require less overall time for completion, and thus offer more time to satisfy existing deadlines or take on new commitments. Once the desired schedule ordering is determined, the new schedule must be integrated with the existing set of actions. The areas of parallelism found during this stage are cached, along with a description of the current resource context. If a similar task structure is instantiated later under a similar resource context, these parallelism hints are used to mark up the task structure before it is sent to DTC for planning. These modifications allow DTC to more accurately reason about the time a series of actions will actually take when they are performed (in parallel), potentially allowing it to choose a cheaper or higher quality plan. This technique will be covered in more detail in section 4.3.

The partial order scheduler makes use of two other technologies to integrate the new goal with existing scheduled tasks. The first is a conflict resolution module, which determines how best to handle un-schedulable conflicts, given the information at hand. Most time-constrained tasks in the agent are added through negotiation with other agents, which will have assigned an importance value to their particular commitment. This value remains associated with the task structure and scheduled methods as they are created. Thus, when scheduling conflicts arise, the conflict resolution component can compare the relative importance of the conflicting actions, and remove the one of lesser priority. If such a decommittal is made, or if no valid resolution can be found, the problem solving component is notified of the situation so that it can take appropriate action. We are also investigating the feasibility of "just-in-time" schedule modifications. This technique would attempt to use the original task structure to quickly replace an existing infeasible sequence of actions with an alternative that can satisfy the commitment in the current context. A second component handles the job of merging the new goal's schedule with those of prior goals. The specific mechanism used is identical to that which determines order of execution. Interdependencies between this large set of methods, either direct or indirect, are used to determine which actions can be performed relative to one another. This information is then used to determine the final desired order of execution.

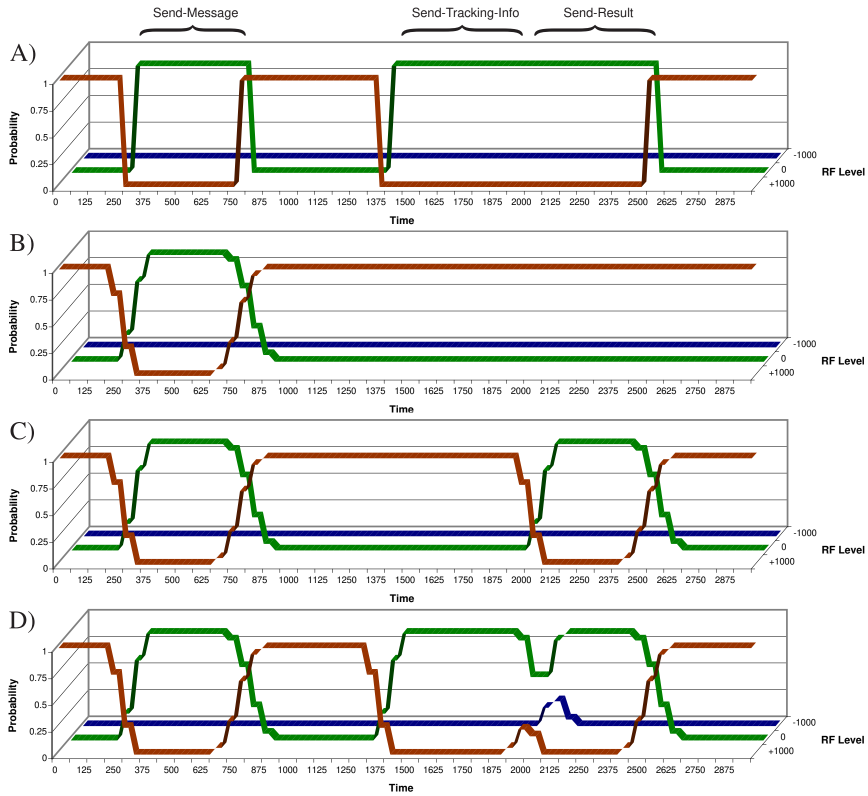



Figure 9: Resource model states after various usage patterns have been added. Each series represents the probability over time of a particular RF resource level occurring (1000, 0, or -1000). A) shows the model after all the resource uses from the simple, certain schedule are added. B-D) show the model after uses from each of the three methods in the uncertain schedule are added.

To this point in our example, the agent has been asked to work towards three different goals, each with slightly different execution needs. `Task1` allows some measure of parallelism within itself, as `Init` and `Calibrate` can run concurrently because no ordering constraints exist between them. `Task2`, received some time later, must be run sequentially, and its method `Send-Result` must be completed by time 2500. `Task3` is received later still, and also must be run sequentially. All three, however, require the use of the `RF` resource, for communication needs, and are thus indirectly dependent on one another. The partial order scheduler produces the schedule seen in Figure 10A, where all the known constraints are met. Some measure of parallelism can be achieved, seen with `Set-Parameters` and `Send-Message`, and also between `Track-Medium` and the methods in `Task3`. Note that the resource modeler detected the incompatibility between the methods using `RF` (shaded gray), however, and therefore do not overlap.

### 3.6 Method Execution

The execution details of particular methods specified in the TÆMS structure will vary from one environment to the next. In all cases, however, they are initiated and monitored by an execution component resident in SRTA. During each cycle, the component will analyze the current schedule(s) to find a group of candidate actions which may be started by comparing their scheduled start times with the current time. Each member of this group is then checked to see if its preconditions have been met. This includes verifying the success of enabling tasks, determining sufficient resources are available, and meeting earliest start time criteria. Any methods which fail a precondition are delayed, using the scheduler's shifting mechanism described earlier. Methods which meet their preconditions are started. In subsequent cycles, the execution component will continue its operation by comparing the observed performance of the actions against their expectations as modeled in the schedule. Any differences indicate a place where the schedule is no longer accurate, so the schedule is maintained as time progresses to reflect these changes. A common problem is that an action may take longer than expected, in these cases the execution component will use the partial order scheduler to update the schedule to reflect the new duration, again shifting dependent methods as necessary. This process will be covered in more detail in the next section. Finally, when a method completes, the execution characteristics are recorded in the TÆMS structure, and also propagated as an event to the rest of the agent.

How methods are actually performed is a relatively domain-dependent issue. The execution component as-

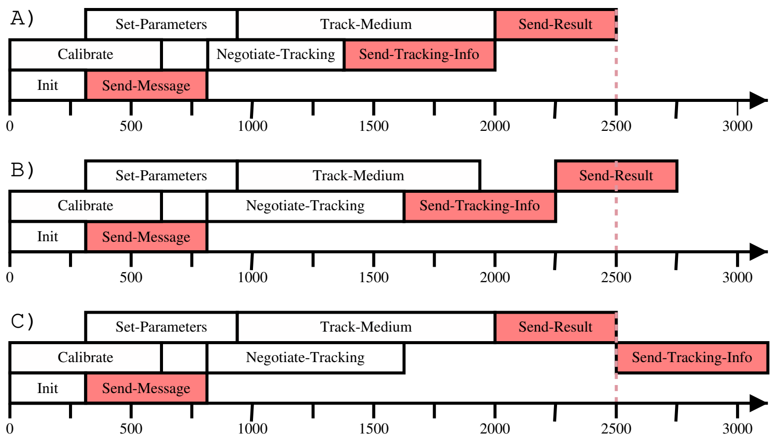



Figure 10: A) Initial schedule produced after all the goals have been received, with a `Send-Result` deadline of 2500, B) the invalid schedule showing that constraint broken by the unexpected long duration of `Negotiate-Tracking`, and C) the corrected schedule respecting the deadline.

sumes that methods are asynchronous (i.e. that control returns immediately after the action is initiated), and can potentially be run in parallel. The component itself is responsible only for determining when the action should start and tracking its progress. In other respects, however, the specific form the action takes is up to the system designer. For example, when running within the MASS simulation environment[32], actions are sent to the simulator to be performed. MASS uses the quantitative characteristics of the TÆMS structure itself (or some variant of it) to determine what the appropriate execution characteristics are. In real-world systems, the action is performed locally. The process begins when the execution component fires an event indicating that an action has started. Elsewhere in the agent, domain-specific code would respond to that event by actually performing the action. This might be accomplished by spawning a new thread, creating a separate process, or operating intermittently in response to the agent's own execution cycle. In the resource allocation domain we have been using, sensor actions (e.g. measurements, changes to various settings) are relayed to the sensor itself, which performs the operation asynchronously. Other actions, such as message sending or data fusion, are performed directly by the agent. It is important to note that the mechanism employed, and how it interacts with the underlying operating system and competing tasks on the local processor can ultimately affect the execution characteristics of the action itself. Like any other aspect which can affect performance, the design and qualitative elements of the TÆMS model should reflect the variance created by these interactions.

