The Soft Real-Time Agent Control Architecture *

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

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

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

Abstract

Real-time control has become increasingly important as technologies are moved from the lab into real world situations. The complexity associated with these systems increases as control and autonomy are distributed, due to such issues as precedence 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 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 conflict resolution in an uncertain environment. Technologies which facilitate on-line real-time control, including schedule caching and variable time granularities are also discussed.

Overview

In the field of multi-agent systems, much of the research and most of the discussion focuses on the dynamics and interactions between agents and agent groups. Just as important, however, is the design and behavior of the individual agents themselves. The efficiency of an agent's internal mechanics contribute to the foundation of the system as a whole, and the degree of flexibility these mechanics offer affect the agent's achievable level of sophistication, particularly in its interactions with other agents (Lesser 1991; 1998). We believe that a general control architecture, responsible for both the planning for the achievement of temporally constrained goals of varying worth, and the sequencing of actions local to the agent that have resource requirements, can provide a robust and reusable platform on which to build high level reasoning components. In this article, we will discuss the design and implementation of the Soft Real Time Architecture (SRTA), a generic planning, scheduling and execution subsystem designed to address these needs.

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

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

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

*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.

unexpected events take place. Together, this system can assume responsibility for the majority of the goal-satisfaction process, which allows the high-level reasoning system to focus on goal selection, determining goal objectives and other potentially domain-dependent issues. For example, agents may elect to negotiate over an abstraction of their activities or resource allocations, and only locally translate those activities into a more precise form (Mailler *et al.* 2001). SRTA can then use this description to both enforce the semantics of the commitments which were generated, and automatically attempt to resolve conflicts that were not addressed through negotiation.

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

In recent work on a distributed sensor network application(Horling *et al.* 2001), which will be discussed in more depth below, we exploited SRTA to create a virtual agent organization of simple, single-threaded agents. These "virtual" agents were in fact goals, created as needed and dynamically assigned to a specific sophisticated "real" agent based on information approximating the current resource usage of agents and the type of resources available at each agent. The "real" agent then performed detailed planning/scheduling based on local resource availability and priority of these goals, and multiplexed among the different goals that it was concurrently executing in order to meet soft real-time requirements.

SRTA operates as a functional unit within a Java-based agent, which itself it running on a conventional (i.e. not real-time) operating system. The SRTA controller is designed to be used in a layered architecture, occupying a position below the high-level reasoning component in an agent (Zhang *et al.* 2000; Bordini *et al.* 2002) (see Figure 2). In this role, it will accept new goals, report the results of the activities used to satisfy those goals, and also serve as a knowledge source about the potential ability to schedule future activities by answering what-if style queries. Within this context, SRTA offers a range of features designed to provide support for operating in a distributed, intelligent environment. The goal description language supports quantitative, probabilistic models of activities, including non-local effects of actions and resources and a variety of ways to define how tasks decompose into subtasks. In particular, the uncertainty associated with activities can be directly modeled through the use of quantitative distributions describing the different outcomes a given action may produce. Commitments and constraints can be used to define relationships and interactions formed with other agents, as well as internally generated limits and deadlines. The planning process uses this information to generate a number of different plans, each with different characteristics, and ranked by their predicted utility. A plan is then used to produce a schedule of activities, which is combined with existing schedules to form a potentially parallel sequence of activities, which are partially ordered based on their interactions with both resources and one another. This sequence is used to perform the actions in time, using the identified preconditions to verify if actions can be performed, and invoking light-weight rescheduling if necessary. Finally, if conflicts arise, SRTA can use an extendable series of resolution techniques to correct the situation, in addition to passing the problem to higher level components which may be able to make a more informed decision.

