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, Régis
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 real-time planning and scheduling of activities, particularly in its interactions with other agents [24, 30]. 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 [41]. The SRTA architecture 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 use explicit representations of uncertainty and efficiently handle deviations in expected plan behavior that arise out of variations in resource usage patterns and unexpected action characteristics.

While the immediate motivation for this work is the domain described in section 2, we are more generally interested in demonstrating that agents employing complex modeling and decision making techniques can address problems posed by real-world scenarios. The architecture presented in this paper uses such techniques, enabling it to operate effectively in open, unpredictable environments by using online planning and scheduling algorithms that explicitly reason about uncertainty and have the ability to explore alternative ways to satisfy goals, temporal constraints and resource requirements. At the same time, it also efficient enough to work in soft real-time, manage interdependencies between tasks and resources, and satisfy commitments that may be formed between entities. In particular, this work is differentiated from others [33, 1, 32] in its explicit use of uncertainty, soft interrelationships, probabilistic action expectations, and a range of commitment types. We will show how this type of framework can provide many capabilities needed for sophisticated multi-agent applications.

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.

Abstractly, SRTA operates as a single functional unit within an agent, which itself is 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 [51, 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.

The system has evolved and been constructed over several research projects into a set of interacting components and representations, as shown in Figure 1. We first assume that goals can arrive at any time, in response to environmental change, local planning, or because of requests from another agents. A domain-independent, hierarchical task network description language called TÆMS is used to describe goals, which 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 (for example, Figure 4) [6, 15]. 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. A planner process, called DTC, uses the information encoded in the TÆMS structure to generate a number of different plans for achieving the desired goals, each with different characteristics, and ranked by their predicted utility within the current operating context. This permits the system to adjust which goals it will achieve, and how well it will achieve these chosen goals based on the dynamics of the current situation. A viable plan is then selected and used by a scheduling component (POS) to produce a partially-ordered sequence of activities addressing the goal. A resource modeling component is also used to identify and schedule any resource accesses that the activities perform. This schedule is then combined with schedules from existing goals to form a coherent, potentially parallel sequence of activities, which are partially ordered based on their interactions with 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 or if a viable schedule cannot be produced, SRTA can make use of an extensible series of resolution techniques to correct the situation, or generate an event to elevate the problem to higher level components which may be able to make a more informed decision. The components that comprise SRTA can assume responsibility for the majority of the goal-satisfaction process, which allows the high-level reasoning system to focus on goal selection, determining goal objectives and other potentially domain-dependent issues.

To demonstrate SRTA's capabilities and behaviors we will use a running example throughout the paper, taken from the distributed sensor network domain. We can make the sequence of actions described above more concrete by using a synopsis of this example. The goals of the sensor in question are described using the task structures shown in Figures 4 and 5. These goals may have several different ways to be satisfied, can interact with one another, and are identified dynamically at runtime. SRTA's responsibility is to select an appropriate set of actions and monitor the progress of those actions such that the goals are achieved. `Task1` arrives first, which requires the sensor to initialize itself and send a notification message to its manager. This goal is first retrieved from the local task library, after which the scheduling and resource modeling components are used to select a sequence of actions and generate a parallel schedule. An execution component then uses this schedule to drive the sensor's actions. `Task2`, which has the sensor take measurements by a certain deadline to help track a target, arrives before `Task1` has completed. It also is represented as a structure and used to produce a partially ordered sequence of actions, which must then be merged with the current working schedule. In this case, both goals make use of the `RF` resource, so using the resource modeling component the scheduler merges the two such that this constraint is satisfied. This is accomplished by representing that constraint, along with other intra-schedule conditions, as part a precedence graph that is used to which actions may be run in parallel or must be run in series. This graph can then be used to quickly select appropriate execution times for the different actions, respecting the ordering constraints within the goals, the resource constraints over `RF` and the deadline assigned to `Task2`. A `Task3`, which also uses `RF`, is added later in a similar manner. During execution, however, one of `Task3`'s actions takes longer than expected, which in turn affects start times later in the schedule. Resolving this problem is not simply a matter of shifting those tasks, because in this case it would cause a conflict with `Task2`'s use of `RF`. To resolve this, the scheduler uses the existing precedence graph to determine where actions may be shifted. It first recognizes that `Task2`'s deadline prevents its `RF`-using action from being delayed, but that the remaining `Task3` actions are unconstrained. The necessary changes are made to the schedule and all three goals are eventually satisfied.

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, hard real-time, which attempts to rigorously quantify and formally bound 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 [38], 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 poten-

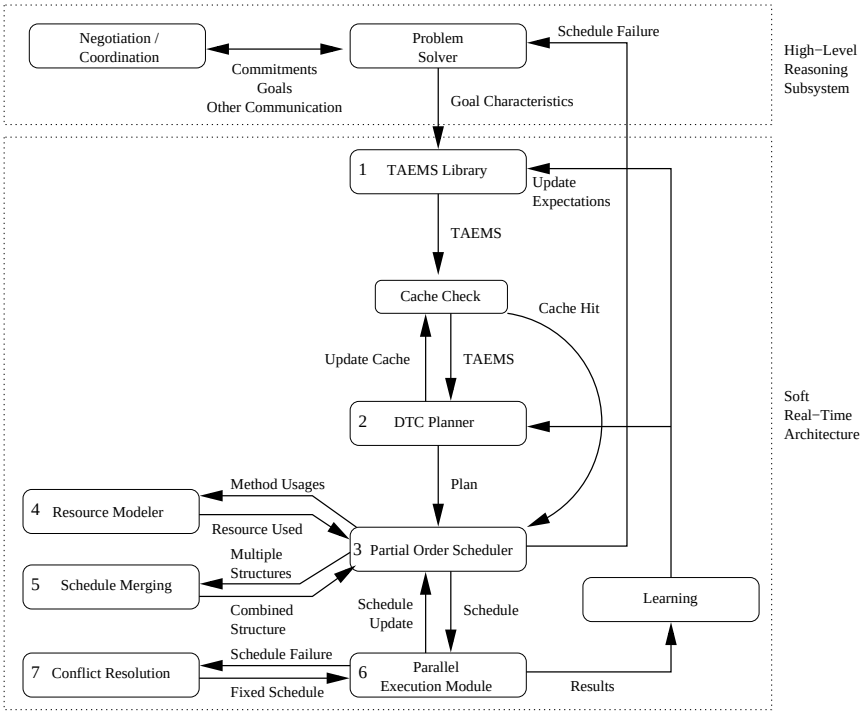


Figure 1: High-level agent control architecture.

tial 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, coordination 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, this technology has been used to successfully operate in a real-time distributed sensor network 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. SRTA addresses this on two fronts. The first is to implement the technologies needed to directly reason about real-time, while the second is to perform this reasoning efficiently enough to meet deadlines. As mentioned above, SRTA models the quantitative and relational characteristics of goals, activities and commitments in TÆMS, which can be done a priori and accessed as plan library, or dynamically through a runtime learning process[21] or generation by a domain-specific problem solver. This detailed description of the mechanisms used to satisfy a particular goal provides an important foundation upon which later decisions are made. A planning component, Design-to-Criteria (DTC) [43, 48], uses these TÆMS task structures, along with the quantitative knowledge of action interdependence and deadlines, to select the most appropriate set of actions given current environmental conditions. This abstract plan is used by the Partial Order Scheduler (POS) and Probabilistic Resource Modeling (PRM) components to determine when individual actions should be performed, either sequentially or in parallel, within the given precedence and runtime resource constraints, again using the descriptions provided by the task structures. Finally, the execution subsystem reasons about the fine-grained execution times of individual actions, ensuring that preconditions and deadlines are respected as actions are performed. These components address the real time requirements through the interactions of these 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, which enables agents to better manage tasks as they arrive at runtime. Together, these characteristics reduce the need for constant re-planning, in addition to making the scheduling process itself less resource-intensive. Learning can play 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. 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 demonstrating some of the features that are described are included, and we demonstrate how SRTA is used to enable agents to exhibit complex behaviors. We will conclude with an overview of related research and a discussion of overall conclusions, including future directions.

2 Motivation

The design we present assumes large, 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, which motivates a need for the type of agent control SRTA provides. A different approach addresses these issues through use of groups of simpler agents, which do not require complex control because they 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. If these smaller agents share limited local resources (for instance, processor time or network bandwidth), the additional overhead needed to coordinate their activities can exacerbate these restrictions.

Components of the SRTA architecture have been used successfully in several domains, such as intelligent information gathering [28], intelligent home control [25], and supply chain management [12]. The primary motivation for our current work is a distributed sensor network problem [17], which will be used in this paper to ground and provide examples of the topics which are discussed. In this domain, which will be discussed in more depth below, we have exploited SRTA to create an organization of agent roles which are instantiated within a smaller collection of real agents. Each role represents a particular goal or service that needs to be fulfilled, which is created as needed and dynamically assigned to a specific individual agent based on information approximating the current usage and availability of resources and skills. Individuals may be assigned multiple roles, such as tracking, sensing or managing, which motivates the need for detailed planning and scheduling based on local resource availability

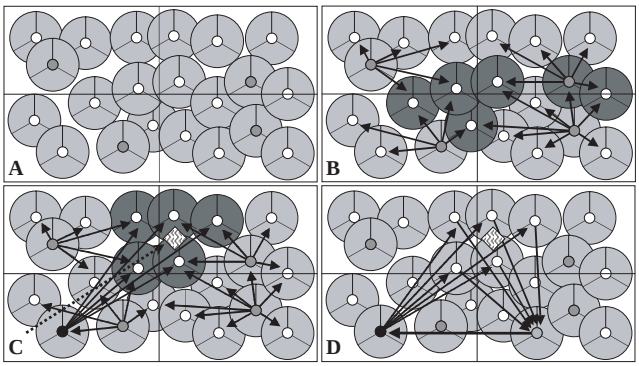


Figure 2: High-level distributed sensor allocation architecture. A) shows the initial sensor layout, decomposition and allocation of sector managers. B) shows the dissemination of scanning tasks. The new track manager in C) can be seen coordinating with sensors to track a target, while the resulting data is propagated in D) for processing.

and the priority of the different responsibilities associated with the roles. More abstractly, each of these roles is in some sense a distinct entity based on the actions it performs and the interactions it is involved with. SRTA is used to allow a combination of roles to coexist within a single agent in the environment.