### 3.7 Conflict Resolution

Suppose next that `Negotiate-Tracking` is taking longer than expected, forcing the agent to dynamically reschedule its actions. Because the method `Send-Tracking-Info` cannot start before the completion of `Negotiate-Tracking`, due to the *enables* interrelationship shown in Figure 7B, the partial order scheduler must delay the start of `Send-Tracking-Info`. A naive approach would simply delay `Send-Tracking-Info` by a corresponding amount. This has the undesirable consequence of also delaying `Send-Result`, because of the contention over the `RF` resource. This will cause `Send-Result` to miss its deadline of 2500, as shown in the invalid schedule seen in Figure 10B.

Fortunately, the partial order scheduler was able to detect this failure, because of the propagation of execution windows. `Send-Result` was marked with a latest start time of 2000. This caused the partial order schedule to try other permutations of methods, which resulted in the schedule shown in Figure 10C, which delays `Send-Tracking-Info` in favor of `Send-Result`. This allows the agent to proceed successfully despite a failed expectation. This process is accomplished by first delaying the finish time of the offending method in the schedule to reflect the current state of affairs, and then recursively delaying any other methods which are dependent on that method until a valid solution is found or a recursive limit is reached. At each step, the schedule generation is performed in the same way the initial schedule was generated, i.e. through analysis of the precedence graph and resource usage patterns, Note that this is a "saticificing" process, which will attempt to find the best solution, but only guarantees that the minimum set of criteria are met. Not all permutations are explored, thus an acceptable solution may exist but not be found.

This type of simple conflict resolution is performed automatically, through the cooperation of the execution module, which detects the unexpected behavior, and the scheduling component which attempts to repair the problem using the quick shifting technique shown above. In some cases, in particular when methods actually fail to achieve their goal, this sort of simple shifting is not sufficient to repair the problem. To handle these cases, we have developed a conflict resolution module capable of analyzing a particular situation and suggesting solutions.

Abstractly, the conflict resolution module is a customizable engine, which applies techniques encoded as "plug-ins" to a particular situation. If the set of techniques available is not appropriate for the agent designer, they are free to add or remove techniques as needed. Each technique plug-in is associated with a discrete numeric priority rating, typically specified by the designer of the plug-in, which controls the ordering in which the techniques are applied. When searching for a conflict resolution, the engine will begin by applying all techniques marked with the highest priority. If one or more solutions are suggested, then that set of solutions is returned for the caller to select from. If no solutions are suggested, the engine will apply the techniques at the second-to-highest level, and so on. If the designer orders the techniques appropriately, for instance with quick or highly effective techniques first followed by slower or less applicable ones, the engine should make efficient use of its time. The ordering of these plug-ins is currently more art than science, so a certain amount of domain knowledge and experience is required on the designer's part. The engine and resolution techniques described below exist in prototype form, and have not been extensively tested.

Several different types of conflict resolution techniques have been implemented. For instance, when DTC generates plans for a task structure, it automatically produces a range of plans, from which the best rated is generally selected for use. A simple and effective resolution tactic is then to select the next best-rated schedule after the first which is not affected by the failure, and use that in place of a failed one. Another plug-in implements a variant of this technique which allows the modeler to actually pre-specify an alternate plan, which is then read in and used similarly. This latter technique is somewhat more costly, as it involves file access and data parsing, so it would likely have a lower priority than the first.

If no viable alternate plans are available, the entire structure can be sent back to DTC for re-planning. Because the structure would incorporate new information about the current context (in particular, the conflict or failure would be modeled), the plans DTC would return would bear this in mind. For instance, in the example above where `Negotiate-Tracking` took too much time, this updated duration information would be incorporated into the structure, which would cause DTC to ignore a subset of schedules which previously would have been valid. A more expensive version of this technique allows the TÆMS modeler to explicitly mark up methods such that they trigger a particular structural change when they fail. This could, for instance, swap a failed method A with alternate method or task A' which would presumably achieve the same results using a different method. This structure would also be sent to DTC for re-planning. Finally, because DTC is somewhat detached from the global planning process, in that it typically plans for only one of many possible concurrent goals at a time, a resolution technique might be to artificially restrict the desired level of quality, cost or duration exhibited by the plans it produces. In this way, constraints imposed by goals which are unrelated, except that they are owned by the same agent, may be abstractly represented in the structure as a limited resource, restrictive criteria, or other artificial bound. This may result in a different and hopefully more applicable set of plans being generated. For instance, if all plans which DTC generates for a particular goal are incompatible with the existing schedule because they require more time than is available, one might limit the desired solution quality to cause DTC to return schedules which it otherwise would have dropped. In another case, if one goal had a precedence constraint with another that is currently running, an artificial deadline or earliest start time could be added to the new goal to allow DTC to correctly reason about the interaction without actually possessing direct knowledge of it.

A final efficient way of resolution is to use the knowledge gained from prior resolved conflicts and cache it for later use. In this case, the plug-in will monitor both the resolution strategies which are selected to be applied, and the context in which they were chosen. Later, when the same context is seen, the earlier solution can be immediately suggested. If this plug-in is given a high priority, then a potentially expensive search process may be avoided with no detrimental effects.

As an example, consider the TÆMS structure shown in Figure 4. We will assume three different resolution plug-ins are in use by the agent, corresponding to several of the techniques outlined above. At the highest priority level is Check-Cache, which searches for cached resolution techniques which are applicable to the current problem. At the next level is Alternate-Plan, which looks for compatible results from the previous scheduling activity. At the lowest priority level is Regenerate-Plans, which uses DTC to generate a completely new set of viable plans. The initial schedule generated from this structure would be { `Set-Parameters`, `Track-High`, `Send-Results` }. In this instance however, `Track-High` fails, forcing the conflict resolution subsystem to find an appropriate solution. Check-Cache has never seen this problem and context before, so it offers no solution. The prior planning activity, however, returned three different plans, so two potentially viable plans remain for Alternate-Plan to examine. In this case, the plan { `Set-Parameters`, `Track-Low`, `Send-Results` } both avoids the failed method and still fulfills related commitment criteria. This schedule is offered as a solution. Since a solution was offered at a lower level, Regenerate-Plans is not invoked. Because only one solution is provided, the execution subsystem will instantiate the Alternate-Plan solution. If multiple solutions were provided, they would be discriminated through their respective expected qualities (which can be obtained from the task structure).

Note that if this problem were seen again, Check-Cache would recognize the context and provide this same solution, avoiding further search through the available resolution plug-ins.

## 4 Optimizations

The high-level technologies discussed above address the fundamental issues needed to run in real-time. Unfortunately, even the best framework will fail to work in practice if it does not obtain the processor time needed to operate, or if activity expectations are repeatedly not met. A good example of this is the execution subsystem. It may be that planning and scheduling have successfully completed, and determined that a particular method must run at a particular time in order to meet its deadline. If, however, some other aspect of the agent has control of the processor when the assigned start time arrives, the method will not be started on time and may therefore fail to meet its deadline. In this section we will several techniques which aim to reduce the overhead of different aspects of the system, in an attempt to avoid such situations.

### 4.1 Meta-Level Accounting

Several issues cause this problem described above. Of primary concern in this example is the fact that the agent is not accounting for and scheduling all the activities the agent is performing. Many systems only schedule for the low-level tasks they perform - the actions which directly and tangibly affect the goal at hand. At the same time, however, there is an entire class of actions which the agent is performing, and therefore compete for the same processing time, which are not accounted for. Such tasks include many elements seen in figure 2: communication, negotiation, problem solving, planning, scheduling and the like. These so-called "meta-level" activities can constitute a significant fraction of the agent's running time, but are not being directly scheduled.