An important aspect of most real-world systems is their ability to handle real-time constraints. This is not to say that they must be fast or agile (although it helps), but that they should be aware of deadlines which exist in their environment, and how to operate such that those deadlines are reasoned about and respected as much as possible. This notion of real-time is weaker than its relative, strict real-time, who's adherents attempt to rigorously quantify and formally bound their systems' execution characteristics. Instead, systems working in soft real-time operate on tasks which may still have value for some period after their deadlines have passed (Stankovic & Ramamritham 1990), and missing the deadline of a task does not lead to disastrous external consequences. Our research addresses a derivative of this concept, where systems are expected to be statistically fast enough to achieve their objectives, without providing formal performance guarantees. This allows it to successfully address domains with highly uncertain execution characteristics and the potential for unexpected events, neither of which are well suited for a hard real-time approach. As its name implies, SRTA operates in soft real-time, using time constraints specified during the goal formulation and scheduling processes, and acting to meet those deadlines whenever necessary. In this system, we have sacrificed the ability to provide formal performance guarantees in order to address more complex and uncertain problem domains. As will be shown shortly, this technology has been used to successfully operate in a real-time distributed environment.

To operate in soft-real time, an agent must know when actions should be performed, how to schedule its activities and commitments such that they can be performed

or satisfied, and have the necessary resources on hand to complete them. Our solution to this problem addresses two fronts. The first is to implement the technologies needed to directly reason about real-time. As mentioned above, we begin by modeling the quantitative and relational characteristics of the goals and activities the agent may perform, which can be done a priori and accessed as plan library or through a runtime learning process(Jensen *et al.* 1999). This information is represented, along with other goal achievement and alternative plan information, in a TÆMS task structure (Decker & Lesser 1993; Horling *et al.* 1999) (discussed in more detail later). A planning component, the Design-to-Criteria scheduler (DTC) (Wagner, Garvey, & Lesser 1998; Wagner & Lesser 2000), uses these TÆMS task structures, along with the quantitative knowledge of action interdependence and deadlines, to select the most appropriate plan given current environmental conditions. This plan is used by the Partial Order Scheduler process to determine when individual actions should be performed, either sequentially or in parallel, within the given precedence and runtime resource constraints. In general, we feel that real-time can be addressed through the interactions of a series of components, operating at different granularities, speed and satisficing (approximate) behaviors.

The second part of our solution attempts to optimize the running time of our technologies, to make it easier to meet deadlines. The partial order schedule provides an inherently flexible representation. As resources and time permit, elements in the schedule can be quickly delayed, reordered or parallelized. New goals can also be incorporated piecemeal, rather than requiring a computationally expensive process involving re-analysis of the entire schedule. Together, these characteristics reduce the need for constant re-planning, in addition to making the scheduling process itself less resource-intensive. Learning plays an important role in the long-term viability of an agent running in real time, taking advantage of the repetitive nature of its activities. Schedules may be learned and cached, eliminating the need to reinvoke the DTC process when similar task structures are produced, and the execution history of individual actions may be used to more accurately predict their future performance. A similar technique could be used to track the requisite actions and time needed to devote to particular goals.

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. We will conclude with a more description of SRTA's ability to adapt to varying conditions, and summarize the significant characteristics of the architecture. Note also that a more thorough overview of this architecture can be found in (Horling *et al.* 2002).

Problem Domain

A distributed resource allocation domain which motivated much of this work (Horling *et al.* 2001) will be used throughout this article to ground the topics which are discussed and formulate examples. This section will briefly describe the environment and the particular challenges it offers. Components of the SRTA architecture have also been used successfully in several other domains, such as intelligent information gathering (Lesser *et al.* 2000), intelligent home control (Lesser *et al.* 1999), and supply chain (Horling, Benyo, & Lesser 2001).

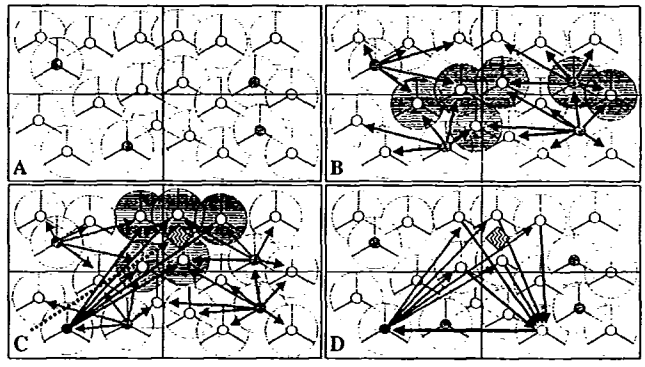


Figure 1: High-level distributed sensor allocation architecture. A) shows the initial sensor layout, decomposition and allocation of sector managers. B) shows the dissemination of scanning tasks. The new track manager in C) can be seen coordinating with sensors to track a target, while the resulting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