The distributed sensor environment consists of several sensor nodes arranged in a region of finite area, as can be seen in Figure 2A. 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 trade-off 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 on the same target.[1]. Individually, each sensor can only determine the target's distance

[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.

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 trade-off 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 2A), target discovery scanning (2B), track management and measurement tasks for other targets (2C), and data processing (2D). Briefly, sector managers are responsible for storing local sensor information and arranging target detection scans, among other things. Track managers coordinate collection activities and fuse measurement data. Individual sensors perform measurements for both scanning and tracking tasks. These responsibilities, as well as a more complete description of the solution's organization, are covered in much more detail in [16]. We do not see these roles as separate agents or threads, but rather as a set of potentially interdependent tasks and goals that a single agent is fulfilling concurrently. For example, in 2C the agent performing the track negotiation is one that previously received a scanning task, and it theoretically could also be performing sector managerial duties (although it generally won't, as this would be poor load-balancing). 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, the 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.

3 Soft Real-Time Control Architecture

Our previous work in agent control architectures were tested almost exclusively in controlled-time simulation environments [40, 25, 14, 5] (one exception is [27]), and were also fairly large grained in their view of planning and scheduling. 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) planner/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 single, monolithic structure containing 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 re-scheduling. This technique leads to "optimal" plans and schedules at each point if meta-level overheads are not included. As will be discussed in section 3.3, 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. As will be shown, the scheduling and planning process in SRTA is more incremental and compartmentalized, enabling goals to be added piecemeal to the execution schedule as they are generated, without the need to re-plan all the agent's activities.

We will begin with a high level trace of the system's behavior, before continuing with more detailed explanations of each component. Activity begins as new incoming goals are used by the problem solving component to generate a TÆMS task structure, which can be produced in a variety of ways; in our case we use a TÆMS "template" library, which we use to dynamically instantiate and characterize structures to meet current conditions. Other options include generating the structure directly in code [28], or making use of an approximate base structure and then employing learning techniques to refine it over time [21]. This task structure provides the system with a quantitative description of how the goal might be satisfied. A single structure can define many alternative solutions with differing characteristics.

The task structure must then be analyzed, to determine which alternative most appropriately satisfies the goal in question. The Design-To-Criteria component, used in our earlier work on control architecture, retains a critical role in SRTA by addressing this need. Where before it was responsible for both selecting an appropriate plan of activities from the task structure and producing a schedule of actions for a monolithic structure describing all the agent's current goals, SRTA instead exploits its planning capabilities for discrete, individual goals. Using the TÆMS structure, 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, using the TÆMS structure to determine precedence constraints and search for actions which can be performed in parallel. Several components operate during this final scheduling phase. A resource modeling component is first used to ensure that resource constraints are 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 could relax constraints to resolve the problem. For example, it could alter the goal completion criteria or delay its deadline, complete a substitute goal with different characteristics, or decommit 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 in the originating TÆMS structure, and potentially re-invokes 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 [13]. 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 service guarantee, competing external processes may add an additional level of performance uncertainty.

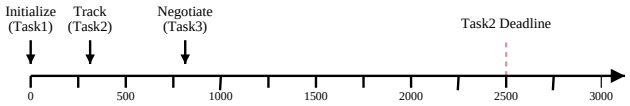


Figure 3: The timeline of events for the running scenario in the paper. Shown are the arrival times in milliseconds for the goals shown in Figures 4 and 5, along with the negotiated deadline for Task 2.

A few specific aspects of the JAF agent architecture are relevant to understanding how SRTA functions and will help understand the design and behavior of SRTA presented later. JAF agents are implemented as a collection of loosely coupled components, and a number of generic components are available for common tasks such as logging, communication, low-level agent control. SRTA conforms to this design principle by implementing the services shown in Figure 1 as individual components. A general information storage component also exists in JAF, which provides an agent-global repository for data. This is used to facilitate indirect knowledge sharing; for example, the schedule produced by the partial order scheduler can be stored there and easily used by the execution component. JAF also provides an event system, which allows a more dynamic flow of information through the agent. Events produced by individual components are automatically propagated to other components which have registered their interest in those events. We will see later how this is used to support the dissemination of method execution results and the handling of scheduling exceptions. The implementation details of these underlying support mechanisms are beyond the scope of this article, it is sufficient to know that they exist. More information on JAF can be found in [13].

To better explain our architecture's functionality, we will work through an 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 near time 300 to track a target and sent the results before time 2500. Later, a third goal, to negotiate for delegating tracking responsibility, is received near time 800. We will show how these different goals may be achieved, and their constraints and interdependencies respected.

3.1 TÆMS

Before progressing, we must provide some background on our task description language, TÆMS, although space limitations preclude a complete definition of the language, which can be found in [15]. TÆMS, the Task Analysis, Environmental Modeling and Simulation language, is used

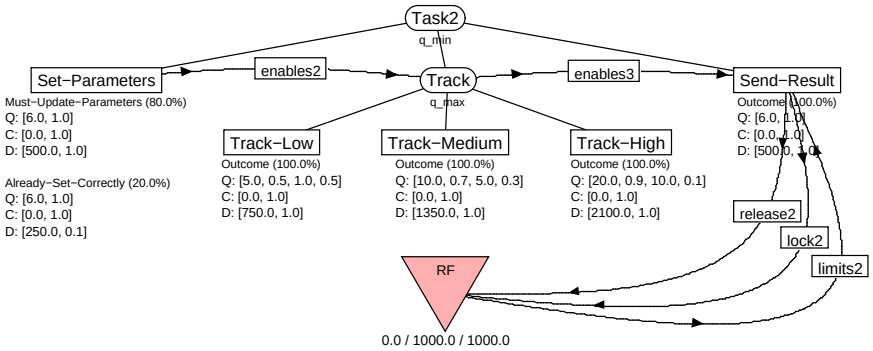
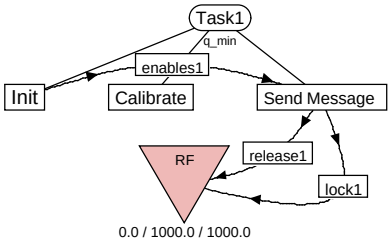
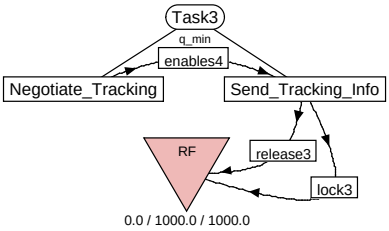


Figure 4: An example TÆMS task structure for tracking. The expected execution characteristics are shown below each method, and the `Send-Results` method in this figure has a deadline of 2500.



A) Initialization task structure.



B) Tracking goal negotiation task structure.

Figure 5: Two TÆMS task structures, abstractions of those used in our agents.

to quantitatively describe the alternative ways a goal can be achieved [6]. 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 subgoals, 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 as shown below the leaf nodes in Figure 4. These quantitative descriptions are themselves grouped together as outcomes, which abstractly represent the different ways in which an action can conclude. For example, `Track-Low` is shown to have a 50% chance of getting quality 5, a 50% change of getting quality 10, and a 100% chance of costing 0 and having a duration of 750. We will use such distributions at many locations in the structure to model uncertainty. If there is a connection between multiple characteristics of a particular method or if we desire a more explicit representation of method behavior, we can use a slightly different representation to more explicitly model individual behaviors. For example, `Set-Parameters` 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. 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. Section 3.6 describes in more detail how methods are used to cause agent behavior.

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`. Conceptually, these two QAFs are analogous to the behavior of standard *and/or* decision trees. Other accumulation functions such as *sum* (quality is the sum of completed subtasks's qualities), *last* (quality is that of the last completed subtask), and *exactly_one* (only one subtask may be performed) also exist. Complete descriptions of these and other QAFs can be seen in [15].

Interactions between methods, tasks, and affected resources are also quantitatively described as interrelation-

ships. The effect of an interrelationship is controlled by its source node, and its effects are imparted on its target. For example, the *enables* interrelationships in Figure 4 represent precedence orderings, which in this case say that `Send-Results` is enabled only when `Track` has successfully completed (i.e. has non-zero quality), which can only happen after `Set-Parameters` has successfully completed. This ensures that `Set-Parameters`, `Track`, and `Send-Results` are performed in-order. An analogous *disables* interrelationship exists, as well as the softer relations *facilitates* and *hinders*. When such a soft interrelationship is active, it affects one or more of its target node's quality, cost and duration characteristics as indicated by fields in the interrelationship's definition. For instance, one could *facilitate* a method by reducing its expected duration by 20%, or *hinder* it by increasing its cost by 50%. Soft interrelationships 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 as defined in the interrelationship. The resource effect is further quantified through the *limits* interrelationship, which defines the changes in its target method's execution characteristics when that resource is over-consumed or over-produced. It is similar to the *hinders* interrelationship, in that it will typically change its target's expected performance in a negative way. 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 implement 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.

Although not strictly part of a goal's decomposition, the TÆMS structure can also include descriptions of commitments associated with the goal. For example, if `Agent_A` has asked `Agent_B` to `Track` for it, this concept can be quantified and represented along with the goal that commitment relates to. This information can be represented in a pair of ways. First, methods or tasks can have deadlines associated with them, which would directly represent the notion that those activities have to be completed by some predetermined time. Second, a more complete description of the commitment can be bundled with the structure, including the source and target of the commitment, the type of commitment, how important it is, the desired minimum quality and any bounding start or finish times. The former specification is simple to use, while the details included in the latter definition permit more sophisticated reasoning because the merits and flexibility of the commitment can be evaluated if needed. We will see examples of both these specifications in figure 6 in the next section, and examples of their use in section 5.2. The agent can coordinate and negotiate with other agents, and use these features to instantiate the results of that communication so the commitments can be respected by other components.

