Implementing Soft Real-Time Agent Control *

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

Wagner, Thomas
University of Maine
Computer Science Department
Orono, ME 04469
wagner@umcs.maine.edu

ABSTRACT

Real-time control has become increasingly important as technologies are moved from the lab into real world situations or physical simulations. 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 real-time environment requiring distributed control, and how we modified our existing multi-agent technologies to meet this need. Two types of enhancements are covered: those which enable planning to meet real-time constraints, such as our task representation, meta-level costing, alternative plan selection, and partial-order scheduling, and those which facilitate on-line real-time control, including scheduling flexibility, caching, and windowed commitments.

1. OVERVIEW

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 task can become harder yet when the system is distributed, as the ability of a component to meet its deadlines can depend on the performance of another component not under its control. Conversely, a particular component in such a system should be able to reason about both its local deadlines, and those imposed on it through interactions with other parts of the system. In this paper we describe our efforts to migrate our multi-agent system framework into an environment which requires us to reason about and act in real-time. Some or all of the technologies used in this framework have been used successfully in several other environments [10, 9]. They have not, however, been deployed in an environment demanding real-time control that included real-time coordination between agents.

The particular environment we are operating under consists of several sensor nodes arranged in a region of finite area. Each sensor node is autonomous, capable of communication, computation and observation through the sensor itself. For the purposes of the scenario we will 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 the scenario 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 strung together to form estimated movement tracks.

The real-time requirement of this scenario 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 length of one second).

Competing with the tracking measurement activity are a number of other local goals, including target discovery scanning, data processing and measurement tasks for other targets. We don't see these as separate agents or threads, but rather as different objectives that an agent is multiplexing. Meta-level functionality such as negotiation, planning and scheduling also contend for local resources. To operate effectively, while still meeting the deadlines posed above, the agent must be capable of reasoning about and acting upon the importance of each of these activities.

In summary, our real-time needs for this application require us to synchronize several measurements on distributed sensors with a granularity of one second. A missed deadline may prevent the data from being fused, or the resulting triangulation may be inaccurate - but no catastrophic failure will occur. This provides individual agents with some minimal leeway to occasionally decommit from certain deadlines, or to miss them by trivial amounts of time, without failing to achieve the overall goal. Thus, our notion of real-time

*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 Grant No. IIS-9812755. 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.

here is relatively soft, enabling the agents to operate at a higher level of detail than hard real-time systems do.

The real-time aspects of this problem come down to several principles. An agent must know when actions should be performed, how to schedule its time and commitments such that they can be performed, 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. We begin by accurately modeling the activities the agent may perform, which can be done a priori or through a runtime learning process. This information is represented, along with other goal achievement and alternative plan information, in a TÆMS task structure [3, 5]. In addition to modeling primitive actions, we also model and schedule meta-level activities, such as negotiation. This permits us to cost-out the characteristics of these activities, allowing the agent to, for instance, directly reason about what sort of negotiation is appropriate for the given context. A planning component, the Design-to-Criteria scheduler (DTC) [17, 19], 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 then given to a partial order scheduling process which determines when individual actions should be performed, given precedence and runtime resource constraints. In general, we feel that real-time can be addressed by 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 dramatic 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. Finally, the notion of a "windowed" commitment, as seen in the synchronization process above, permits agents some measure of flexibility when satisfying commitments, reducing the need for renegotiation. In the remainder of this paper we will cover the functional details of the architecture, discuss the need for the various optimizations that have been added, and describe our performance experiences with the new design.

2. 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 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 rescheduling. Execution failure would also lead to a complete replanning and re-scheduling. This technique leads to "optimal" plans and schedules at each point if meta-level overheads are not included. As will be shown in section 2.2, however, the combinatorics associated with such large structures can get quite high. This

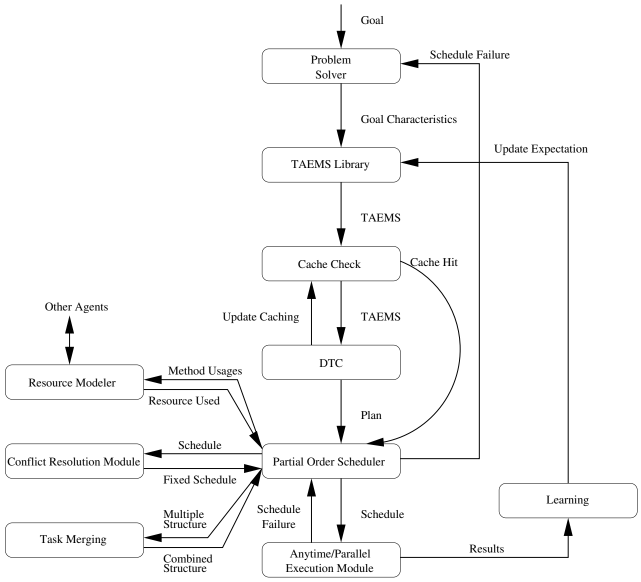


Figure 1: High-level agent control architecture.

made agents ponderous when working with frequent goal insertion or handling exceptions, because of the need to constantly perform the expensive DTC process. In a real-time environment, where the agent must constantly reevaluate their execution schedule in the face of varied action characteristics, this sort of control architecture was impractical.