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

The real-time requirement of this environment is derived from the triangulation process. Under ideal conditions, three or more sensors will perform measurements at the same instant in time. Individually, each sensor can only determine the target's distance and velocity relative to itself. Because each node will have seen the target at the same position, however, these gathered data can then be fused to triangulate the target's actual location. In practice, exact synchronization to an arbitrarily

high resolution of time is not possible, due to the uncertainty in sensor performance and clock synchronization. A reasonable strategy then is to have the sensors perform measurements within some relatively small window of time, which will yield positive results as long as the target is near the same location for each measurement. Thus, the viable length of this window is inversely proportional to the speed of the target (in our scenarios we use a window of one second for a target moving one foot per second).

Competing with the sensor measurement activity are a number of other local goals, including sector management (Figure 1A), target discovery scanning (1B), measurement tasks for other targets (1C), and data processing (1D). We don't see these as separate agents or threads, but rather as different objectives/goals that an agent is multiplexing. To operate effectively, while still meeting the deadlines posed above, the agent must be capable of reasoning about and acting upon the importance of each of these activities.

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

Further details on this domain and the multi-agent architecture designed to address it can be found in (Horling *et al.* 2001).

Real-Time Control Architecture

Our previous agent control architecture, used exclusively in controlled time environments, was fairly large grained. As goals were addressed by the problem solving component, they would be used to generate task structures to be analyzed by the Design-To-Criteria (DTC) scheduler. The resulting linear schedule would then be directly used for execution by the agent. Task structures created to address new goals would be merged with existing task structures, creating a monolithic view of all the agent's goals. This combined view would then be passed again to DTC for a complete re-planning and re-scheduling. Execution failure would also lead to a complete re-planning and 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 a later section, 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

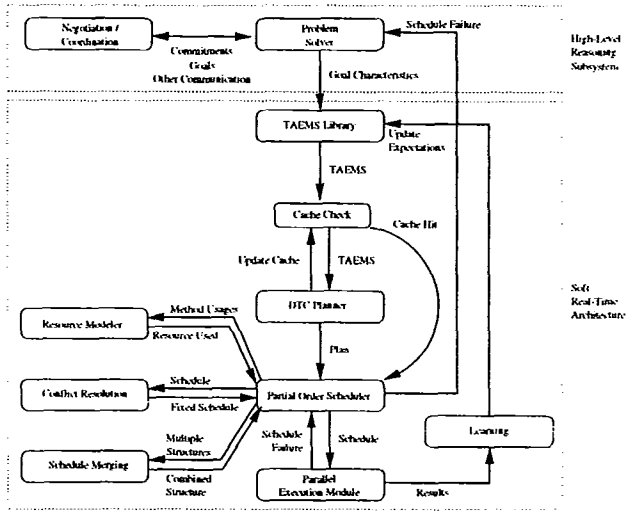


Figure 2: High-level agent control architecture.

the need to constantly perform the relatively expensive DTC process. In a real-time environment, characterized by a lot of uncertainty in the timing of actions and the arrival of new tasks, where the agent must constantly reevaluate their execution schedule in the face of varied action characteristics, this sort of control architecture was impractical.

In the SRTA architecture, we have attempted to make the scheduling and planning process more incremental and compartmentalized. New goals can be added piecemeal to the execution schedule, without the need to replan all the agent's activities, and exceptions can be typically be handled through changes to only a small subset of the schedule. Figure 2 shows the new agent control architecture we have developed to meet our soft real-time needs. We will first present an overview of how it functions, and cover the implementation in more detail in later sections. In this architecture, goals can arrive at any time, in response to environmental change, local planning, or because of requests from another agents. The goal is used by the problem solving component to generate a TÆMS task structure, which quantitatively describes the alternative ways that goal may be achieved. The TÆMS structure can be generated in a variety of ways; in our case we use a TÆMS "template" library, which we use to dynamically instantiate and characterize structures to meet current conditions. Other options include generating the structure directly in code (Lesser *et al.* 2000), or making use of an approximate base structure and then employing learning techniques to refine it over time (Jensen *et al.* 1999).