In this research we have added meta-level accounting of communication and negotiation. Although not strictly a feature of the architecture itself, it is still a sufficiently important issue to merit discussion in SRTA's context. Reasoning over meta-level costs was accomplished by first modeling these activities using TÆMS task structures. From a planning and scheduling point of view, there is no difference between low and meta-level actions, so to account for this time we need just an accurate model and a component capable performing these actions in response to a method execution. Given this, we can use our existing TÆMS processing components to correctly account for this time. The task structure from our running example, seen in Figure 7B, models both negotiation and communication activities. The duration of a negotiation task is relatively deterministic, or at least can be described within some bounds, so creating the task structures was a matter of learning the characteristics of our negotiation scheme. An additional benefit of describing these activities in TÆMS is that it permits the planning component to reason about the selection of negotiation schemes. Consider a system where one had several different ways to negotiate over a particular commitment, each with different quality, cost and duration expectations. By describing these in TÆMS , we can simply pass the structure the generic DTC planning component, which will determine the most appropriate negotiation scheme for the current environmental conditions. Furthermore, once a given scheme is selected, it may also be parallelized by the partial order scheduler for greater efficiency.

In future research we hope to model other meta-level activities, such as scheduling and planning[28]. These topics are more complicated due to their non-deterministic nature, i.e. the agent does not necessarily know a priori how long it will take to schedule an arbitrary set of interdependent actions nor precisely when this activity will be needed. In addition, the need to quickly schedule and plan in the face of unanticipated events, and the potential need to schedule the scheduling of activities itself makes these processes particularly difficult to account for. We currently handle the time for these activities implicitly by adding slack time to each schedule. This is accomplished by reasoning with the maximum expected duration time for a given schedule, rather than the average time. This simple approach works when the variance of the schedule's duration is not particularly wide; different characteristics might require a more suitable approximation. In the future, we plan to revise this approach with a more intelligent accounting of time in the schedule.

### 4.2 Plan Caching

An issue affecting the agent's real time performance is the significant time that meta-level tasks such as planning and scheduling can take themselves. In systems which run outside of real-time, the duration performance of a particular component will generally not affect the success or failure of the system as a whole - at worst it will make it slow. In real time, this slowdown can be critical, for the reasons cited previously. Complicating this issue is the fact that these meta-level activities may be randomly interspersed with method executions. New goals, commitments and negotiation sessions may occur at any time during the agent's lifetime, and each of these will require some amount of meta-level attention from the agent in a timely manner. To address this, our control architecture attempts to optimize the meta-level activities performed by the agent.

One particular computationally expensive process for our agents is planning, primarily because the DTC planner runs as a separate process, and requires a pair of disk accesses to use. Unfortunately, this is an artifact caused by DTC's C++ implementation; the remainder of the architecture is in Java. We noticed during our scenarios that a large percentage of the task structures sent to DTC were similar, often differing in only their start times and deadlines, and resulting in very similar plan selections. This is made possible by the fact that DTC is now used on only one goal at a time, as opposed to our previous systems which manipulated structures combining all current goals. To avoid this overhead, a plan caching system was implemented, shown as a bypass flow in Figure 2. Each task structure to be sent to DTC is used to generate a key, incorporating several distinguishing characteristics of the structure. If this key does not match one in the cache, the structure is set to DTC, and the resulting plan read in, and added to the cache. If the key does match one seen before, the plan is simply retrieved from the cache, updated to reflect any timing differences between the two structures (such as expected start times), and returned back to the caller. This has resulted in a significant performance improvement in our agents, which leaves more time for low-level activities, and thus increases the likelihood that a given deadline or constraint will be satisfied. Quantitative effects of the caching system can be seen in Table 2.

To test the caching subsystem, we performed 1077 runs using eight sensors and one target in the RADSIM environment[3], which models the distributed sensor environment discussed in this paper. As shown in the table, the caching system in these tests was able to avoid calling DTC 30% of the time, resulting in a significant savings in both time and computational power.

### 4.3 Parallel Activity Recognition

The major disparity which exists between the DTC planning component and the remainder of the SRTA architecture is its inability to plan for parallel activities[4]. It assumes a sequential set of actions, and generates plans accordingly. Under constrained conditions, this can lead DTC to eliminate potential candidate plans which would otherwise have functioned successfully if their innate parallelism were recognized and exploited.

One way to solve this problem would be to update DTC's logic to directly reason about these types of interactions. After consideration, it was determined that the amount of effort needed to do this would outweigh the benefits. In addition, there are certain advantages in reduced complexity which arise from a layered system, where some aspects of the problem space are hidden at different levels. Instead, we have worked around this issue through the use of a mapping function, which is able to translate some classes of parallelism into an analogous form in TÆMS which DTC is able to correctly reason about.

The process is best explained through an example. Consider the abstract task structure shown in Figure 11A. This structure has two subtasks, `Sub1` and `Sub2`, which must both be performed successfully and in order, because of the *enables* interrelationship between them and `Task1`'s *min* QAF. Note that `Sub1` also has a *min* QAF, so that `A` and `B` must both be performed, while `Sub2` has a *max*, requiring either or both of `C1` and `C2`. The method `C2` both requires more time to complete and has a higher expected quality than `C1`. Finally, the entire structure has a deadline which must be respected.

The initial DTC plan is shown in 11A which will then be used by the partial order scheduler to generate the schedule below it. Note that because `A` and `B` could be performed in parallel, the initial plan does not make efficient use of the available time. In fact, the deadline caused DTC to select `C1` over the higher quality `C2`, which would otherwise have had sufficient time to complete in the final schedule. To compensate for this, schedules are analyzed for these areas of parallelism. If any are found, that information is used to annotate subsequent TÆMS structures that are structurally identical before they are sent to DTC. Such a structure is shown in 11B, where a pair of mutual *facilitates* relations have been added between methods `A` and `B`. These interrelationships are quantified in such a way that if either method is performed, the model indicates that the remaining method will take zero time. This will be interpreted by DTC as meaning, for instance, that `B` may be performed instantaneously once `A` has completed, which has roughly the same characteristics as a true parallel schedule. Because of this, more time will be available within the plan, and the higher quality `C2` method will be selected as shown. This will result in the higher quality schedule as shown.

The notion of parallel activity recognition is one aspect of a more general problem where there exists a class of conditions which a subsystem is unable to detect or exploit due to its lack of context or functionality. In SRTA, because DTC may be used to plan for structures without complete knowledge of potentially competing local schedules, it can produce plans which are unacceptable. To compensate for this, at runtime one can condition the task structures given to the subsystem to compensate for this lack of information. For example, to model the effects of a concurrent process, DTC might be asked to generate plans with artificially limited durations by modifying the planning criteria shown in Figure 5. Similarly, if it is known that a resource will be restricted in the future, one might present DTC with a more tightly bounded view of that resource to avoid possible conflicts. More generally, we can address this class of issues by first learning or anticipating that such conditions will exist, and then augmenting the information used by the subsystem to provide a suitable abstraction of the otherwise unobservable constraint.

### 4.4 Learning

Much of the material discussed in previous sections assumes that the TÆMS models describing our activities are faithful to real world performance. It should be clear that without accurate models, it will be quite difficult for the agent to correctly allocate its time. In prior research, [17] some quantitative and structural elements of TÆMS structures have been shown to be learnable using off-line analysis of a large corpus of results. While this technique would work to a certain extent for our application, we are more interested in using a lightweight runtime learning component to give the agent the capability to dynamically adapt to changing conditions.

Our current learning system automatically monitors all method executions in the agent, and maintains a set of the last $n$ results. When queried, the component uses these results to compute a duration distribution for the particular method in question. This data can then be used to condition new task structures, improving their predictive accuracy and along with the agent's scheduling success.