As will be seen throughout this paper, probability and uncertainty play a large role in SRTA's technologies. Much of that information is derived from the quantitative distributions associated with the various elements in TÆMS. This feature, which is used to accurately model real-world situations where effects are not deterministic, allows the agent to effectively reason about the trade-off between reliability and performance under such circumstances. For example, in our distributed sensor network domain, the quality of the results produced by a tracking task is not always known. Some measurement techniques are reliable but of low quality, while others offer the potential of higher quality with possibility of failure – think of a wide area scan versus a focused beam which would need to hit a particular target. In situations where any knowledge is much better than none, such as when an agent is attempting to reacquire a lost target, the more reliable method would be preferred despite its reduced precision (represented here as lower quality). In other circumstances, such as when redundant sensors are also tracking a target, increased uncertainty is a reasonable cost for potentially higher quality data. One can also use TÆMS to model the probability that a remote agent will successfully meet its commitments, that sufficient resources will be available for a task, or that one action will help another complete. Using this type of information, SRTA's planning and scheduling components can select and enact a course of action which has a level of certainty appropriate to the current situation.

In later sections we will cover more technique-specific components of TÆMS, including schedules, criteria and preconditions that can be associated with a structure.

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. Furthermore, the structure is maintained while the goal is in progress, and by updating it with the results of activities as they are performed it also serves as a valuable source of runtime information.

3.2 TÆMS Library

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 directly in the source code using planning techniques, but this tends to be impractical if the agent must define multiple complex or heterogeneous structures and deadlines are relatively tight. 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.