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

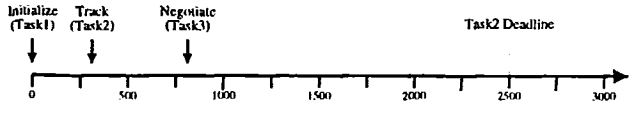


Figure 3: The timeline of events for our running scenario. Shown are the arrival times for the goals shown in Figures 4 and 5, along with the negotiated deadline for Task 2.

minimum quality, and external commitments, DTC selects an appropriate plan.

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

Once the schedule has been created, an execution module is responsible for initiating the various actions in the schedule. It also keeps track of execution performance and the state of actions' preconditions, potentially re-invoking the partial order scheduler when failed expectations require it. As will be shown later, the partial order scheduler uses a shifting mechanism to resolve such failures with minimal overhead where possible.

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

TÆMS Generation

Before progressing, we must provide some background on our task description language, TÆMS. TÆMS, the Task Analysis, Environmental Modeling and Simulation language, is used to quantitatively describe the alternative ways a goal can be achieved (Decker & Lesser 1993; Horling *et al.* 1999). A TÆMS task structure is essentially an annotated task decomposition tree. The highest-level nodes in the tree, called task groups, represent

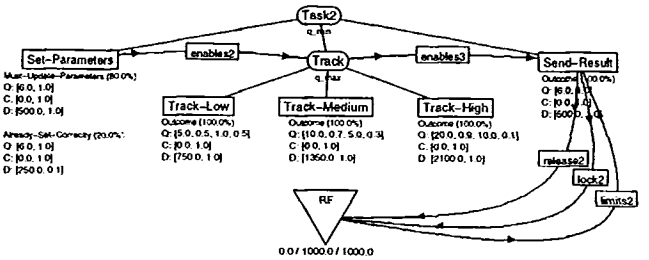


Figure 4: An example TÆMS task structure for tracking. The expected execution characteristics are shown below each method, and the `Send-Results` method in this figure has a deadline of 2500.

goals that an agent may try to achieve. The goal of the structure shown in Figure 4 is `Task2`. Below a task group there will be a set of tasks and methods which describe how that task group may be performed, including sequencing information over subtasks, data flow relationships and mandatory versus optional tasks. Tasks represent sub-goals, which can be further decomposed in the same manner. `Task2`, for instance, can be performed by completing subtasks `Set-Parameters`, `Track`, and `Send-Results`.

Methods, on the other hand, are terminal, and represent the primitive actions an agent can perform. Methods are quantitatively described, with probabilistic distributions of their expected quality, cost and duration. These quantitative descriptions are themselves grouped together as outcomes, which abstractly represent the different ways in which an action can conclude. cost incurred. `Set-Parameters`, then, is described with two potential outcomes, `Must-Update-Parameters` and `Already-Set-Correctly`, each with its relative probability and description of expected duration.

The quality 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.

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

`lock2` and `release2` are resource interrelationships, describing, in this case, the *consumes* and *produces* effects method `Send-Results` has on the resource `RF`.

These indicate that when the method is activated, it will consume or produce some quantity of that resource. The resource effect is further described through the *limits* interrelationship, which defines the changes in the method's execution characteristics when that resource is over-consumed or over-produced. The resource itself is also modeled, including its bounds and current value (as shown below the RF triangle), and whether it is consumable or not (e.g. printer paper is consumable, where the printer itself is not).

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

The problem solver is responsible for translating its high-level goals into TÆMS, which serves as a more detailed representation usable by other parts of the agent. This can be done by building TÆMS structures dynamically at runtime, reading static structures from a library, or using a hybrid scheme consisting of a library of template structures which can be annotated at runtime.