A more ambitious goal that we hope to address in future

[3] RADSIM was designed and built by the Air Force Research Laboratory under the direction of Jamie Lawton (`lawton@rl.af.mil`).

[4] To be more precise, DTC does support a particular kind of parallelism associated with a method's percentage of processor usage, but it is not sufficiently general for SRTA's needs[36].

| Component | Average Number of Calls | Execution Time |
|---|---|---|
| DTC Scheduler | 72.14 | 300 ms |
| DTC Caching | 31.12 | 74 ms |
| Partial Order Scheduler | 531.03 | 36 ms |

Table 2: Average results over 1077 runs of 180 seconds each.

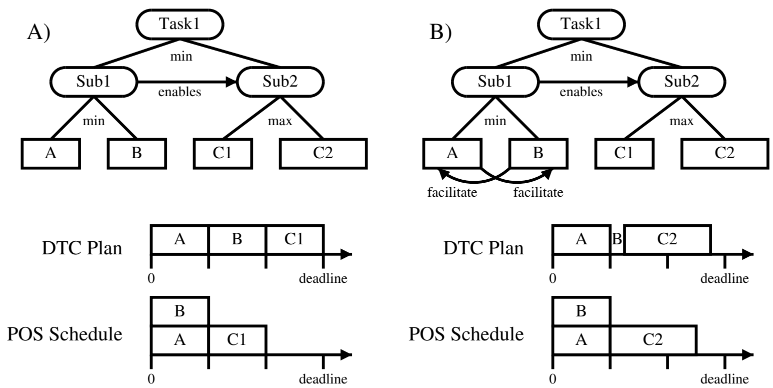



Figure 11: The effects of parallel activity recognition. A) shows the original task structure, which leads to an inefficient final schedule, B) shows the task structure with modifications, which results in a higher quality schedule.

work is the ability to learn how much time and resources the meta-level activities associated with a goal require, and how to better predict and account for interactions between local activities. This metric could then be used to augment or annotate the goal's structure or modify the objective criteria in such a way that the agent is able to reason about those requirements. Consider a situation where the agent uses the resource modeler and the current schedule to compare the availability of resources and time in the current context to the agent's ability to successfully complete a particular plan or schedule. If a correlation is able to be drawn from such observations, future planning instances in similar contexts could implement a change the criteria to avoid potential pitfalls. In such cases, by varying the desired quality, duration or cost in the criteria provided to DTC, more appropriate plans can be produced. Examples of such changes are covered in further detail in Section 5.3.

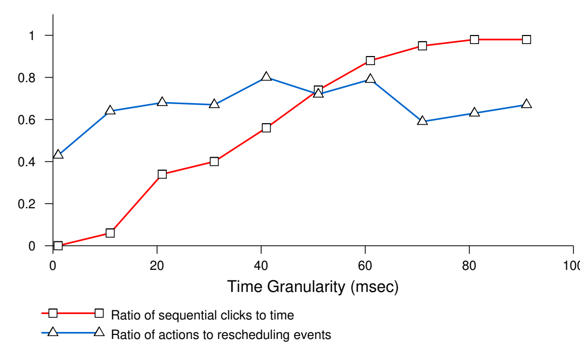



Figure 12: The effect of varying time granularity on agent behavior. A higher time ratio indicates that a greater percentage of sequential time units are seen, which should reduce the need for rescheduling. A higher action ratio indicates the available time was used more efficiently.

## 4.5 Time Granularity

The standard time granularity of agents running in our example environment is one millisecond, which dictates the scale of timestamps, execution statistics and commitments. Because we run in a conventional (i.e. not real-time) operating system, in addition to our relatively unpredictable activity durations, it becomes almost impossible to perform a given activity at precisely its scheduled time. For instance, some action $X$ may be scheduled for time 1200. When the agent first checks its schedule for actions to perform, it is time 1184. In the subsequent cycle, 24 milliseconds have passed, and it is now time 1208. To maintain schedule integrity (especially with respect to predicted resource usage), we must shift or reschedule the method which missed its execution time before performing it. Despite our existing optimizations, although each of these events are individually quite fast, combined in large numbers they can consume a significant portion of the agents' operating time.

To compensate for this, we scale the agents' time granularity by some fixed amount. This theoretically trades off prediction and scheduling accuracy for responsiveness [6, 7], but in practice a suitably chosen value has few drawbacks, because the agent is effectively already operating at a lower granularity due to the real time missed between agent activity cycles. Using this scheme, if we say that every agent tick corresponds to 20 milliseconds, the above action would be mapped to run at time 60. At

time 1184, the agent would operate as if it were time
while 1208 would become 60, the correct scheduled ti
for $X$, thus avoiding the need to shift the action. Clea
we can not eliminate the need for rescheduling, due
the inherent uncertainty in action duration in this en
ronment, but the hope is to reduce the frequency it
needed. Experimentation can find the most appropri
scaling factor for an agent running on a particular syste
by searching for the granularity which optimizes the nu
ber of actions which are able to be performed against t
number of rescheduling events which must take place. O
experiments, the results of which can be seen in Figu
12, resulted in a 35% reduction in the number of shifted or rescheduled activities by using a time granularity between 40 and 60 ms. Ideally, the system should "see" each sequential time click, but as the graph shows, as the system reaches that point, the coarse timeline unnecessarily restricts the number of actions which may take place, reducing the overall efficiency.

## 5 Execution Characteristics

This section will provide more concrete evidence showing how the SRTA architecture performs in practice. We will begin by showing how SRTA can be used to support the sort of sophisticated problem solving behavior which was described in the introduction. We will then show how commitments and constraints can be used to control method execution at runtime. Finally, we will show how alternative plans can be used to encode and support adaptive behavior, dependent on the current runtime context.

### 5.1 Supporting High-Level Reasoning

SRTA supports the problem solving aspects of a sophisticated agent through its capability of responding to "what-if" style queries, using the TÆMS language as the descriptive medium. Consider the case where one agent is attempting to coordinate with another. In this situation, the agent must first determine the goals it is capable of achieving which will satisfy the coordination request. Next, it must determine the constraints under which the coordination is being requested, such as deadlines or earliest start times. These two features are provided to SRTA, which takes into account the current activity schedule, environmental context and existing commitments during its analysis. SRTA will then both determine if that goal may be achieved, and if so, what the resulting execution schedule will look like if the needed activities were integrated with the existing schedule. If no solution is found, the reasoning component may decline the commitment or adjust the goal structure or constraints. If a solution is found, it may use the resulting schedule to provide the remote agent with expected completion characteristics.

Using TÆMS , the agent will first model the goal and its subtasks, along with any constraints that exist. Consider the schedule shown in Figure 14A. In this scenario, the agent has previously scheduled two goals, `Setup-Sensor` and `Perform-Track`, as modeled in Figure 13. The three `Track` methods in that model each have an expected level

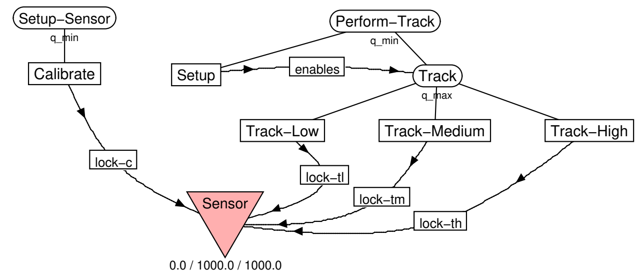



Figure 13: A pair of abbreviated task structures for calibration and tracking.