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)
)
```

Figure 6: 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.

This works well, except it lacks the flexibility to easily handle the minor variations in structure needed when environmental conditions shift. We have developed a hybrid scheme, which uses a library of TÆMS *templates* that allow particular aspects of the structure to remain undefined until it is instantiated. Template are then dynamically instantiated at runtime by passing in parameters allowing the structure to better represent the agent's current working conditions. In this way it is easy to handle such things as varying execution performance, negotiation partners and commitment details without the need to hard-code the entire structure or plan it entirely from first principles.

The template itself is defined using a relatively simple one-pass language. It was intentionally designed to be similar to the lightweight macro facilities available in C compilers, as this is a familiar paradigm to most developers that strikes a reasonable balance between static structures and ones completely specified in code. Unlike a traditional macro language, it includes support for creation and manipulation of individual numeric and string variables, as well as composite lists of these elements. `if-then-else` and `while` control structures are available, as are the standard boolean and inequality operators. The template language also includes a number of built-in functions, which can perform simple arithmetic and string operations, generate random numbers and select random elements from lists. The latter two are useful when one needs to create a range of structures with similar characteristics for testing purposes. More details on the textual specification of TÆMS structures can be found in [15].

A small example of a template structure is shown in Figure 6, which compares the 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 Figure 6 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 min-

imum 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).

At time 0 the agent will use its template library to generate the initialization structure seen in Figure 5A. This is done by selecting the template associated with the goal `Task1`. In this structure, the agent must first `Init` and `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.

Although it is not a requirement, each goal will typically be represented by a single task structure, distinct from other structures currently in execution. Once parsed and instantiated, this structure is then added to the collection of current working tasks. As with the scheduling process's similar characteristic, this incremental addition of new structures from the library facilitates the task generation process by focusing on only the single goal in question. The alternative, to keep a single working task structure which is regenerated and augmented as new goals arrive becomes much more expensive to produce when multiple goals are being pursued concurrently. `Task2` and `Task3`, shown in Figures 4 and 5B, respectively, are generated later in our scenario. They are produced and remain structurally distinct from the currently running `Task1`, although the three goals are not necessarily independent from one another. In this case, the tasks will eventually interact through their use of a shared resource. It is also possible to define *nonlocal* nodes within one structure that refer to tasks in another structure, providing a different mechanism to specify inter-task structure interactions.

3.3 DTC Planner / Initial Scheduler

The Design-to-Criteria (DTC) [43, 48, 36, 26, 44, 45] component is responsible for evaluating different possible courses of action that can be used to solve an agent's goal, and then choosing one or more solutions that best fit 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, start-to-finish view of goal satisfaction, 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 techniques used by the DTC component have certain characteristics in common with both planning and more traditional scheduling problems, and it suffers from pronounced combinatorics on both fronts. DTC's function is to read as input a TÆMS task structure (or a set of task structures) provided by the problem solver and to 1) decide which set of tasks to perform, 2) decide how to go about performing each task (there are generally several alternative ways to go about performing a task), 3) decide sequencing constraints among the tasks and their subtasks, taking advantage of soft relationships where possible, 4) to perform these 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. The result of this process is a set of candidate plans, which are ranked by how well their characteristics satisfy the provided criteria.

DTC's planning/scheduling problem differs from more conventional problem spaces in several ways. At the root of the differences lies the TÆMS task modeling framework and the problem space richness it is designed to model. In TÆMS any task may affect any other task and these interactions are not limited to binary effects – all interactions in TÆMS are quantified in terms of their impact on the recipient node's quality, cost, and duration distributions. The implications of this are that tasks can directly and indirectly affect one another's quality, cost, and duration distributions, and that these effects must be evaluated in context. The planning/scheduling problem faced by DTC is thus not one where constraints are independent or where the affects of constraint violation are fixed. Instead, one might view the process as one in which each time a step along a particular course of action is considered, a new set of constraints (and their effects) must be propagated through the remaining network and their effects considered. In addition to this inherently complex reasoning process, DTC can take as input a specification of the agent's dynamic objective function or design criteria. Said criteria describe for DTC the types of solutions that are desirable for the agent at this point in time. In English, one might describe a situation as needing agent plans that are "highly certain but where overall duration is less important," or where "fast solutions are desired as long as the cost stays under limit X and the quality is over threshold Y." From a human user's perspective, these characteristics can be modeled with a set of sliders, as shown in Figure 7, each of which define a particular attribute of the criteria [42]. This criteria bundle or objective function is dynamic and specified at run time. Thus DTC must work toward the objective function while reasoning about the hard constraints present in the TÆMS task structure, e.g., deadlines or earliest-

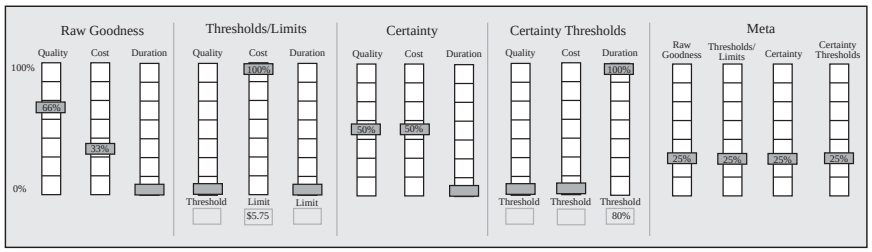


Figure 7: The "slider" model for specifying and interpreting the range of criteria DTC uses to weight its plan generation and selection process.

start-times on tasks, and while considering interactions that span tasks (and their effects), considering commitments or deals made with other agents (another form of both hard and soft constraints), and considering a course of action from an end-to-end view to meet overall real-time deadlines – and it must perform these operations online. Table 2 in the later section 5.3 shows an example of how the planning process produces different results given different objectives.

Solving this hybrid planning/scheduling problem is non-trivial. 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, DTC 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, DTC 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 to produce the final schedule of activities in SRTA, this 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 as a whole. An incomplete view might allow a plan to be selected which has undesirable characteristics past the horizon bounding the analysis.

$$\sum_{i=0}^{n} \binom{n}{i} i! \qquad (1)$$

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 [43, 48, 36, 26, 44, 45]. From a very high level, DTC uses goal directed focusing, approximation, scheduling heuristics, and schedule improvement/repair heuristics [55, 37] 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 [18], but differs from

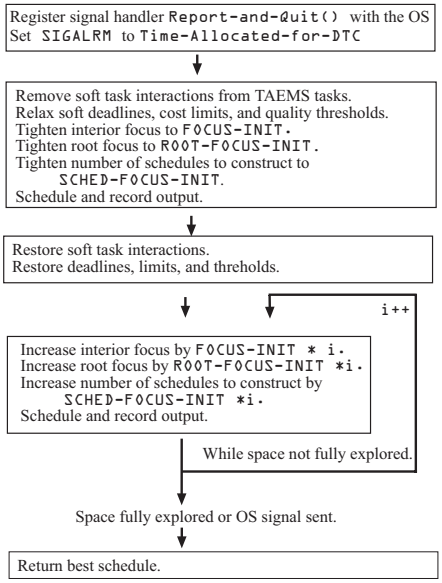


Figure 8: Real-Time Control for DTC

most flexible computation approaches in its use of multiple actions to achieve flexibility (one exception is [19]) in contrast to *anytime algorithms* [4, 54]. 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 [52], but we can work to reduce it when in the criteria for achieving a goal designates this characteristic as important.

For many applications, DTC supplies 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 [28], scheduling/planning accounted for less than 1% of the agent's execution time. Other examples include dynamic supply chain management [45], aircraft service team coordination [44], and crisis response management [46]. However, in these applications the time available to control problem solving ranges from ten seconds upwards to minutes. In tighter real-time situations, being fast enough may not be sufficient if the execution time is not strictly predictable or bounded, as discussed in [48]. With the previous generations of DTC it was possible for DTC to generally take on the order of a few seconds (being "fast enough") for most

input sets but to occasionally take much longer to generate a solution deemed good by the criteria bundle and DTC's termination control. While the possibility of this can be predetermined by evaluating the problem instances (or classes thereof) and the elements that are dynamic in each problem episode, in some domains the possibility of an over-run is unacceptable. This sensor application is one such domain. To address the problem, the current generation of DTC supports real-time deadlines governing its execution time at the grainsize afforded by the underlying operating system, i.e., a means to specify a hard bounds on the amount of time allocated to DTC to force it to return by a given time. The control algorithm that is used is shown in Figure 8. To meet hard deadlines on the amount of time DTC 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. DTC 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 rescheduling starts again. This process continues until the hard-deadline is met or DTC explores the entire scheduling space. If the hard deadline occurs before DTC 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[34, 35].

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 [38] 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 [48]. 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.

3.4 Partial Order Scheduler

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. With respect to that one goal, these plans are complete and appropriate. However, within the larger context of other activities and goals being addressed by the agent, the DTC scheduling process has a limited view, and produces plans that may require additional analysis before they are used. For efficiency purposes, and because DTC had a different focus during much of its development history, it does not directly consider potentially conflicting actions or resource usage caused by competing goals or remote agents. The set of candidate plans it produces is instead used as part of a broader scheduling process performed by a pair of control components. These components expand upon these plans by recognizing constraints based on an understanding of method interactions and resource usage. The first component, the partially ordered scheduler (POS), performs a more detailed analysis of the structures produced by DTC to determine interdependencies and interactions, which are then used to determine appropriate execution times. The second, a probabilistic resource modeling component (PRM), is discussed in the next section.

Despite the fact that they are separate components, some overlap exists between DTC and POS, particularly in the area of sequencing activities. This separation both helps and hinders SRTA's scheduling process. On one hand, some of the effort expended by DTC is either replicated or not needed by the POS process. On the other hand, this separation does help to bound the already complex task of reasoning effectively about TÆMS tructures. In addition, this section will show that the aspects not handled by DTC, particularly resource and parallel activity scheduling, are effectively handled by other components.

A deeper question is whether the candidate set of plans produced by DTC is appropriate given that these additional characteristics are not represented, because the final schedule is generated from a member of that set. Because SRTA operates in a satisficing manner, rather than optimal, it is generally the case that this set will be

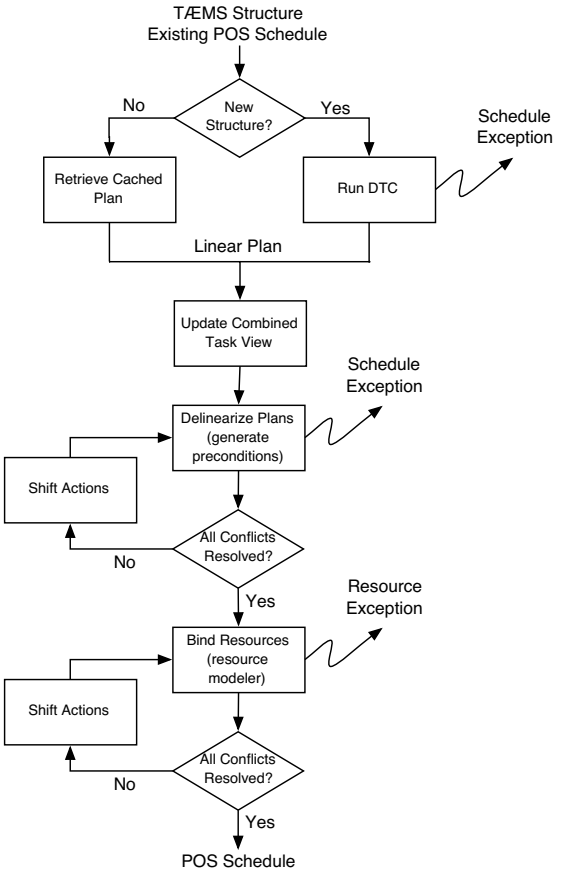


Figure 9: Control flow employed by the partial ordered scheduler.

sufficient and appropriate. Clearly there can exist pathological conditions which exploit this condition, but in our experience this is a very rare occurrence. Thus, the complexity of DTC, coupled with the seemingly minimal gains that would result from a tighter integration, provides insufficient motivation for merging these two technologies. However, new systems not affected by this issue would likely benefit from a tighter coupling.

Figure 9 shows the flow of actions performed by the partial ordered scheduler. It first provides DTC with the incoming task so it can produce a linear plan, or it retrieves an appropriate cached plan if it is available (more details about schedule caching are given in a later section). The linear plan is capable of meeting specified commitments, including hard deadlines generated by commitments to other agents and overall goal deadlines. Internally, the POS maintains a combined view of all the tasks the agent is currently working on, so it has ready access to existing task interactions. Before proceeding, the POS removes any task structures from this combined view that have been completed or canceled by the problem solver, after which it will add the new task. The POS will then use this combined view, building on the results from previous analyses to delinearize the schedule by generating the appropriate preconditions for the newly added task. These preconditions include descriptions of the interrelationships between the scheduled actions in addition to their desired execution times. The heart of this partially ordered schedule is a precedence graph, which explicitly represents the relationships between methods, constraints and deadlines. Assuming there are no constraint violations, the scheduler will then try to bind resources to the schedule with the help of the PRM. If the POS cannot resolve all the constraints and resource requirements, it will generate a schedule or resource exception, so that other components (such as the conflict resolution module or problem solver) can attempt to eliminate the conflict.