At time 0 the agent will use its template library to generate the initialization structure seen in Figure 5A. In this structure, the agent must first `Init` and then `Calibrate` its sensor. Properties passed into the template specify the particular values used in `Init`, such as the sensor's desired gain settings or communications channel assignment, as well the number of measurements to be used during `Calibrate`. As specified by the *enables* interrelationship, `Init` must successfully complete before the agent can `Send-Message`, reporting its capabilities to its local manager. `Send-Message` also uses resource interrelationships to obtain an exclusive lock on the RF communication resource. Only one action at a time can use RF to send message, so all messaging methods have similar locking interrelationships. As we will see later, this indirect interaction between messaging methods creates interesting scheduling problems. `Task2` and `Task3`, shown in Figures 4 and 5B, respectively, are generated later in the run in a similar manner.

DTC Planner / Initial Scheduler

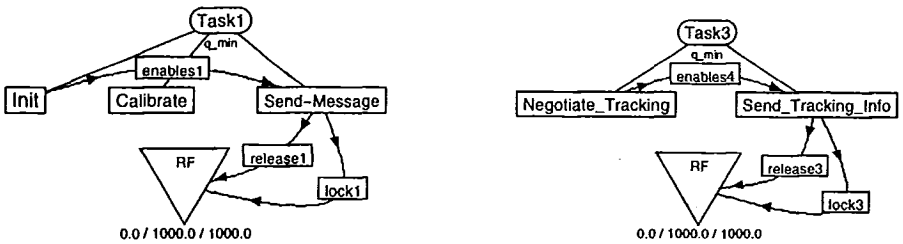
Design-to-Criteria (DTC) scheduling is the soft real-time process of evaluating different possible courses of action for an intelligent agent and choosing the course that best fits the agent's current circumstances. For example, in a situation where the RF resource is under a great deal of concurrent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

One would expect any reasonable planning process to enforce so-called "hard" constraints - ones which must be satisfied for a goal to be achieved or a commitment satisfied. It is DTC's additional ability to reason about weaker, optional interactions which sets it apart. The *sum* QAF in TÆMS , for instance, defines a task who's quality is determined by the sum of all its subtasks' qualities. In a time critical situation, DTC may opt for a shorter, but lower quality plan which only calls for one of these subtasks to be executed. In more relaxed conditions, more may be added to the plan. Similarly, soft interrelationships such as *facilitates* or *hinders* may be respected or not, depending on their specific quantitative effects and the current planning context. DTC's behavior is governed through the use of a criteria description, which is provided to it along with each TÆMS structure. This criteria specifies, for example, the desired balance between plan quality and duration, or what level of uncertainty is tolerable(Wagner, Garvey, & Lesser 1997). More information covering the techniques DTC uses can be found in (Wagner, Garvey, & Lesser 1998).

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 (Wagner & Lesser 2000). 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.

Partial Order Scheduler

DTC was designed for use in both single agents and agents situated in multi-agent environments. Thus, it makes no assumption about its ability to communicate with other agents or to "force" coordination between agents. This design approach, however, leads to complications when working in a real-time, multi-agent environment where distributed resource coordination is an issue. When resources can be used by multiple agents at the same time, DTC lacks the ability to request communication for the development of a resource usage model. This is the task of another control component that forms



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

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

scheduling constraints based on an understanding of resource usage. In most applications, these constraints are formed by rescheduling to analyze the implications of particular commitments. In the real-time sensor application, the rescheduling overhead is too expensive for forming these types of relationships. The solution we have adopted is to use a subset of DTC's functionality, and then offload the distributed resource and fine grained scheduling analysis to a different component - the partial order scheduler. Specifically, DTC is used in this architecture to reason about tradeoffs between alternative plans, respect ordering relationships in the structure, evaluate the feasibility of soft interactions, and ensure that hard duration, quality and cost constraints are met.