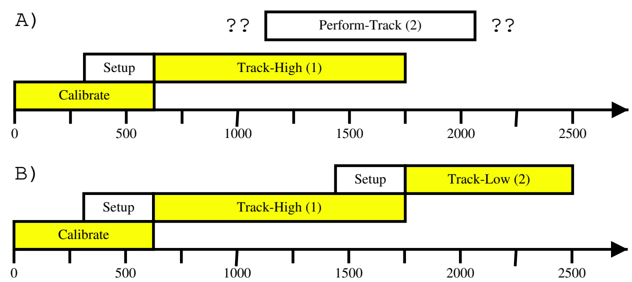
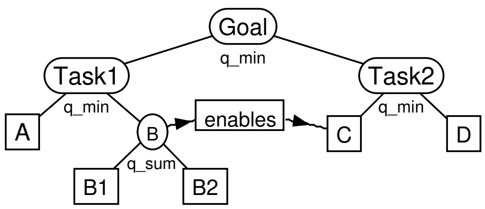


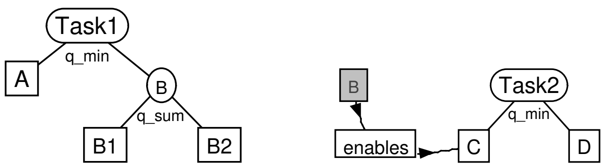

Figure 14: A) The agent's initial schedule, along with new actions which describe the what-if condition, been received, B) The resulting consolidated schedule.

of quality which corresponds to their duration (i.e. long duration → high quality). Because no competing methods existed, `Track-High` was selected for the `Perform-Track` goal. No direct interrelationships exist between the activities, but they do interact indirectly through the shared resource `Sensor`. In this case, both `Calibrate` and the three tracking methods use the `Sensor` resource to take measurements, and thus cannot be performed at the same time. This would be modeled using a similar locking mechanism to that used with the `RF` resource described earlier. Next, the agent is asked by another if it can satisfy the goal `Perform-Track` for it, within a deadline of 2500. To check this, the agent would pose a what-if query to SRTA with the appropriate task structure and the existing schedule, as shown in Figure 14A. Because of the deadline and the preexisting schedule, SRTA selects `Track-Low` to satisfy that goal, as shown in 14B. This result can then be used to support a commitment structure with the remote agent.

Note that SRTA did not suggest changing the preexisting method `Track-High` to a `Track-Medium`, which might have resulted in a more equitable arrangement where the second commitment could have also been accomplished with `Track-Medium`. SRTA is a satisficing architecture, to reduce both the combinatorics of planning and scheduling and the potential need for re-negotiation over existing commitments it does not optimize over all potential schedules and thus will not necessarily find the "perfect" solution. If it were the case that `Track-Low` was not a viable solution for this goal, it is expected that the reasoning component would have removed the method through a process of task structure conditioning prior to planning.




A) Global view of the task, including the *enables*, indicating there is a precedence relationship between B and C, which belong to different agents.




B) The runtime view of Task1.

C) The runtime view of Task2.

Figure 15: The abstract global view and segregated runtime structures showing a potential point where coordination would be needed.

Alternately, it could also validate the expected quality of the schedule after planning, and generate an appropriately modified what-if query if the initial result did not meet minimum criteria.

### 5.2 Modeling and Respecting Commitments

Commitments are an important class of structures because they allow an agent to formally define an agreement that it has with a remote party. These agreements, which can come in many forms in both cooperative and competitive systems, form a part of the foundation of multi-agent systems by adding structure to the actions that part of the system will take at the request of another. Because of this, SRTA provides facilities for both defining potential points where coordination may be necessary or fruitful, and mechanisms for defining and respecting commitments as they are needed.

Abstractly, a commitment is usually formed when the actions (or their results) of one agent may directly or indirectly affect the state of another. We have previously shown in section 3.1 how interrelationships between nodes in a single structure can model the effects between them. These types of effects may be simply extended to span nodes between structural elements belonging to disparate agents to model inter-agent effects. Refer to Figure 15A, which represents how a particular goal might be represented at the global, organizational level. In this case, `Task1` and `Task2` each belong to different agents, so that the *enables* relationship between them represents a point of interaction between them. More specifically, if the agent pursuing `Task2` is to be successful, it must ensure that the `Task1` is successfully completed before it begins `Task2`.

At runtime, it is unusual that any one agent would possess a complete global view as is shown in 15A. Instead, each agent would have its own local view of the problem, as is seen in 15B and C. In this case, we assume that `Task1` agent has no knowledge of the interrelationship. Instead, that information is represented in `Task2`'s task structure, which indicates that a *nonlocal* method `B` enables `C`. Thus, the agent working on `Task2` will recognize that `B` must come before `C`, `B` will be performed by a remote agent, and it must ensure that this condition is satisfied before proceeding. In a deliberative system, this would be accomplished through coordination between the two agents, where the coordination would result in a commitment specifying how and when `B` is to be completed. Note also that while `B` is represented as a method in `Task2`, it is actually just an abstraction which refers to a task subtree in `Task1`.

These interactions can occur at any point in the task structure where nodes can both affect one another and are the responsibility of different agents. For example, two nodes which are related through a common supertask might have this characteristic. Shared resources also provide an indirect point of coordination, where agents may need to coordinate there activities to ensure the resource is not over or under-loaded[23].

Once an agent has detected a point of remote interaction, it can then engage in a process of coordination. The specifics of such a process are beyond the scope of this article, more details can be found here [21, 5]. More germane is the concept that the agent must both determine what sort of commitment is necessary, and how that can be represented. The example in Figure 15 shows a situation where the success of one action depended on the successful completion of another. Thus a commitment is needed which lets the dependent agent know when that enabling activity will be completed. We refer to this as a *do* commitment. Conversely, if we replace the *enables* with a *disables* relationship, this would indicate that the dependent action would fail if the disabling action were successfully completed. In this case, the dependent agent would require a *dont* commitment, which indicated that the action would not performed within some window in time. The commitments themselves may be represented directly in TÆMS The structure allows one to define the commitment type, participating agents, relevant methods, relative importance, deadlines, earliest start times, and other relevant details. During planning and scheduling, these commitments are included with the TÆMS structure itself, enabling those components to use that data to influence their respective activities.

During commitment formation, the agent would use the what-if capability described in the previous section to determine if the commitment could be satisfied. Once it has been agreed upon the commitment is added to the agent's local structure, where SRTA may use it do drive local behavior. For example, return to the agents working on `Task1` and `Task2` above. In this case, `Task2` agent would initiate coordination with another agent capable of completing `B`. That remote agent would use the requested execution characteristics to create a commitment and pose a what-if query to SRTA seeking a candidate schedule.

In this case, depending on the temporal constraints, the schedule may include either or both of `B1` and `B2`, which will affect the solution quality the agent can offer. We assume they both agree on the proposed commitment, and `Task1` will then instantiate a task structure containing `B`. Meanwhile, `Task2` agent would also instantiate, plan and schedule its task structure. `Task2`'s activities would start, but immediately be suspended because the enablement from `B` is not active. This will continue until that enablement is activated (either by the local agent acting on the assumption `B` has completed, or from an explicit message from `Task2`), when that schedule will resume and complete.

Many of the details of coordination are left intentionally unspecified, to avoid restricting the designer to a particular class of interactions. SRTA instead tries to provide a suitably general set of modeling, analysis and execution primitives which can be used as a foundation for a range of different coordination alternatives.

### 5.3 Adapting to Environmental Conditions

An agent's ability to adapt to changing conditions is essential in an unpredictable environment. SRTA supports this notion with TÆMS , which provides a rich, quantitative language for modeling alternative plans, and DTC and the partial order scheduler, which can reason about those alternatives. As discussed previously, this combination can also make use of activity and resource constraints in addition to results of completed actions, providing the necessary context for analysis and decision making.

Consider the model shown in Figure 16, where a variety of strict and flexible options are encoded. Because `Goal` has a *seq_sum* QAF, it will succeed (e.g. accrue quality) if all of its subtasks are completed in sequence. The quality it does accrue will be the sum of the qualities of it's subtasks. The structure indicates that `D` must be performed for `Task2` to succeed, and also that the agent cannot execute `E` after `F`. `Task1` and `Task2` have slightly more flexible satisfaction criteria. Their *sum* QAFs specify that they will obtain more quality as more subtasks are successfully completed – without any ordering constraints. Finally, the *facilitates* relationships between `A`, `B` and `C` model how the agent can improve `C`'s performance through the successful prior completion of one or more of `A` or `B`. Specifically, `A` will augment `C`'s quality by 25%, while `B` will both increase `C`'s quality by 75% and reduce its cost by 50%.