In our new architecture, we have attempted to make the scheduling and planning process more incremental and compartmentalized. New goals can be added piecemeal to the execution schedule, without the need to re-plan all the agent's activities, and exceptions can be typically be handled through changes to only a small subset of the schedule. Figure 1 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. This structure is then used by our Design-To-Criteria (DTC) planning component, along with criteria such as potential deadlines, maximum cost, and minimum quality, to select an appropriate plan to achieve the goal. The planning process itself is expensive, so a caching scheme has been devised which retrieves past planning results from a cache whenever possible, bypassing the DTC call.

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[1]. Several components are used during this final scheduling phase. A resource modeling component is also used during this analysis to ensure that resource constraints are also

[1]From private conversations, it appears that the technique used to generate this schedule is similar to that seen in the DRU scheduler from the DECAF framework [4], a system developed concurrently with the research presented here.

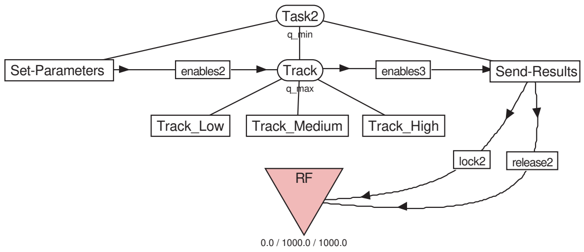


Figure 2: An example TÆMS task structure for tracking.

respected. A conflict resolution module reasons about mutually-exclusive tasks and commitments, determining the best way to handle conflicts. Finally, a task 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, decommiting from 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. A learning component also monitors execution performance, potentially updating the TÆMS template library when new trends are observed.

To better explain our architecture's functionality, we will work through a integrated example in the next several sections, using simplified versions of task structures in the actual sensor network application. 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 various goals may be achieved, and their various constraints and interdependencies respected.

2.1 TÆMS Generation

Before progressing, we must provide some background on our task description language, TÆMS. TÆMS, the Task Analysis, Environmental Modeling and Simulation language, is used to quantitatively describe the alternative ways a goal can be achieved [3, 5]. 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 2 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 `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. `Set-Parameters`, then, would be described with its expected duration and quality, allowing the scheduling and planning processes to reason about the effects of selecting this method for execution. 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 `q_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 `q_max` below `Track` says that its quality will be the maximum of any of its subtasks - the agent has a choice of one or more alternatives to complete `Track` (complete descriptions of these and other QAFs can be seen in [5]). Interactions between methods, tasks, and affected resources are also quantitatively described as interrelationships. The `enables` interrelationships in figure 2 represent precedence relationships, which in this case say that `Set-Parameters`, `Track`, and `Send-Results` must be performed in-order. `lock2` and `release2` are resource interrelationships, describing, for instance, the consumes and produces effects method `Send-Results` has on the resource `RF`. We will see in later sections how the different parts of a structure affect planning and scheduling.

The problem solver is responsible for translating high-level goals into TÆMS, which serves as a more detailed representation usable by other parts of the agent. This could be done by building TÆMS structures in the source code, but this tends to be impractical for all but the most trivial goals. On the other hand, the problem solver could read static structures from a plan library, selecting the one designed to address the particular goal in question. This works well, except is lacks the flexibility to easily handle the minor variations in structure needed when environmental conditions shift. We developed a hybrid scheme, which uses a library of TÆMS templates, which are dynamically instantiated at runtime, taking into account the agent's current working conditions. In this way we can handle such things as varying execution performance, negotiation partners and commitment details. For instance, in figure 2, the `Send-Results` method must specify which agent in the system the results should be sent to. 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.

At time 0 the agent will use its template library to generate the initialization structure seen in figure 4A. In this structure, the agent must first `Init` and then `Calibrate` its sensor. Properties passed into the template specifying the specific values used in `Init`, and the number of measurements used during `Calibrate`. As specified by the enables interrelationship, `Init` must successfully complete before the agent can `Send-Message-1`, reporting its capabilities to its local manager. `Send-Message-1` 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 2 and 4B, respectively, are generated later in the run in a similar manner.