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

Consider the tracking task structure shown in Figure 4. *Enables* interrelationships between the tasks and methods indicate a strict ordering is necessary for the three activities to succeed. In addition (although not shown in the figure), a deadline constraint exists for `Send-Result`, which must be completed by time 2500. Next look at the initialization structure in Figure 5A. While an *enables* interrelationship orders `Init` and `Send-Message`, it does not affect the `Calibrate` method. Internally, the partial order scheduler will use this information to construct a precedence graph. In this example, the graph will first be used to determine that `Calibrate` may be run in parallel with the other two methods in its structure. Later, when `Task2` arrives, the updated graph can be used to find an appropriate starting time for `Set-Parameters` which still respects the deadline of `Send-Result`.

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

Resource Modeler

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

As with most elements in TÆMS the resource usage is probabilistically described, so the scheduler must also provide a minimum desired chance of success to the modeler. At any potential insertion point, the modeler computes the aggregate affects of the new resource usage, along with all prior usages up to the last known actual value of the resource. The expected usage for a given time slot can become quite uncertain, as the probabilistic usages are carried through from each prior slot. If the probability of success for this aggregate usage lies above the range specified by the scheduler, then the resource modeler assumes the usage is viable at that point. Since a given usage may actually take place over a range of time, this check is performed for all other points in that range as well. If all points meet the success requirement, the resource modeler will return the

valid point in time. After this, the scheduler will insert the usage into the model, which will then be taken into account in subsequent searches. If a particular point in time is found to be incompatible, the resource modeler continues its search by looking at the next "interesting" point on its timeline - the next point at which a resource modification event occurs. The search process becomes much more efficient by moving directly from one potential time point to the next, instead of checking all points in between, making the search-time scale with the number of usage events rather than the span of time which they cover. Caching of prior results, especially the results of the aggregate usage computation, is also used to speed up the search process.

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

Schedule Merging

Once potential interactions, through interrelationships, deadlines or resource uses are determined, the partial order scheduler can evaluate what the best order of execution is. Wherever possible, actions are parallelized to maximize the flexibility of the agent, as was shown earlier. 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 partial order scheduler makes use of two other technologies to integrate the new goal with existing scheduled tasks. The first is a conflict resolution module, which determines how best to handle un-schedulable conflicts, given the information at hand. A second component handles the job of merging the new goal's schedule with those of prior goals. The specific mechanism used is identical to that which determines order of execution. Interdependencies between this large set of methods, either direct or indirect, are used to determine which actions can be performed relative to one another. This information is then used to determine the final desired order of execution.

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,

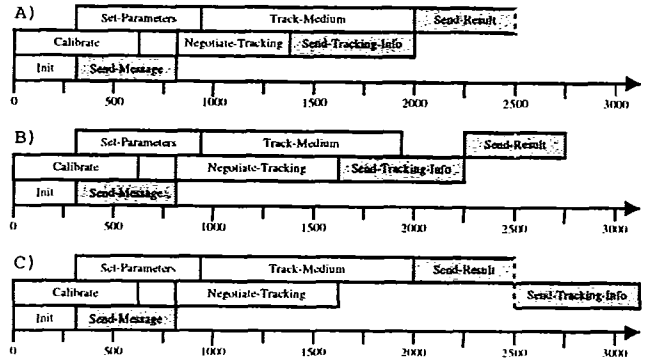


Figure 6: A) Initial schedule produced after all the goals have been received, with a `Send-Result` deadline of 2500, B) the invalid schedule showing that constraint broken by the unexpected long duration of `Negotiate-Tracking`, and C) the corrected schedule respecting the deadline.

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 6A, 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.

Conflict Resolution

Suppose next that `Negotiate-Tracking` is taking longer than expected, forcing the agent to dynamically reschedule its actions. Because the method `Send-Tracking-Info` cannot start before the completion of `Negotiate-Tracking`, due to the *enables* interrelationship shown in Figure 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 in Figure 6B.