There are several other classes of alternatives which are not shown in the figure. Resource interrelationships, for example, may be used to model a variety of effects on both the resources and the activities using them. The presence or absence of nonlocal activities, as discussed in the previous section, can indicate alternative means of accomplishing a task. Multiple outcomes on methods may indicate alternative solutions which may arise from a method's execution, so the probability densities associated with each outcome provide an additional source of discriminating information which can help control the uncertainty of generated plans. The individual probability distributions for the quality, cost and duration of each outcome serve in the same capacity, as do analogous probabilities modeling the quantitative effects of interrelationships. The available time, desired quality, and maximum cost, along with other execution constraints provide the context in which to generate and evaluate the alternative plans such a structure may produce.

To demonstrate how the system adapts to varying conditions, several plans derived from the task structure in Figure 16 are shown in Table 3. These plans are produced for different environmental conditions that place different resource constraints on the agent. As one would expect, when the agent is completely unconstrained and has a goal to maximize quality, the plan shown in row one is produced. Note that the selected plan has an expected quality of 49.9, expected cost of 7.0, and an expected duration of 90.0. The quality in this case is not a round integer even though the qualities shown in Figure 16 are integers because methods A and B *facilitate* method C and increase C's quality when they are performed before method C.[5]

Row two shows the plan selected for the agent if it has a hard deadline of 40 seconds. This is the path through the network with the shortest duration that enables the agent to perform each of the major subtasks. Note the difference in quality, cost, and duration between rows one and two.

Row three shows the plan selected for the agent if it is given a slightly more loose deadline of 50 seconds. This case illustrates an important property of scheduling and planning with TÆMS – optimal decisions made locally to a task do not combine to form decisions that are optimal across the task structure. In this case, the agent selected methods `ADEF`. If the agent were planning by simply choosing the best method at each node, it would select method `C` for the performance of Task 1 because `C` has the highest quality. It would then select `D` as there is no choice to be made with respect to method `D`. It would then select method `E` because that is the only method that would fit in the time remaining to the agent. The plan `CDE` has an expected quality of 25, cost of 10, and duration of 50. Scheduling and planning with TÆMS requires stronger techniques than simple hill climbing or local decision making. This same function holds when tasks span agents and the agents work to coordinate their activities, evaluate cross agent temporal constraints, and determine task value.

Row four shows the plan produced if the agent is given a hard deadline of 76 seconds. What is interesting about this choice is that DTC selected `BCDEF` over `ACDEF` even though method `B` has a lower quality than method `A` and they both require the same amount of time to perform. The reason for this is that `B`'s facilitation effect (75% quality multiplier) on method `C` is stronger than that of method A (which has a 25% quality multiplier). The net result is that `BCDEF` has a resultant expected quality of 43.5 whereas `ACDEF` has a resultant expected quality of

[5] Recall that facilitation models one process having a positive impact on another, e.g., producing a result that enables the other to do a better job or take less time to perform.

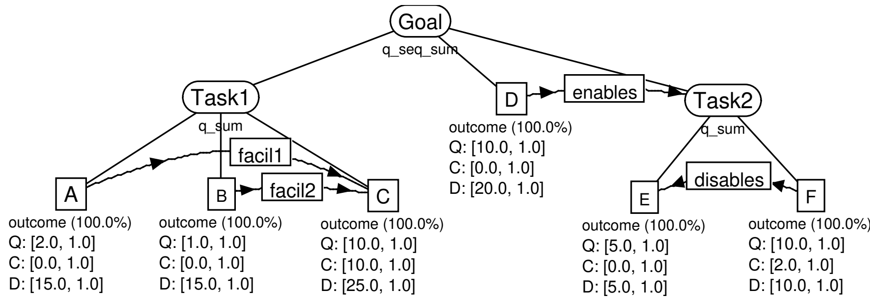



Figure 16: A TÆMS task structure modeling several different ways to achieve the same goal.

| | Conditions | Schedule | Q | C | D |
|---|---|---|---|---|---|
| 1 | Unconstrained | `A B C D E F` | 49.9 | 7.0 | 90.0 |
| 2 | Deadline 40 | `A D E` | 17.0 | 0.0 | 40.0 |
| 3 | Deadline 50 | `A D E F` | 27.0 | 2.0 | 50.0 |
| 4 | Deadline 76 | `B C D E F` | 43.5 | 7.0 | 75.0 |
| 5 | Cost 3 | `A B D E F` | 28.0 | 2.0 | 65.0 |
| 6 | Balanced | `A D E F` | 27.0 | 2.0 | 50.0 |

Table 3: A variety of schedules, and their expected qualities, costs and durations, generated from the TÆMS structure in Figure 16 under different conditions.

39.5.

Row five shows the plan produced by DTC if the agent has a soft preference for schedules whose cost is under three units. In this case, schedule `ABDEF` was selected over schedules like `ADEF` because it produces the most quality while staying under the cost threshold of three units. DTC does not, however, deal only in specific constraints. The "criteria" aspect of Design-to-Criteria scheduling also expresses relative preferences for quality, cost, duration, and quality certainty, cost certainty, and duration certainty. Row six shows the plan produced if the scheduler's function is to balance quality, cost, and duration. Consider the solution space represented by the other plans shown in Table 3 and compare the expected quality, cost, and duration attributes of the other rows to that of row six. Even though the solution space represented by the table is not a complete space, once can see where the solution in row six falls relative to the rest of the possible solutions – it is a good balance between maximizing quality while minimizing cost and duration.

These examples do not illustrate DTC's ability to trade-off certainty against quality, cost, and duration. The examples also omit the quality, cost, and duration distributions associated with each item that is scheduled/planned for and the distributions that represent the aggregate behavior of the schedule/plan. All computation in TÆMS and DTC is performed via discrete probability distributions. The role of uncertainty and its advantages are more completely documented in [38].

An additional example of adaptation taken from the distributed sensor domain is shown in Figure 17. The architecture we have developed to address this domain uses a notion of periodic commitments along a discrete timeline to reduce negotiation complexity. Specifically,

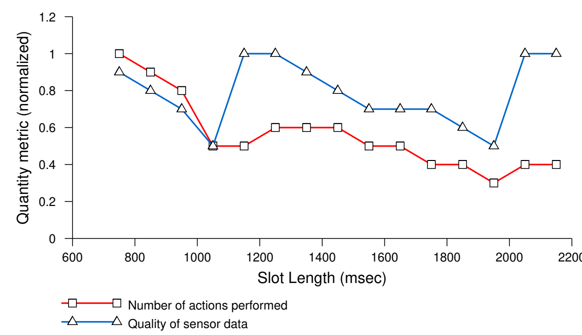



Figure 17: Monitoring track measurement quality under different temporal conditions.

the timeline is conceptually broken into a number of repeating periods, each of which is comprised by a set of equal-duration slots. Agents negotiate over these slots in such a way that a given commitment (which in our case represents a sensor measurement action which should take place) is satisfied by an action taken within one of these slots. Thus, the length of the slot represents a limiting factor, which will directly control the maximum duration of individual activities, and indirectly affect the total number of activities which may take place over time.

Figure 17 shows how our system adapts to varying this slot length. The task structure in Figure 4 is used, which provides the agent with three different types of tracking measurements to perform, each with a quality proportional to its duration. Thus, we would hope given these different alternatives, SRTA would adapt to increasing slot length by choosing higher quality measurements, offsetting the effect of a reduced number of total activities. The graph shows, as we would expect, that the total number of activities performed by the agent reduces as the slot length grows. It also shows that the aggregate quality of the measured sensor data followed more of a saw-tooth pattern. In this case, each jump in the pattern represents a slot length threshold which permitted the use of a higher quality, higher duration measurement activity. Following these jumps, the trend falls with a rate comparable to the number activities until it is able to schedule the next best measurement type. We can infer that further increasing the number of alternatives available to SRTA would lead to greater quality stability, by allowing it to more frequently "jump" to a more appropriate set of activities.