2.2 DTC Planner / Initial Scheduler

Design-to-Criteria (DTC) scheduling is the soft real-time process of evaluating different possible courses of action for an intelligent agent and choosing the course that best fits the agent's current circumstances. For example, in a cost constrained situation the agent may be unable to purchase desired data and may thus be forced to spend more time doing its own local processing to produce the same quality result. Or, in a different situation when both time and cost are constrained, the agent may have to sacrifice some degree of solution quality in order 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.

As TÆMS task structures model a family of plans, the DTC scheduling problem has conceptually certain characteristics in common with planning and certain characteristics of more traditional scheduling problems, and it suffers from pronounced combinatorics

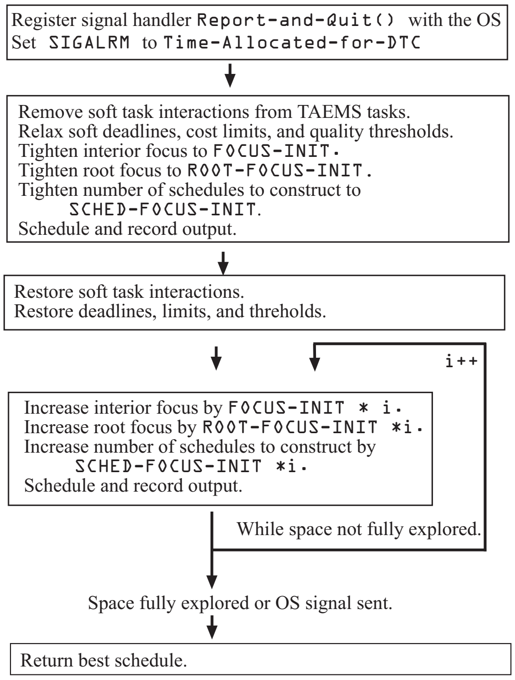


Figure 3: Real-Time Control for DTC

on both fronts. The scheduler's function is to read as input a TÆMS task structure (or a set of task structures) and to 1) decide which set of tasks to perform, 2) decide in what sequence the tasks should be performed, taking advantage of soft relationships where possible, 3) to perform the first two functions so as to address hard constraints, e.g., deadlines on tasks, and to balance the soft design/goal criteria specified by the client, to do this computation in soft real-time so that it can be used online.

Meeting these objectives is a non-trivial problem. In general, the upper-bound on the number of possible schedules for a TÆMS task structure containing n actions is given in Equation 1. Clearly, for any significant task structure the brute-strength approach of generating all possible schedules is infeasible – offline or online. This expression contains complexity from two main sources. On the "planning" side, the scheduler must consider the (unordered) $O(2^n)$ different alternative different ways to go about achieving the top level task (for a task structure with n actions). On the "scheduling" side, the scheduler must consider the $m!$ different possible orderings of each alternative, where m is the number of actions in the alternative.

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

The types of constraints that may be present in TÆMS and the existence of interactions between tasks (and the different *QAFs* that define how to achieve particular 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 [17]. From a very high level, the scheduler uses goal directed focusing, approximation, scheduling heuristics, and schedule improvement/repair heuristics [22, 14] 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 [6], but differs from most flexible computation approaches in its use of multiple actions to achieve flexibility (one exception is [7]) in contrast to *anytime algorithms* [2, 13, 20]. 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 [21], but we can work to reduce it when important to the client.

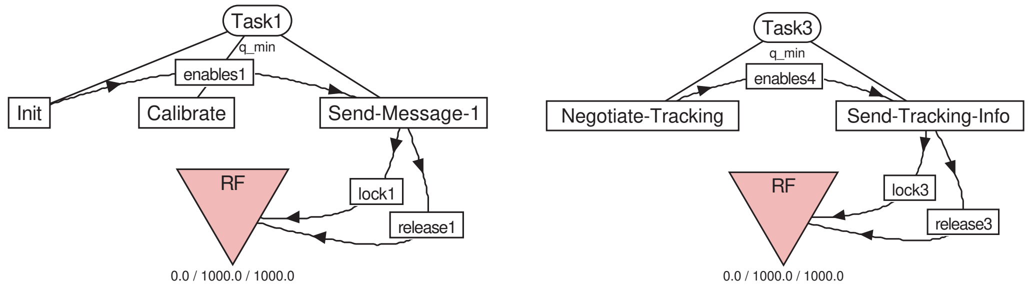
Until recently, DTC supplied online scheduling/planning services to other components by being "fast enough" for the activities being scheduled. For example, in the BIG information gathering agent [10], scheduling/planning accounted for less than 1% of the agent's execution time. However, in hard real-time situations, being fast enough is not sufficient, as discussed in [19]. The current generation scheduler supports hard real-time deadlines at the grainsize afforded by the unix/Linux operating system. The control algorithm used by the scheduler is shown in Figure 3. To meet hard deadlines on the amount of time the scheduler can take to plan/schedule, it first relaxes constraints that are likely to produce worst-case behavior and schedules. It then records the most highly rated schedule, restores a portion of the constraints, and schedules again. This schedule is also recorded. The scheduler then lessens its degree of focusing, thus enabling it to explore a larger percentage of the schedule solution space, and reschedules. The resulting schedule is then recorded, the degree of focusing is decreased again, and the scheduler again reschedules. This process continues until the hard-deadline is met or the scheduler explores the entire scheduling space. If the hard deadline occurs before the scheduler is able to produce a single viable schedule, no schedule is returned to the client.