Fortunately, the partial order scheduler was able to detect this failure, because of the propagation of execution windows. `Send-Result` was marked with a latest start time of 2000. This caused the scheduler to try other permutations of methods, which resulted in the schedule shown in Figure 6C, which delays `Send-Tracking-Info` in favor of `Send-Result`. This allows the agent to proceed successfully despite a failed expectation. This process is accomplished by first delaying the finish time of the offending method in the schedule to reflect the current state of affairs, and then recursively delaying any other methods which are dependent on that method until a valid solution is found or a recursive limit is reached.

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, particularly when methods actually fail to achieve their goal, this sort of simple shifting is not sufficient to repair the problem. To handle these cases, we designed a conflict resolution module capable of analyzing a particular situation and suggesting solutions.

Abstractly, the conflict resolution module is a customizable engine, which applies techniques encoded as "plug-ins" to a particular situation. If the set of techniques available is not appropriate for the agent designer, they are free to add or remove them as needed. Each technique plug-in is associated with a discrete numeric priority rating, typically specified by the designer of the plug-in, which controls the ordering in which the techniques are applied. When searching for a conflict resolution, the engine will begin by applying all techniques marked with the highest priority. If one or more solutions are suggested, then that set of solutions is returned for the caller to select from. If no solutions are suggested, the engine will apply the techniques at the second-to-highest level, and so on. If the techniques are ordered appropriately, such as with quick or effective techniques first followed by slower or less applicable ones, the engine can make efficient use of its time.

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

Optimizations

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

Plan Caching

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

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

Component	Average Calls	Execution Time
DTC Scheduler	72.14	300 ms
DTC Cache	31.12	74 ms
PO Scheduler	531.03	36 ms

Table 1: Average results from 1077 runs of 180 seconds.

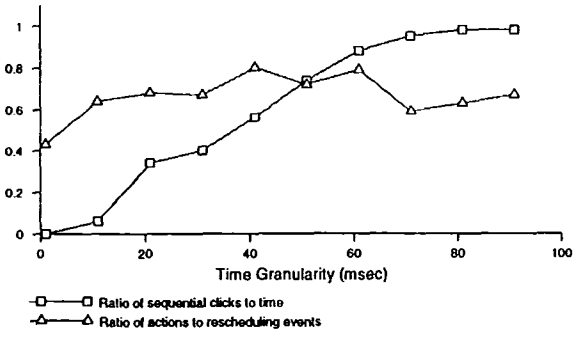


Figure 7: The effect of varying time granularity on agent behavior. A higher time ratio indicates that a greater percentage of sequential time units are seen, which should reduce the need for rescheduling. A higher action ratio indicates the available time was used more efficiently.

or constraint will be satisfied. Quantitative effects of this system can be seen in Table 1.

To test the caching subsystem, we performed 1077 runs in simulation using eight sensors and one target. As shown in the table, the caching system was able to avoid calling DTC 30% of the time, resulting in a significant savings in both time and computational power.

Time Granularity

The standard time granularity of agents running in our example environment is one millisecond, which dictates the scale of timestamps, execution statistics and commitments. Because we run in a conventional (i.e. not real-time) operating system, in addition to our relatively unpredictable activity durations, it becomes almost impossible to perform a given activity at precisely its scheduled time. For instance, some action X may be scheduled for time 1200. When the agent first checks its schedule for actions to perform, it is time 1184. In the subsequent cycle, 24 milliseconds have passed, and it is now time 1208. To maintain schedule integrity (especially with respect to predicted resource usage), we must shift or reschedule the method which missed its execution time before performing it. Despite existing optimizations, in large number these actions can still consume a significant portion of 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 (Durfee & Lesser 1988), but in practice a suitably chosen value has few drawbacks, because the agent is effectively already operating at a lower granularity due to the real time missed between agent activity cycles. Using this scheme, if we say that every agent tick corresponds to 20 milliseconds, the above action would be mapped to run at time 60. At time 1184, the agent would operate as if it were time 59, while 1208 would become 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, but 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 7, resulted in a 35% reduction in the number of shifted or rescheduled activities by using a time granularity between 40 and 60 ms. Ideally, the system should "see" each sequential time click, but as the graph shows, as the system reaches that point, the coarse timeline unnecessarily restricts the number of actions which may take place, reducing the overall efficiency.