The basis for the precedence graph is a set of preconditions and time constraints associated with each method. A precondition is not a specific single structural element of TÆMS per se; they represent any one of a group of TÆMS elements and characteristics which control when a method may be successfully performed. For instance, preconditions are used to indicate that a method has an *enables* interrelationship targeting it. They are generally not added by the structure designer (although nothing precludes this), but instead are produced as part of the scheduling and analysis process performed by the POS. It looks at each method present in the highest rated plan provided by DTC, and evaluates its context in its associated structure. If the POS finds a characteristic which imposes an ordering constraint, an appropriate precondition is created and attached to the method to represent that fact. We will show later how these are used during method execution in section 3.6.

We will describe the two categories of constraints that are of interest in this process: those which are statically defined by the structure, and those that are dynamically imposed by the environment. The static components in the structure are relatively easy to find, while dynamic relations, which are deduced from the execution context, typically require further analysis to discover.

Interrelationships. We first saw interrelationships in section 3.1. *enables*, *facilitates*, *disables* and *hinders* interrelationships each define a particular effect which the completion of one task can have on another. When one is found targeting a method in the schedule, an appropriate precondition is generated which specifies which action should be performed before the other. The set of preconditions generated for a given methods is the union of all the direct preconditions and all the "inherited" preconditions.

Sequenced QAFs. Sequential QAFs (e.g. *seq_sum*, *seq_min*, etc.) are just like regular QAFs except they also enforce task ordering constraints. They are generally used for convenience, to represent the common situation where all of a task's subtasks would otherwise need a chain of enablements. They are handled just like such a chain of interrelationships existed.

Predefined Time Constraints. Deadline and earliest start time constraints can be specified directly by tasks, methods and commitments. When found, appropriate preconditions are set which force such actions to run within these proscribed time bounds.

Dynamic Time Constraints. Like their predefined

cousins, dynamic time constraints also bound the start and finish times of actions. Their origins are different, however, as they are discovered from the effects of other constraints, rather than directly observed. For example, if a series of methods are ordered by a set of enables interrelationships, and the last method has a deadline associated with it, then a dynamically computed deadline constraint will be created, propagating back through preceding methods. Each new deadline will take into account the deadline of the method coming after it, along with the expected duration of that method. By computing the difference of these two values, one can determine the appropriate deadline for the preceding method. Because durations are uncertain, and defined with a distribution, while deadlines and earliest start times are single values, we use the maximum duration in these computations.

Resource Usages. Resource constraints, which are based on the amount of resources consumed or produced by a method, are discovered with the assistance of the resource modeler discussed in the next section. When a valid point in time has been found for a method, the resources it requires are bound to it, which is accomplished by providing the resource modeler with a description of that requirement. From the partial order scheduler's point of view, the resource model itself is used to detect and cache resource relationships, which can be used to facilitate the scheduling process just like any other ordering relationship. These requirements are also stored as preconditions associated with the methods, facilitating the checks which must be performed before the method is executed.

The partial order scheduler searches the task structure for the elements listed above to generate the preconditions needed for the precedence graph. 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 5A. While an *enables* interrelationship orders `Init` and `Send-Message`, it does not affect the `Calibrate` method. 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. This information is used to construct the static portions of the graph shown in Figure 10.

Precondition creation is really an iterative process, because of the dynamic nature of some of the conditions. The complete precedence graph is generated as these constraints propagate, and is updated when attempts to schedule methods reveal new conditions. This is particularly true of resource constraints, which are not discovered until an active search is made to bind a method's resource consumption that allows these interactions to be manifested in the resource model.

The precedence graph and shifting 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. For example, we will see in section 3.6 that the execution component needs the ability to make minor adjustments to the schedule to keep the existing schedule in agreement with the reality of the agent's actions. Using the precedence graph, the POS can quickly see if a shifted method remains satisfiable in the schedule by analyzing its preconditions and time constraints. In addition, if the movement of that method in turn affects the timing of other actions, they too can be quickly reevaluated. If their new positions occur withing previously computed bounds, no further actions is necessary. If their predefined bounds are passed, the precedence graph can be used to search for a new location if one exists.

Once the graph has been produced, it can then be used both to determine which activities may potentially be run concurrently, because they have no precedence relation between them[2], and where particular actions may be placed in the execution timeline. Wherever possible, actions are parallelized to maximize the flexibility of the agent. 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 specific mechanism used to merge the schedules is identical to that used to determine order of execution within an individual plan, and in fact the majority of the information it uses to accomplish this is previously produced by the POS. Interdependencies between this combined set of methods, either direct or indirect, are used to determine which actions can be performed relative to one another along with time and resource preconditions to determine the final desired order and timing of actions. The resulting merged schedule is again stored as a precedence graph, so that preconditions can be quickly checked and actions shifted in time as needed.

Figure 10 models the complete set of our running activities, including the dynamic constraints which have been discovered through the iterative analysis mentioned earlier. For instance, the deadline (2500) constraint on `Send-Result`, which causes other, dynamic 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 later using the resource modeler. 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. In

[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.

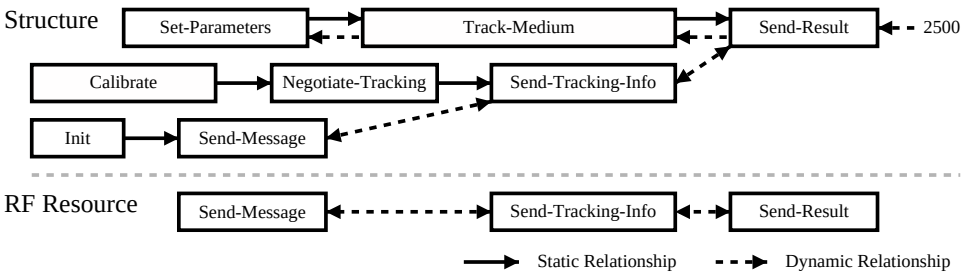


Figure 10: A partially ordered precedence graph modeling the running example on the task structures shown in Figures 4 and 5. Both static (i.e. those derived from interrelationships, QAFs, etc.) and dynamic (from resources, deadlines, etc.) constraints are shown.

this example, the constraint over the `RF` resource between `Send-Message` and `Send-Result` has not been found because no attempt was made to schedule them concurrently.

While schedules are being combined, the scheduler must be mindful that any shifting of tasks which takes place does not break any existing preconditions. Importance values are used to determine how best to handle unschedulable 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. All tasks have estimated quality values which can further discriminate among competing actions. Thus, when scheduling conflicts arise, it can compare the relative importance of the conflicting actions, and remove the one of lesser priority. This does presume that these values are assigned using the same scale, or are normalized in some way. Since the agent itself is assigning these values, however, this is a reasonable condition. If a decommittal is made, or if no valid resolution can be found, a scheduling exception event is generated which enables other components, such as domain problem solver or the conflict resolution component described in section 3.7, to take appropriate action.

The results currently produced by the POS are sensitive to the order in which the TÆMS task structures are inserted, because previously scheduled and executed methods will take precedence in the final schedule. If the POS cannot find a schedule accepting a new task, it does not backtrack and try another insertion order. This feature could be added, but in most of our application domains the combinatorics of such a search preclude SRTA from performing it.

In the future, we hope to add the ability to produce schedules which posses more general characteristics. For example, a forward-looking problem solver might want to produce schedules which retain a certain level of flexibilty. By leaving excess or slack time in the schedule, the agent would have greater flexibility in scheduling future actions, at the expense of an immediate schedule with potentially lower utility. Similar preferences could be modeled for controlling resource flexibility. In some sense, this would be an enhancement of the critera specification currently used by DTC to control schedule preferences for cost, quality, uncertainty, and the like.

3.5 Probabilistic Resource Modeler

The responsibility of binding (conceptually allocating) resources to specific activities belongs to a separate component called the Probabilistic Resource Modeler. It maintains a structure which tracks the expected use of resources over time. This is done by creating a model for each resource containing descriptions of the usage events that are currently known, which are used to estimate aggregate demand over time. This approach is similar to the resource allocation task in the factory scheduling domain from [39], with an additional probablistic component that adds uncertainty to resource consumption levels, start times and durations. As new activities are scheduled, the timeline is searched to ensure that their resource requirements can be met, and the model is updated with these new usages once a valid spot has been found.

All resources have a current level associated with them. This might indicate how many sheets of paper are left in a printer, how much bandwidth is available in a network, or what the current noise level is in a room. They can be *consumable*, as with the paper, or *nonconsumable*, as with the network. These two differ in that nonconsumable resources will automatically resume their previous state when usage stops, while a consumable one must be explicitly produced. The `RF` resource in our running example is like the network case. It indicates how much of the `RF` resource, which corresponds to the radio-frequency based wireless communication medium, is available for the agents to use. In our environment, two messages sent at the same time will collide and be corrupted, so whenever a method uses `RF`, the model should indicate that it is completely consumed. So, if our resource starts with a level of 1000, a single agent sending a message will consume all 1000 available units and bring it down to 0. If two agents send a message, they will both consume 1000, and the level will drop to -1000. At this point the resource will be over-consumed, and the affected methods will fail.

The timeline for a resource is initialized when that resource is seen for the first time in a TÆMS structure. As structures are produced by the problem solving component, they are quickly scanned by the resource modeler

for resources. If a resource is found which does not yet have a model, one is created from the data in that task structure. For instance, we first saw the `RF` resource in Figure 4. Associated with that node is a current level, as well as minimum and maximum values. In this case, the `RF` node indicates that its level is currently at its maximum value of 1000, and that it has a minimum value of 0. These values would be used to initialize the `RF` timeline maintained by the resource modeler.

Although it is a distinct component, the resource modeler is generally used as a tool of the partial order scheduling process. During its analysis, the POS produces a description of how a prospective schedule is expected to use resources, if at all. This is created using the TÆMS structure, by searching for the *produces* and *consumes* interrelationships mentioned in section 3.1. This description specifies the resource in question, the duration of the usage, what quantity 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, all of which can be obtained from those interrelationships. If multiple resources are affected by a method, then an equal number of usage descriptions are created, all of which must be satisfied. The scheduler then gives these 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 has some amount of uncertainty associated with it. The start time, duration and quantity of the usage are all probabilistically described, so the scheduler must also provide a minimum desired chance of success to the modeler. In this case, a usage is a success when it can be performed without violating the minimum and maximum values of the resource. While searching for an available point on the timeline, the cumulative effects of these uncertain usages must be computed for an accurate assessment to take place. Therefore, at any potential insertion point, the modeler computes the aggregate effects of the new resource usage, along with all prior usages up to the last known actual value of the resource. This is done by first computing the joint probability table for the discrete usage distributions in each time slot and then using these to compute the aggregate probability of the resource's level, using the algorithm presented in Figure 11 in both cases. Usage events are considered independent. For consumable resources, this aggregate consolidates all the data from the most recent measured level of the resource. Because a nonconsumable resource will automatically replenish itself, the current level can be assumed to be the base (original) level, and we can compute its aggregate using only this level and the usages from the time slot in question. These joint distributions are cached where possible. If the probability of success for this aggregate usage lies above the level 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 must then be performed for all other points in that range as well. If all points meet the success requirement, the resource modeler will return the valid