## 6 Related Work

It is important to note that the architecture presented here falls into the soft real-time computation class. In contrast to architectures like CIRCA [26], we cannot make performance guarantees [31] about agent control. However, in contrast to CIRCA, the approach presented here operates on multiple distributed agents and the statistically "fast enough" model addresses the requirements of this application. In addition, action primitives are permitted to have unpredictable performance results across several dimensions. In the future, hard real-time approaches for multiple distributed agents may be possible, but, currently, the complexity of the distributed agent control problem, particularly when agents have complex activities and are situated in dynamic and uncertain environments, prevents such approaches.

PRS [16] and the more recent work on UMPRS [18] both offer architectures capable of operating effectively in unpredictable domains. Like SRTA, PRS can use context to select from among alternative goal satisfaction plans, and its continuous reevaluation of these intentions allows it to be more responsive to unexpected events. This reactive nature prevents it from forming a complete end-to-end view of activity, so, unlike SRTA, future behavior cannot be predicted. PRS does offer blocking points, so synchronization messages can be used to facilitate more reactive coordination among agents [2].

Our work also relates to [41], which provides a scheme for selecting control policies in context through the use of progressive reasoning and opportunity cost. This technique, operating in an environment consisting of a set of tasks which may have uncertain qualities and duration, reactively chooses subsets of modules from a progressive processing unit in response to newly arrived goals. Each subset of modules is compared using a characterization of its expected execution performance, and the most appropriate plan chosen based on opportunity cost. Although using opportunity cost to discriminate among plans does implicitly consider their time-related interactions, SRTA is able to reason more directly over temporal constraints, such as deadlines and earliest start times, between both goals and the individual actions which are used to achieve a goal. SRTA also differs in its use of satisfying techniques for plan selection, and its ability to directly reason about task and method interactions, resource consumption and the external constraints needed to coordinate with other agents.

The DECAF framework [9] and associated DRU scheduler [8] are closely related to this work, the latter having been leveraged from SRTA's DTC scheduler. Like TÆMS , the DECAF language allows the designer to model tasks and actions, and includes notions similar to quality accumulation functions and enablement. It does not have support for the explicit modeling of resource interactions, and also does not have a notion of soft interrelationships. In general, this framework trades off the additional complexity seen in SRTA to achieve performance improvements through reduced combinatorics. In addition, the DRU scheduler uses a potentially more efficient, threaded execution process which can take advantage of multiple processor environments.

The partially ordered schedule representation used by SRTA is also similar to that used in [40], although that framework has a simpler model of resource interactions. In comparison to SRTA's satisficing technique, this framework also employs a more formal search process which will lead to an optimal schedule if one exists.

## 7 Conclusion and Future Directions

The SRTA architecture has been designed to facilitate the construction of sophisticated agents, working in soft-real time environments possessing complex interactions and a variety of ways to accomplish any given task. With TÆMS , it provides domain independent mechanisms to model and quantify such interactions and alternatives. DTC and the partial ordered scheduler reason about these models, using information from the resource modeler, current execution characteristics, and the runtime context to generate, rank and select from a range of candidate plans and schedules. An execution subsystem executes these actions, tracking performance and rescheduling or resolving conflicts where appropriate. The engine is capable of real-time responsiveness, allowing these techniques to be used to analyze and integrate solutions to dynamically occurring goals.

SRTA's objective is to provide domain independent functionality enabling the relatively quick and simple construction of agents and multi-agent systems capable of exhibiting complex and applicable behaviors. It's ability to adapt to different environments, respond to unexpected events, and manage resource and activity-based interactions allow it to operate successfully in a wide range of conditions. We feel this type of system can form a reusable foundation for agents working in real-world environments, allowing designers to focus their efforts on higher-level issues such as organization, negotiation and domain dependent problems.

More generally, the significance of the work presented in this paper comes from its demonstration that it is possible to perform in soft real-time the complex modeling, planning and scheduling that has been described in our prior research. Previously, these techniques were analyzed only in theory or simulation, and it was not clear that our heuristic approach would be sufficiently responsive and flexible to address real-world problems. The SRTA architecture shows that engineering can be used to combine and streamline these approaches to make a viable, coherent solution.

There are several technical directions that we think are important in developing this framework further. While the current architecture does work in soft real-time in the domain described in this paper, that is no guarantee it will do so in other domains with different problem characteristics and responsiveness constraints. Allowing individual components to operate in an anytime [43] or time-bounded fashion would allow the system's performance to be more predictable. DTC already provides this capability to a certain degree. An efficient meta-meta reasoning component would allow the agent to directly decide how much effort to allocate to the DTC component in the current situation[27]. Another role for this new component is to decide where and how much slack to put in the schedule to accommodate unexpected primitive and meta-level activities. As we discussed earlier in the paper, we do not explicitly allocate slack time for unexpected meta-level events such as planning and scheduling. Direct accounting for this time would better equip the agent to meet strict deadlines.

SRTA also is currently unable to provide a meaningful description in case of failure, which makes it unclear how to react in these situations. For example, when a schedule or plan cannot be found within the provided context, no feedback is available to help determine what aspects of the context were most restrictive. It could be useful, to know if a resource is unavailable, a deadline was too tight, or if the desired quality level was unachievable. Similarly, when an action fails during execution, it is primarily the responsibility of the high-level reasoning component to determine why it failed and how to recover if the built-in conflict resolution system is unable to do so. Improving this capability, particularly in a domain independent fashion, is an area of future work.

## References


[1] R.H. Bordini, A.L.C. Bazzan, R.O. Jannone, D.M. Basso, R.M. Vicari, and V.R. Lesser. Agentspeak(xl): Efficient intention selection in bdi agents via decision-theoretic task scheduling. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-2002)*, Bologna, Italy, 2002.

[2] Jeffrey S. Cox, Bradley C. Clement, Pradeep M. Pappachan, and Edmund H. Durfee. Integrating multiagent coordination with reactive plan execution. (abstract). In *Proceedings of the ACM Conference on Autonomous Agents (Agents-01)*, 2001.

[3] T. Dean and M. Boddy. An analysis of time-dependent planning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 49–54, St. Paul, Minnesota, August 1988.

[4] Keith S. Decker and Victor R. Lesser. Quantitative modeling of complex environments. *International Journal of Intelligent Systems in Accounting, Finance, and Management*, 2(4):215–234, December 1993. Special issue on "Mathematical and Computational Models of Organizations: Models and Characteristics of Agent Behavior".

[5] Keith S. Decker and Victor R. Lesser. Designing a family of coordination algorithms. In *Proceedings of the First International Conference on Multi-Agent Systems*, pages 73–80, San Francisco, June 1995. AAAI Press.

[6] E. Durfee and V. Lesser. Predictability vs. responsiveness: Coordinating problem solvers in dynamic domains. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 66–71, St. Paul, Minnesota, August 1988.

[7] S. Fujita and V.R. Lesser. Centralized task distribution in the presence of uncertainty and time deadlines. In *Proceedings of the Second International Conference on Multi-Agent Systems (ICMAS-96)*, Japan, 1996.

[8] John Graham. *Real-Time Scheduling in Distributed Multi Agent Systems.* PhD thesis, University of Delaware, January 2001.

[9] John Graham, Victoria Windley, Daniel McHugh, Foster McGeary, David Cleaver, and Keith Decker. Tools for developing and monitoring agents. In *Proceedings of the Distributed Multi Agent Systems Workshop on Agents in Industry at the Fourth International Conference on Autonomous Agents*, Barcelona, Spain, June 2000.

[10] Bryan Horling, Brett Benyo, and Victor Lesser. Using self-diagnosis to adapt organizational structures. In *Proceedings of the Fifth International Conference on Autonomous Agents*, pages 529–536, 2001.

[11] Bryan Horling and Victor Lesser. A reusable component architecture for agent construction. Master's thesis, Department of Computer Science, University of Massachusetts, Amherst, 1998. Available as UMASS CS TR-98-30.

[12] Bryan Horling, Victor Lesser, Regis Vincent, Anita Raja, and Shelley Zhang. The tæms white paper, 1999. http://mas.cs.umass.edu/research/taems/white/.