Adapting to Environmental Conditions

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

Consider the model shown in Figure 8, 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 `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

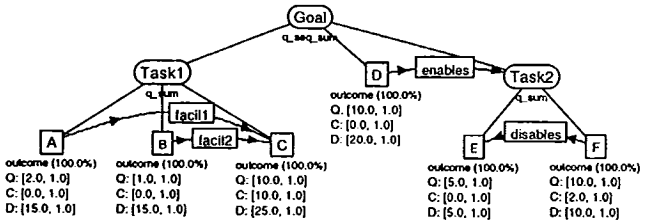


Figure 8: A TÆMS task structure modeling several different ways to achieve the same goal.

	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 8 under different conditions.

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 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 8 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 shown is not a round integer even though the qualities shown in Figure 8 are integers because A and B *facilitate* method C and increase C's quality when they are performed before it.

Row two shows the plan selected for the agent under a hard deadline of 40 seconds. This path through the network has the shortest duration enabling 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. 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 an 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, it shows how the solution in row six falls relative to the rest of the possible solutions – a good balance between maximizing quality while minimizing cost and duration.

Note 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.

Conclusion

The SRTA architecture has been designed to facilitate the construction of sophisticated agents, working in soft-real time environments possessing complex interactions and a variety of ways to accomplish any given task. With TÆMS , it provides domain independent mechanisms to model and quantify such interactions and alternatives. DTC and the partial ordered scheduler reason about these models, using information from the resource modeler 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 analyze and integrate solutions to dynamically occurring goals.

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

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

References

Bordini, R.; Bazzan, A.; Jannone, R.; Basso, D.; Vicari, R.; and Lesser, V. 2002. 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).*

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

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

Horling, B.; Benyo, B.; and Lesser, V. 2001. Using self-diagnosis to adapt organizational structures. In *Proceedings of the Fifth International Conference on Autonomous Agents*, 529–536.

Horling, B.; Lesser, V.; Vincent, R.; Raja, A.; and Zhang, S. 1999. The tæms white paper. http://mas.cs.umass.edu/res-earch/taems/white/.

Horling, B.; Vincent, R.; Mailler, R.; Shen, J.; Becker, R.; Rawlins, K.; and Lesser, V. 2001. Distributed sensor network for real time tracking. In *Proceedings of the Fifth International Conference on Autonomous Agents*, 417–424.

Horling, B.; Lesser, V.; Vincent, R.; and Wagner, T. 2002. The soft real-time agent control architecture. Computer Science Technical Report TR-02-14, University of Massachusetts at Amherst.

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

Lesser, V.; Atighetchi, M.; Horling, B.; Benyo, B.; Raja, A.; Vincent, R.; Wagner, T.; Xuan, P.; and Zhang, S. X. 1999. A Multi-Agent System for Intelligent Environment Control. In *Proceedings of the Third International Conference on Autonomous Agents (Agents99).*

Lesser, V.; Horling, B.; Klassner, F.; Raja, A.; Wagner, T.; and Zhang, S. X. 2000. BIG: An agent for resource-bounded information gathering and decision making. *Artificial Intelligence* 118(1-2):197–244. Elsevier Science Publishing.

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

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

Mailler, R.; Vincent, R.; Lesser, V.; Shen, J.; and Middlekoop, T. 2001. Soft real-time, cooperative negotiation for distributed resource allocation. In *Procceedings of the 2001 AAAI Fall Symposium on Negotiation.*

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

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

Wagner, T.; Garvey, A.; and Lesser, V. 1997. Complex Goal Criteria and Its Application in Design-to-Criteria Scheduling. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, 294–301. Also available as UMASS CS TR-1997-10.

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

Zhang, X.; Raja, A.; Lerner, B.; Lesser, V.; Osterweil, L.; and Wagner, T. 2000. Integrating high-level and detailed agent coordination into a layered architecture. Lecture Notes in Computer Science: Infrastructure for Scalable Multi-Agent Systems. Springer-Verlag, Berlin. 72–79. Also available as UMass Computer Science Technical Report 1999-029.