```
compJointDist(d̄) {
    compJointDist(0, 1, d̄, 0)
}
compJointDist(v_c, p_c, d̄, i) {
    if (i < |d̄|)
        ∀d ∈ {d_i, ..., d_{|d̄|-1}}
            ∀⟨v, p⟩ ∈ d
                compJointDist(v_c + v, p_c × p, d̄, i + 1)
    else
        output ⟨v_c, p_c⟩
}
```

Figure 11: Computing the joint probability from a set of discrete distributions. $\bar{d}$ is the vector of input distributions, while v_c and p_c represent the current value and probability being generated, respectively.

point in time.

The `RF` resource starts out with a level of 1000. Assume that a single usage of exists with a 0.5 probability of consuming 1000, and an equal chance of consuming 0. A method is being added which has a 0.2 probability of consuming 1000, and a 0.8 probability of consuming 0, and we wish to determine if it can be added to the end of the timeline. The resulting effects of these usages leads to a 10% chance `RF` will be left at level -1000, 50% chance at level 0, and 40% chance at level 1000. The resource's bounds are exceeded if it reaches -1000, so this usage may be added there with 90% certainty that it will succeed. If the scheduler requires greater than 90% certainty, then this particular insertion point is not viable.

If a particular point in time is not free, and the finish time bound has not yet been reached, the resource modeler will continue its search. Rather than looking at every sequential point in time, the modeler instead progresses 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 interesting time point to the next, by 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 expensive 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 5. `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 produce that same amount when it completes, with a probability of 1.0. 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

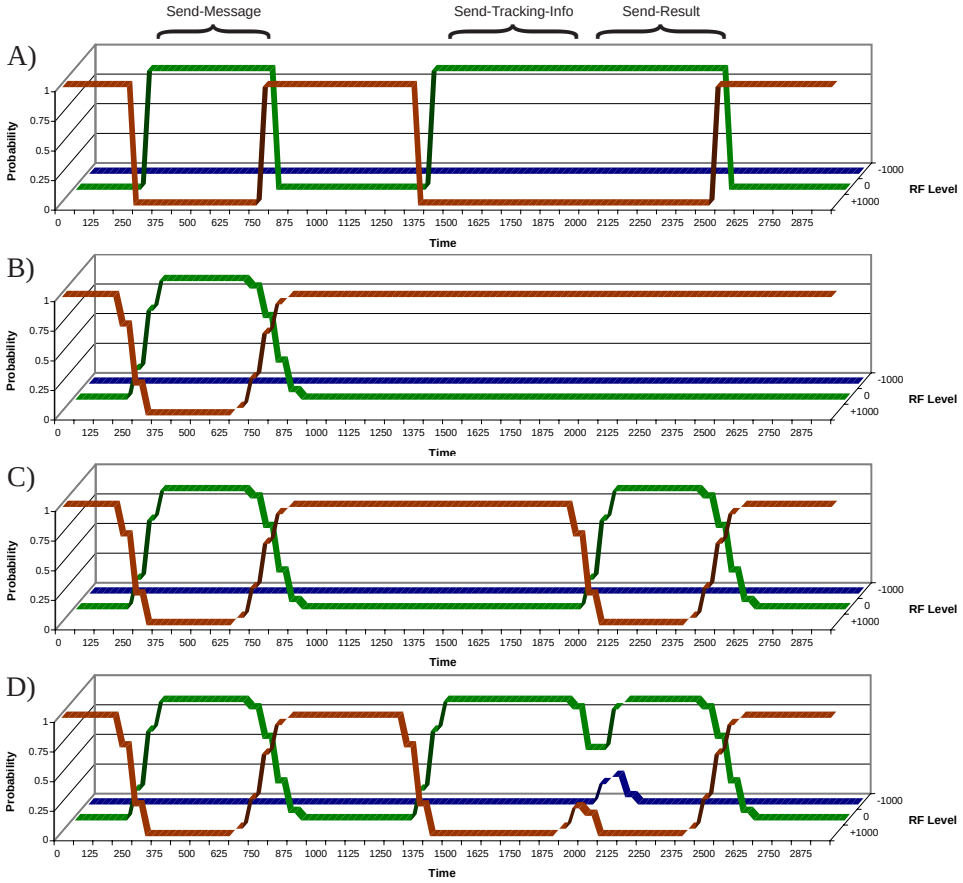


Figure 12: 3D graphs depicting resource model states after various usage patterns have been added. Each of the three series of values in each graph represents the probability of a particular `RF` resource level occurring (labelled 1000, 0, and -1000 on the right side axis) over time. A) shows the model 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.

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 12A. This graph shows what the probability of a given resource level will be at a given time. Each graph depicts the change in probability over time for three different resource levels (-1000, 0, +1000). So, looking at the +1000 line in the foreground shows the probability that the resource will be unused at any given time, while the -1000 line in the back shows the probability the resource will be overconsumed. For instance, at time 2000 in graph A, 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.

Our running example provided a fairly straightforward usage pattern - a more interesting pattern occurs when uncertainty is introduced into the schedule (although our example is not normally uncertain). 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 12B 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 12C. 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 12D 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 0, and would cause one of the methods to fail. If this success probability is too low, the search will continue past this point until a better time slot is found, allowing a trade-off between success probability and the timeliness of the activities.

After a viable point is found by the resource modeler, the scheduler will insert the method's usage description into the model, which will then be taken into account by subsequent scheduling. Since not all resource usage knowledge is necessarily derived locally, the modeler also supports the addition of nonlocal usage information. For instance, if the agent has communicated with a third party and subsequently gains knowledge of that entity's resource use, that can be added to the model in a similar manner as above and used during scheduling. This feature allows the scheduling process to easily recognize and avoid resource conflicts with other agents, provided the information can be obtained in the first place. Later, and if the resource itself supports such instrumentation, the resource modeler will periodically check each resource's current level. This information is used to update the models over time, so that subsequent scheduling is based on current information.

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 13A, 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.

The resource modeling component has also been used to help detect and diagnose failures in agent activity. In [12] we describe a causal-model based diagnostic engine, used to monitor an agent's activity and determine potential sources of failures. The resource modeler was used to first model expectations, and then determine if those expectations were met as methods were performed. A failed expectation could then provide evidence for a more detailed diagnosis, such as a method's usage being incorrectly modeled, other agents using the resource without local knowledge, or a malfunctioning resource.

3.6 Method Execution

Method execution is managed by the execute component, using the schedules produced earlier in SRTA. How particular methods are actually implemented and carried out will vary from one environment to the next, but in all cases they are initiated and monitored by the execution component. The execution component accomplishes this by periodically analyzing the current schedules to find a group of candidate actions which have scheduled start times equal to the current time.

Before being started, each member of the candidate group is checked to see if its *preconditions* have been satisfied. These are the same preconditions generated by the POS while building its precedence graph. By convention, we assume that failed preconditions will lead to method failure, so agent resources are not wasted running methods unless they are known to be satisfied. In this context, preconditions are used to abstractly and uniformly represent the notion that some characteristic must be true for a particular scheduled method, to simplify the task of checking them that the execute component must perform. Then, instead of needing to analyze the TÆMS structure for any of the various things which could prevent method execution, the search is limited to only the set of easily verified preconditions associated with method in the schedule. Methods which fail one of their preconditions are delayed, using the scheduler's shifting mechanism described earlier in section 3.4 to postpone the method's scheduled start time. Methods which have missed their start times, because the execute component was unable or not given the opportunity to check the schedule at that time, are also shifted using this method, providing another opportunity to run if possible.

Methods which meet their preconditions can be 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 must be 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, by 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 using structures provided by the JAF architecture. A more domain specific component in the agent could use this data to support other actions, such as notifying collaborating agents, performing learning or data analysis, or looking for new goals.

How methods are actually performed is a relatively domain-dependent issue. They might be performed in name only, modeled or simulated based on their given characteristics, or performed for real by the agent in question. The execution component simply assumes that methods are asynchronous (i.e. that control returns immediately after the action is initiated) and can potentially be run in parallel. The execute component itself is responsible only for determining when the action should start and tracking its progress.

In other respects, however, the specific form the physical action takes is up to the agent designer. For example, when using SRTA with the MASS simulation environ-

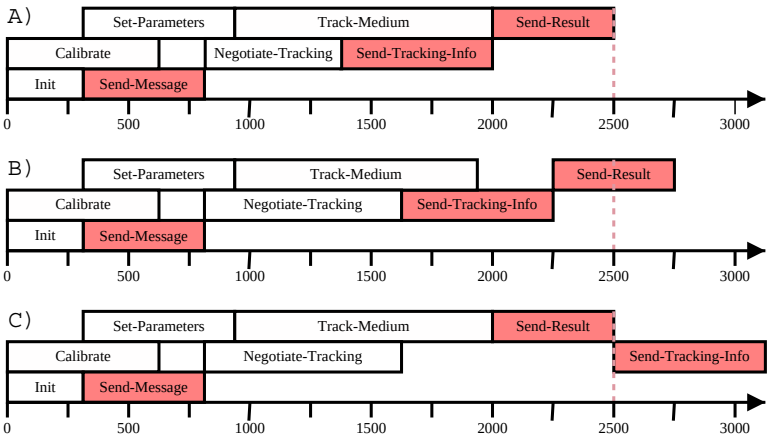


Figure 13: A) Initial schedule produced after all the goals have been received, with a `Send-Result` deadline of 2500, B) the invalid schedule showing that constraint broken by the unexpected long duration of `Negotiate-Tracking`, and C) the corrected schedule respecting the deadline.

ment [40], actions are forwarded to a simulator which uses the quantitative characteristics of the TÆMS structure itself (or some global, objective variant of it) to determine what the appropriate execution characteristics are, and later returns that information to the agent. In real-world systems, methods are performed by some component or object within the agent. 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. Recall that in their barest form, the method specifications used to spawn these actions are limited to just the name of the action and a quantitative description of how it is expected to behave. While these are all the necessary details for scheduling and planning, additional information may also be added which specifies how the action is to be performed in a more concrete sense. Consider the `Send-Results` method. When performed, the problem solving component could implicitly know that it should send measurement data to its track manager. One could also reduce the need for such domain specific code, by using the `attributes` field available in all TÆMS nodes to specify the message data and destination. In this case, a more general `Send-Message` method could include this information, allowing the communication component to perform the action with no additional domain knowledge. A third mechanism allows an arbitrary Java object to be specified by the method, so that the execution subsystem can dynamically create the action code as needed.

In practice, we use a combination of these techniques. 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. The parameters used to control the behavior of the measurement are encoded as attributes in the method. 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.

Once a method has been started, the TÆMS structure is updated to reflect that fact by filling in its start time field. As the method is performed, the execute component monitors its progress by keeping track of how much time has elapsed in comparison to the scheduled completion time. If the method fails to meet its completion time, the execute component will again use the capabilities of the POS to keep the schedule up to date by revising the method's finish time. This may in turn require other methods to be shifted, which can result in the undesirable consequences discussed in the next section.

When the method completes its finish time field is also updated. The specific mechanism employed to perform the task is also responsible for recording the quality and cost of the action. These values are in turn propagated through the graph as needed, correctly taking into account structural details such as task QAFs and interrelationships. For example, the current quality of task with a *sum* QAF will be correctly updated with the sum of its subtasks' qualities. Methods which fail are assigned zero quality. In this way the TÆMS structure is used not only during the planning process, but serves as a record of how the goal is carried out. This quantitative view of the agent's runtime performance is of particular use if rescheduling or conflict resolution is necessary, as we will see in the next section.

3.7 Conflict Resolution

Suppose that the `Negotiate-Tracking` method in Figure 13 is taking longer than expected, forcing the

agent to extend its estimated finish time in the schedule. Because the method `Send-Tracking-Info` cannot start before the completion of `Negotiate-Tracking` due to the *enables* interrelationship shown in Figure 5B, 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 13B.

The partial order scheduler was able to detect this failure, because a simple shifting of the tasks in its dependency graph pushed `Send-Result` past its 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 13C, which delays `Send-Tracking-Info` in favor of `Send-Result`. This allows the agent to proceed successfully despite a failed expectation. The search 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 not attempt to find the best solution, but only guarantees that the minimum set of criteria are met.

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. One solution to this problem is to immediately reschedule the task(s), by re-analyzing them from scratch with DTC and the POS in the current context. While this would resolve the failure, this solution can be costly, may not take advantage of other sources of existing information, and be wasteful if the fault can be equally addressed by simpler techniques - as in the previous example. At the same time, the expensive solution may be the only viable one, so we cannot preclude using it. To handle these cases, we have developed a conflict resolution module capable of analyzing a particular situation and efficiently suggesting appropriate solutions using a range of techniques.