[13] Bryan Horling, Régis Vincent, Roger Mailler, Jiaying Shen, Raphen Becker, Kyle Rawlins, and Victor Lesser. Distributed sensor network for real time tracking. In *Proceedings of the Fifth International Conference on Autonomous Agents*, pages 417–424, 2001.

[14] Eric Horvitz, Gregory Cooper, and David Heckerman. Reflection and action under scarce resources: Theoretical principles and empirical study. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, August 1989.

[15] Eric Horvitz and Jed Lengyel. Flexible Rendering of 3D Graphics Under Varying Resources: Issues and Directions. In *Proceedings of the AAAI Symposium on Flexible Computation in Intelligent Systems*, Cambridge, Massachusetts, November 1996.

[16] Francois F. Ingrand, Michael P. Georgeff, and Anand S. Rao. An architecture for real-time reasoning and system control. *IEEE Expert*, 7(6), December 1992.

[17] D. Jensen, M. Atighetchi, R. Vincent, and V. Lesser. Learning quantitative knowledge for multiagent coordination. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, Orlando, FL, July 1999. AAAI.

[18] Jaeho Lee, Marcus J. Huber, Edmund H. Durfee, and Patrick G. Kenny. Um-prs: An implementation of the procedural reasoning system for multirobot applications. In *Proceedings of the Conference on Intelligent Robotics in Field, Factory, Service, and Space (CIRFFSS '94)*, pages pp. 842–849, 1994.

[19] V. R. Lesser. A retrospective view of FA/C distributed problem solving. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(6):1347–1363, November 1991.

[20] Victor Lesser, Michael Atighetchi, Bryan Horling, Brett Benyo, Anita Raja, Regis Vincent, Thomas Wagner, Ping Xuan, and Shelley XQ. Zhang. A Multi-Agent System for Intelligent Environment Control. In *Proceedings of the Third International Conference on Autonomous Agents (Agents99)*, 1999.

[21] Victor Lesser, Keith Decker, Norman Carver, Alan Garvey, Daniel Neiman, Nagendra Prasad, and Thomas Wagner. Evolution of the GPGP Domain-Independent Coordination Framework. Computer Science Technical Report TR-98-05, University of Massachusetts at Amherst, January 1998.

[22] Victor Lesser, Bryan Horling, Frank Klassner, Anita Raja, Thomas Wagner, and Shelley XQ. Zhang. BIG: An agent for resource-bounded information gathering and decision making. *Artificial Intelligence*, 118(1-2):197–244, May 2000. Elsevier Science Publishing.

[23] Victor Lesser, Atighetchi Michael, Brett Benyo, Bryan Horling, Anita Raja, Regis Vincent, Thomas Wagner, Ping Xuan, and Shelly XQ Zhang. The umass intelligent home project. In *Proceeding of the Third Conference on Autonomous Agent*, 1999. http://mas.cs.umass.edu/research/ihome/.

[24] Victor R. Lesser. Reflections on the nature of multi-agent coordination and its implications for an agent architecture. *Autonomous Agents and Multi-Agent Systems*, 1(1):89–111, 1998.

[25] Roger Mailler, Régis Vincent, Victor Lesser, Jiaying Shen, and Tim Middlekoop. Soft real-time, cooperative negotiation for distributed resource allocation. In *Procceedings of the 2001 AAAI Fall Symposium on Negotiation*, 2001.

[26] David J. Musliner, Edmund H. Durfee, and Kang G. Shin. CIRCA: A cooperative intelligent real-time control architecture. *IEEE Transactions on Systems, Man and Cybernetics*, 23(6), 1993.

[27] Anita Raja and Victor Lesser. Meta-level control in multi-agent systems. Technical report, University of Massachusetts, Amherst, November 2001. ftp://ftp.cs.umass.edu/pub/techrept/techreport/2001/UM-CS-2001-049.ps.

[28] Anita Raja and Victor Lesser. Towards bounded-rationality in multi-agent systems: A reinforcement-learning based approach. Technical report, University of Massachusetts, Amherst, August 2001. ftp://ftp.cs.umass.edu/pub/techrept/techreport/2001/UM-CS-2001-034.ps.

[29] Stuart J. Russell and Shlomo Zilberstein. Composing real-time systems. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pages 212–217, Sydney, Australia, August 1991.

[30] Wolfgang Slany. Scheduling as a fuzzy multiple criteria optimization problem. *Fuzzy Sets and Systems*, 78:197–222, March 1996. Issue 2. Special Issue on Fuzzy Multiple Criteria Decision Making; URL: ftp://ftp.dbai.tuwien.ac.at/pub/papers/slany/fss96.ps.gz.

[31] John A. Stankovic and Krithi Ramamritham. Editorial: What is predictability for real-time systems? *The Journal of Real-Time Systems*, 2:247–254, 1990.

[32] R. Vincent, B. Horling, and V. Lesser. An agent infrastructure to build and evaluate multi-agent systems: The java agent framework and multi-agent system simulator. In *Lecture Notes in Artificial Intelligence: Infrastructure for Agents, Multi-Agent Systems, and Scalable Multi-Agent Systems.*, volume 1887. Wagner and Rana (eds.), Springer,, January 2001.

[33] Régis Vincent, Bryan Horling, Victor Lesser, and Thomas Wagner. Implementing soft real-time agent control. In *Proceedings of the Fifth International Conference on Autonomous Agents*, pages 355–362, 2001.

[34] Thomas Wagner, Alan Garvey, and Victor Lesser. Complex Goal Criteria and Its Application in Design-to-Criteria Scheduling. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 294–301, July 1997. Also available as UMASS CS TR-1997-10.

[35] Thomas Wagner, Alan Garvey, and Victor Lesser. Criteria-Directed Heuristic Task Scheduling. *International Journal of Approximate Reasoning, Special Issue on Scheduling*, 19(1-2):91–118, 1998. A version also available as UMASS CS TR-97-59.

[36] Thomas Wagner and Victor Lesser. Design-to-Criteria Scheduling for Intermittent Processing. UMASS Department of Computer Science Technical Report TR-96-81, November, 1996.

[37] Thomas Wagner and Victor Lesser. Design-to-Criteria Scheduling: Real-Time Agent Control. In O. Rana and T. Wagner, editors, *Infrastructure for Large-Scale Multi-Agent Systems*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 2000. A version also appears in the 2000 AAAI Spring Symposium on Real-Time Systems and as UMASS CS TR-99-58.

[38] Thomas Wagner, Anita Raja, and Victor Lesser. Modeling uncertainty and its implications to design-to-criteria scheduling. Under review, 2002.

[39] Xiaoqin Zhang, Anita Raja, Barbara Lerner, Victor Lesser, Leon Osterweil, and Thomas Wagner. Integrating high-level and detailed agent coordination into a layered architecture. Lecture Notes in Computer Science: Infrastructure for Scalable Multi-Agent Systems, pages 72–79. Springer-Verlag, Berlin, 2000. Also available as UMass Computer Science Technical Report 1999-029.

[40] X.Q. Zhang, V. Lesser, and T. Wagner. A proposed approach to sophisticated negotiation. In *AAAI Fall 2001 Symposium on Negotiation Methods for Autonomous Cooperative Systems*, January 2001.

[41] S. Zilberstein, A.I. Mouaddib, and A. Arnt. Dynamic scheduling of progressive processing plans. In *Proceedings of the ECAI 2000 Workshop on New Results in Planning, Scheduling and Design*, 2000.

[42] S. Zilberstein and S. J. Russell. Optimal composition of real-time systems. *Artificial Intelligence*, 82(1):181–214, December 1996.

[43] Shlomo Zilberstein. Using anytime algorithms in intelligent systems. *AI Magazine*, 17(3):73–83, 1996.

[44] M. Zweben, B. Daun, E. Davis, and M. Deale. Scheduling and rescheduling with iterative repair. In M. Zweben and M. Fox, editors, *Intelligent Scheduling*, chapter 8. Morgan Kaufmann, 1994.