Abstractly, the conflict resolution module is a customizable engine, which applies different techniques 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 is associated with a discrete numeric priority rating specified by the designer, 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 type and ordering of these techniques, along with the domain and environment it is used it, dictates the efficacy of the resolution module. While a completely domain independent configuration can lead to acceptable performance, the design of this system allows the designer's knowledge and experience to easily address the specific faults characteristic to their domain.

Several different types of domain independent conflict resolution techniques have been implemented in this engine. Each uses the existing TÆMS structure and schedule, potentially along with technique-specific data of its own, to generate a new repaired schedule. These are presented below, in no particular order.

1. Prior scheduling results. As mentioned in section 3.3, when DTC generates plans for a task structure, it automatically produces a set of candidate plans, from which the best rated is generally selected for use. A simple and effective resolution tactic is to save these results, and select the best-rated plan not affected by the failure to use that in place of a failed one. This is determined by using the existing rating of the schedule along with the current execution results to ensure it contains no failed methods.

2. Contingency plans. A variant of the previous technique allows the modeler to actually pre-specify an alternate plan, which is then read in and used similarly. This is somewhat more costly, as it involves additional file access and data parsing, so it would likely have a lower priority than the first. However, if a particular class of failure occurs which is difficult for the scheduling process to cope with, this can be an effective strategy.

3. Reschedule. 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 compensate accordingly. 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.

4. Predefined alternatives. A more labor intensive version of the rescheduling 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 could achieve the same results using a different method or remove constraining aspects of the structure, such as hard interrelationships or deadlines. This modified structure would be sent to

DTC for re-scheduling.

5. Reduce expectations. Because DTC is somewhat detached from the global planning process, in that it typically plans for a subset of the 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.

6. Learn appropriate strategies. A particularly efficient resolution technique is to use the knowledge gained from prior resolved conflicts and cache it for later use. In this case, both the resolution strategies which are selected to be applied, and the context in which they were chosen are monitored. Later, when the same context is seen, the earlier solution can be immediately suggested. If this technique 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 techniques are in use by the agent, corresponding to several of the techniques outlined above. At the highest priority level is Learn-Strategy (technique 6), which searches for cached resolution techniques which are applicable to the current problem. At the next level is Alternate-Plan (1), which looks for compatible results from the previous scheduling activity. At the lowest priority level is Regenerate-Plans (3), 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. Learn-Strategy 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 prior execution results in the TÆMS structure show that the plan { `Set-Parameters`, `Track-Low`, `Send-Results` } avoids any failed methods, and still fulfills related commitment criteria. In addition, the updated TÆMS structure indicates this will take advantage of the previously completed `Set-Parameters`. 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 also be obtained from the task structure.

Note that if this problem were seen again, Learn-Strategy would recognize the context and provide this same solution, avoiding further search through the available resolution strategies.

4 Working in Soft Real-Time

The high-level technologies discussed above address the fundamental technologies SRTA employs to run in soft 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 describe several techniques which aim to reduce or account for the overhead of different aspects of the system, in an attempt to avoid such situations.

4.1 Accounting for Meta-Level Costs

There are several causes of such failures in an agent's plans or expectations. 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 entirely separate 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 1: communication, negotiation, problem solving, planning, scheduling and the like. In most systems these so-called "meta-level" activities can constitute a significant portion of the agent's running time without being explicitly accounted for during the scheduling process.

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 of 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 5B, 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 to 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. This has been accomplished in [34] with some of the elements of SRTA, but has not yet been extended to the entire architecture. 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 select and 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 circumstances may require a more sophisticated approximation.

4.2 Plan Caching

An issue affecting the agent's soft 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.

Planning is a particularly computationally expensive process for our agents, because of the potentially large range of alternative plans which DTC evaluates during the course of its analysis. We have 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, which results in very similar plan selections. The frequency of this behavior is increased by the fact that DTC generally plans for only one goal at a time, because of the incremental nature of goal addition to SRTA and the fact that other components are responsible for plan merging. To avoid this potentially repeated work, a plan caching system was implemented, shown as a bypass flow in Figure 1. Each task structure to be sent to DTC is used to generate a key, incorporating several distinguishing characteristics of the structure such as the method names, durations, start and finish times, outcome characteristics and interrelationships. 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 1. To test the caching subsystem, we performed 1000 runs using eight sensors and one target in the RADSIM environment [29], 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 with only a third of the original time cost. Related results from the POS scheduling are included for comparison purposes.

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.[3] 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, although it is not clear that adding this additional complexity would not result in a technique with unacceptably high combinatorics. Additional changes would also need to be made to support a more robust model of resource usage, to make use of the information provided by the resource modeling component. After consideration, it was determined that the amount of effort needed to do this would outweigh the benefits, as described in section 3.4. 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.

[3] 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 [47].

Component	Average Number of Calls	Average Time Per Call
DTC Scheduler	72.14	300 ms
DTC Caching	31.12	74 ms
Partial Order Scheduler	531.03	36 ms

Table 1: Results using plan caching over 1000 experimental runs in the DSN domain.

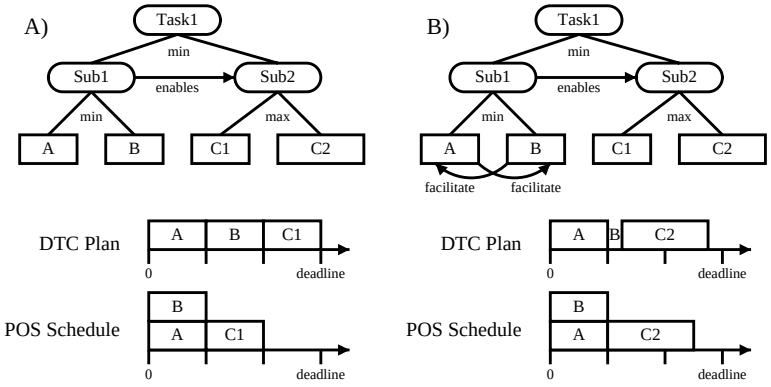


Figure 14: The effects of parallel activity recognition. A) shows the original task structure, which leads to an inefficient final schedule, B) shows the task structure with modifications, which results in a higher quality schedule.

The process is best explained through an example. Consider the abstract task structure shown in Figure 14A. 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 14A 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 14B, 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 [12]. 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 7. 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 Task Characteristics

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 [21], 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 run-

time 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 is accomplished by binning the known duration results, and associating a weight with each bin value based on the relative proportion of actions that exhibited that duration. 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 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.

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 1185. In the subsequent cycle, 24 milliseconds have passed, and it is now time 1205. 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 [8, 9], 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

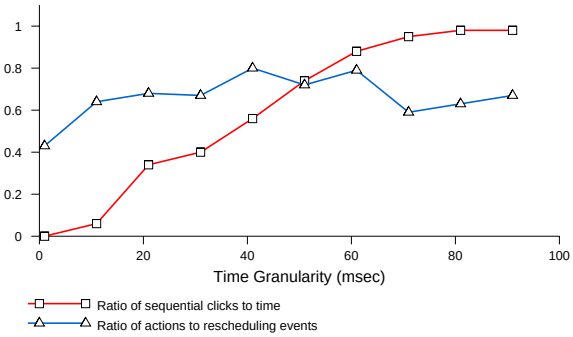


Figure 15: The effect of varying time granularity on agent behavior. A higher time ratio indicates that a greater percentage of sequential time units are seen, which should reduce the need for rescheduling. A higher action ratio indicates the available time was used more efficiently.

every agent tick corresponds to 20 milliseconds, the above action originally scheduled for time 1200 would actually be mapped to run at the agent's time 60 ($1200/20 = 60$). The schedule is first checked at the real-world time of 1185 – in the agent's time scale this would actually be time 59 ($1185/20 = 59$), and it will still be too early to run `X`. However, real-world time 1205 becomes the agent's time 60, the correct scheduled time for `X`, thus avoiding the need to shift the action.

Clearly we can not eliminate the need for rescheduling, due to the inherent uncertainty in action duration in this environment – the hope is to reduce the frequency it is needed. Experimentation can find the most appropriate scaling factor for an agent running on a particular system, by searching for the granularity which optimizes the number of actions which are able to be performed against the number of rescheduling events which must take place. Our experiments, the results of which can be seen in Figure 15, compared a range of granularities from 0 ms (time is unchanged) to 90 ms. These tests showed 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 have a large enough granularity so it can "see" each sequential time click, while avoiding a timeline that is so coarse that the number of actions which may take place is unnecessarily reduced.

5 SRTA in Practice

This section will provide more examples of how the SRTA architecture is used to solve problems 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.

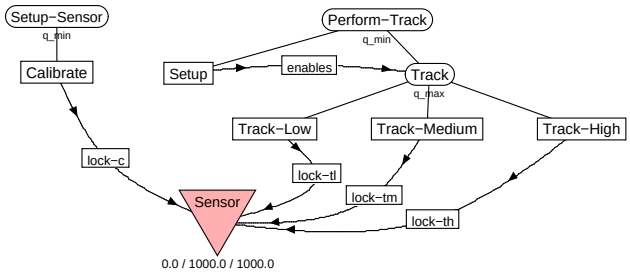


Figure 16: A pair of abbreviated task structures for calibration and tracking.

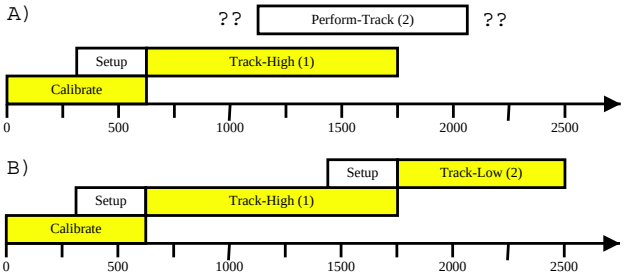


Figure 17: A) The agent's initial schedule, along with new actions which describe the what-if condition, been received, B) The resulting consolidated schedule.

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.

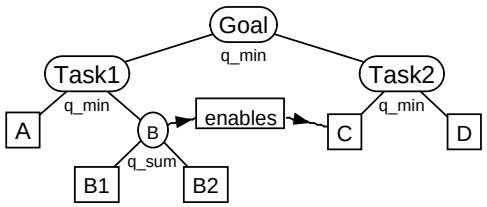
Using TÆMS, the agent will first model the goal and its subtasks, along with any constraints that exist. Consider the schedule shown in Figure 17A. In this scenario, the agent has previously scheduled two goals, `Setup-Sensor` and `Perform-Track`, as modeled in Figure 16. The three `Track` methods in that model each have an expected level 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 17A. Because of the deadline and the preexisting schedule, SRTA selects `Track-Low` to satisfy that goal, as shown in 17B. 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 [12]. 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.

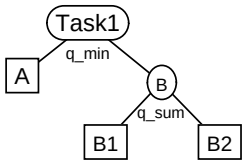
5.2 Modeling and Respecting Commitments

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.

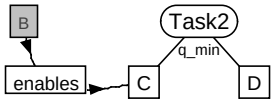
Abstractly, a commitment is usually formed when the actions (or their results) of one agent may directly or indirectly affect the state of another. We have previously shown in section 3.1 how interrelationships between nodes in a single structure can model the effects between them. These types of effects may be simply extended to span nodes between structural elements belonging to disparate agents to model inter-agent effects. Refer to Figure 18A, 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



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

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 18A. Instead, each agent would have its own local view of the problem, as is seen in 18B 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 [25].

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 in [23, 7]. 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 18 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 an example, we 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 means to model 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 19, 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 19 are shown in Table 2. 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.[4]

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 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 2 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 [49].

An additional example of adaptation taken from the distributed sensor domain is shown in Figure 20. The architecture we have developed to address this domain uses a notion of periodic commitments along a discrete

[4] 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.

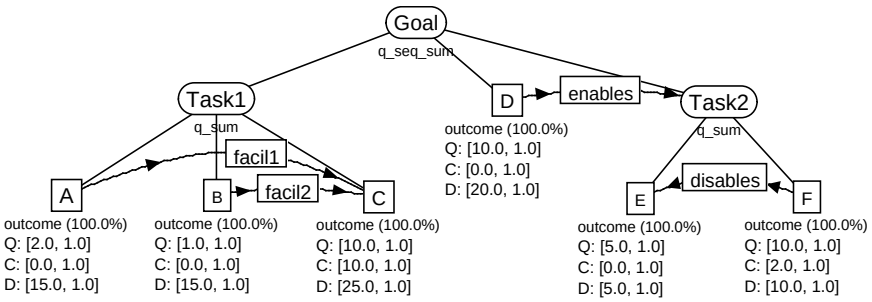


Figure 19: A TÆMS task structure modeling several different ways to achieve the same goal.

	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 2: A variety of schedules, and their expected qualities, costs and durations, generated from the TÆMS structure in Figure 19 under different conditions.

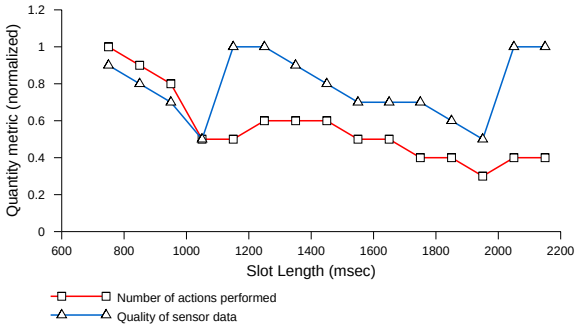


Figure 20: Monitoring track measurement quality under different temporal conditions.

timeline to reduce negotiation complexity. Specifically, 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 20 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 [33], we cannot make performance guarantees [38] about agent control. SRTA differs in that its action primitives are permitted to have unpredictable performance results across several dimensions, and it uses a more complex model of resources during the scheduling phase. We have also seen empirically that the soft real-time model SRTA employs addresses the requirements of the applications it has been applied to, despite the lack of performance guarantees. 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 [20] and the more recent work on UMPRS [22] 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 [3].

Like the Remote Agent (RA) architecture [32], SRTA uses a layered approach to consolidate particular functionalities within discrete components. It employs a planner/scheduler component, plan library, has resource models and an execution subsystem that performs monitoring and is capable of selecting alternatives in the case of failure. RA's execution system allows planning to occur concurrently, and supports multiple parallel actions as SRTA does, but it operates in a separate thread which likely allows for more fine grained control of action initiation. Unlike RA, SRTA accomplishes this with a single modeling language, TÆMS, which is shared by all components. RA's successor, IDEA [31] resolves this issue, thereby reducing the overlapping effort otherwise needed to provide plans to the engine, although SRTA is still differentiated by the ability of TÆMS to model uncertainty, soft interrelationships, and a range of commitment types. RA's model identification and recovery system is analogous to SRTA's conflict resolution model, and also to work we have done separately involving a true diagnosis module [12], although RA uses a model-based system, as compared to SRTA's more heuristic approach.

3T [1] also uses a layered architecture, including a planner and sequencer. In addition to activities which are described with traditional plans, 3T's domain, primarily robotics, also contains a large suite of actions which, although they may be complex, do not require active planning. To support this, 3T offers RAPs, an additional reactive task description below what SRTA considers to be primitive actions. SRTA differs from 3T model in its quantitative use of probability and uncertainty to describe the results and effects of activities during planning and scheduling, although 3T's planning language offers support for more concrete preconditions and effects.

Our work also relates to [53], 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 [11] and associated DRU scheduler [10] 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 [50], 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 flexible and efficient 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. We have successfully used SRTA to address the challenges posed by the distributed sensor network domain, which exhibit such real-time, dynamic characteristics.

SRTA's objective is to provide domain independent functionality enabling the 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 the complex modeling, planning and scheduling that has been described in our prior research, in soft real-time. 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 [52] 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-level reasoning component would allow the agent to directly decide how much effort to allocate towards scheduling, conflict resolution, and execution, and then pass that information to the appropriate components so that they can bound their computations appropriately. This type of direct accounting for meta-level activities would better equip the agent to meet strict deadlines. Recent work in this area in an architecture conceptually similar to SRTA [35] has shown these techniques to be possible, and we hope to integrate this functionality into SRTA.

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. Peter Bonasso, James Firby, Erann Gat, David Kortenkamp, David P. Miller, and Marc G. Slack. Experiences with an architecture for intelligent, reactive agents. volume 9, pages 237–256, April 1997.

[2] 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, July 2002.

[3] 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.

[4] 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.

[5] K. Decker and V. Lesser. A One-Shot Dynamic Coordination Algorithm for Distributed Sensor Networks. *Proceeding of the Eleventh National Conference on Artificial Intelligence*, pages 210–216, January 1993.

[6] 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".

[7] 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.

[8] 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.

[9] 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.

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

[11] John R. Graham, Keith S. Decker, and Michael Mersic. DECAF - a flexible multi agent system architecture. *Autonomous Agents and Multi-Agent Systems*, 2003.

[12] Bryan Horling, Brett Benyo, and Victor Lesser. Using Self-Diagnosis to Adapt Organizational Structures. *Proceedings of the 5th International Conference on Autonomous Agents*, pages 529–536, June 2001.

[13] 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.

[14] Bryan Horling, Victor Lesser, Regis Vincent, Ana Bazzan, and Ping Xuan. Diagnosis as an Integral Part of Multi-Agent Adaptability. *Proceedings of DARPA Information Survivability Conference and Exposition (see also UMASS CSTR 1999-03)*, pages 211–219, January 2000.

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

[16] Bryan Horling, Roger Mailler, Jiaying Shen, Regis Vincent, and Victor Lesser. Using Autonomy, Organizational Design and Negotiation in a Distributed Sensor Network. In Victor Lesser, Charles Ortiz, and Milind Tambe, editors, *Distributed Sensor Networks: A multiagent perspective*, pages 139–183. Kluwer Academic Publishers, 2003.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] V. Lesser, K. Decker, T. Wagner, N. Carver, A. Garvey, B. Horling, D. Neiman, R. Podorozhny, M. NagendraPrasad, A. Raja, R. Vincent, P. Xuan, and X.Q Zhang. Evolution of the GPGP/TAEMS Domain-Independent Coordination Framework. *Proceedings 1st International Conference on Autonomous Agents and Multi-Agent Systems (Plenary Lecture/Extended Abstract)*, pages 1–2, 2002.

[24] Victor Lesser. A Retrospective View of FA/C Distributed Problem Solving. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(6):1347–1363, November 1991.

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

[26] 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.

[27] Victor Lesser, Bryan Horling, Frank Klassner, Anita Raja, Thomas Wagner, and Shelley Zhang. BIG: An Agent for Resource-Bounded Information Gathering and Decision Making. *Artificial Intelligence Journal, Special Issue on Internet Information Agents (also available as UMass Computer Science Technical Report 1998-52)*, 118(1-2):197–244, May 2000.

[28] 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.

[29] Victor Lesser, Charles Ortiz, and Milind Tambe. *Distributed Sensor Networks: A multiagent perspective.* Kluwer Publishers, 2003.

[30] 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.

[31] Nicola Muscettola, Gregory A. Dorais, Chuck Fry, Richard Levinson, and Christian Plaunt. Idea: Planning at the core of autonomous reactive agents. 2002.

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

[33] 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.

[34] Anita Raja and Victor Lesser. Automated Meta-Level Control Reasoning in Complex Agents. *Proceedings of workshop on 'Agents and Automated Reasoning' in the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI 2003)*, August 2003.

[35] Anita Raja and Victor Lesser. Efficient Meta-Level Control in Bounded Rational Agents. *Proceedings of 2nd International Conference on Autonomous Agents and Multiagent Systems*, pages 1104–1105, July 2003.

[36] Anita Raja, Victor Lesser, and Thomas Wagner. Toward Robust Agent Control in Open Environments. In *Proceedings of the Fourth International Conference on Autonomous Agents (Agents2000)*, 2000.

[37] 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.

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

[39] Katia Sycara, Steven F Roth, Norman Sadeh, and Mark S Fox. Resource allocation in distributed factory scheduling. *IEEE Expert*, 6(1):29–40, February 1991.

[40] Regis Vincent, Bryan Horling, and Victor Lesser. An Agent Infrastructure to Build and Evaluate Multi-Agent Systems: The Java Agent Framework and Multi-Agent System Simulator. *Lecture Notes in Artificial Intelligence: Infrastructure for Agents, Multi-Agent Systems, and Scalable Multi-Agent Systems.*, 1887, January 2001.

[41] 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.

[42] 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.

[43] 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.

[44] Thomas Wagner, Valerie Guralnik, and John Phelps. A key-based coordination algorithm for dynamic readiness and repair service coordination. In *Proceedings of the 2nd International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS2003), 2003.*, 2003.

[45] Thomas Wagner, Valerie Guralnik, and John Phelps. Software Agents: Enabling Dynamic Supply Chain Management for a Build to Order Product Line. *International Journal of Electronic Commerce Research and Applications, Special issue on Software Agents for Business Automation*, 2(2):114–132, 2003.

[46] Thomas Wagner, Valerie Guralnik, John Phelps, and Ryan VanRiper. TÆMS Agents for Crisis Response Decision Support. in progress, 2003.

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

[48] Thomas Wagner and Victor Lesser. Design-to-Criteria Scheduling: Real-Time Agent Control. In Wagner/Rana, editor, *Infrastructure for Agents, Multi-Agent Systems, and Scalable Multi-Agent Systems*, LNCS. Springer-Verlag, 2001. Also appears in the 2000 AAAI Spring Symposium on Real-Time Systems and a version is available as University of Massachusetts Computer Science Technical Report TR-99-58.

[49] Thomas Wagner, Anita Raja, and Victor Lesser. Modeling Uncertainty and its Implications to Design-to-Criteria Scheduling. Computer Science Technical Report TR 1998-51, University of Massachusetts, December 1999.

[50] XQ Zhang, V. Lesser, and S. Abdallah. Efficient Ordering and Parameterization of Multi-Linked Negotiation. *Proceedings 2nd International Joint Conference on Autonomous Agents and Multiagent Systems. (Extended abstract)*, AAMAS03:1170–1171, July 2003.

[51] X.Q. Zhang, A. Raja, B. Lerner, V. Lesser, L. Osterweil, and T. 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, 2000.

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

[53] Shlomo Zilberstein and Abdell-Illah Mouaddib. Optimal scheduling of progressive processing tasks. *International Journal of Approximate Reasoning*, 25(3):169–186, 2000.

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

[55] 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.