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 quality-accumulation-functions, etc. Predictability [15] 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 number of times DTC is invoked (see section 3.2.

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-1`. A somewhat more interesting task structure is seen in `Task2` from figure 2, 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, then, 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 [19]. It selects for execution the plan `Set-Parameters`, `Track-Medium`, and `Send-Results`. After they are selected, the plans will next be used by the partial order scheduler to both evaluate precedence and resource constraints, and determine exactly when individual methods will be performed.

2.3 Partial Order Scheduler

DTC was designed for use in both single agents and agents sit-



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

Figure 4: Two TÆMS task structures, abstractions of those used in our agents.

uated in multi-agent environments. Thus, it makes no assumption about its ability to communicate with other agents or to "force" coordination between agents. It is used as an oracle by other coordination components, during agent communication and negotiation. This design approach, however, leads to complications when working in a real-time, multi-agent environment where distributed resource coordination is an issue. When resources can be used by multiple agents at the same time, DTC lacks the ability to request communication for the development of a resource usage model. This is the task of another control component that forms scheduling constraints based on an understanding of resource usage. In most applications, these constraints are formed by rescheduling to analyze the implications of particular commitments. In the real-time sensor application, the rescheduling overhead is too expensive for forming these types of relationships. The solution we have adopted is to use a subset of DTC's functionality, and then offloading 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 order schedule, one which includes descriptions of the interrelationships between the scheduled actions in addition to their desired execution times. In this partially ordered schedule, we explicitly represent precedence relationships between methods, constraints and deadlines. This information arises from commitments, resource and method interrelationships, and the QAFs assigned to tasks. Much of this information can be directly determined from the TÆMS task structure. Resources, however, must be analyzed in a more robust fashion, because of potential interactions from other activities, both locally and those to be performed by other agents.

In order to bind resources, we used 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 some constraints on when the method can start or finish, and asks it to find a point in time when the necessary resources are available to be used.

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. When searching, the resource modeler takes into account all the resource usages that it has been previously told about. 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.

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. In such cases, methods run 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 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. Thus, when scheduling conflicts arise, the conflict resolution component can compare the relative importance of the conflicting actions, and remove the one of lesser priority. If such a decommittal is made, or if no valid resolution can be found, the problem solving component is notified of the situation so that it can take appropriate action. 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. 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 5A, where all the known constraints are met. Some measure of parallelism can be achieved, seen with Set-Parameters and Send-Message-1, 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.

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 4B, 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 5B.

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 equal to 2000. This caused the partial order schedule to try other permutations of methods, which resulted in the schedule shown in figure 5C, which delays Send_Tracking_Info in favor of Send-Result. This allows the agent to proceed successfully despite its failed expectations.

3. 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 star time arrives, the method will not be started on time and may therefore fail to meet its deadline.

3.1 Meta-Level Accounting

Several issues cause this problem described above. Of primary concern in this example is the fact that the agent is not accounting for and scheduling all the activities the agent is performing. Many systems only schedule for the low-level tasks they perform - the actions which directly and tangibly affect the goal at hand. At the same time, however, there is an entire class of actions which the agent is performing, and therefore compete for the same processing time, which are not accounted for. Such tasks include many elements seen in figure 1: communication, negotiation, problem solving, planning, scheduling and the like. These so-called "meta-level" activities can constitute a significant fraction of the agent's running time, but are not being directly scheduled.

In this research we have added meta-level accounting of communication and negotiation. To do this, we first modeled these activities with TÆMS task structures. From a planning and scheduling point of view, there is no difference between low and meta-level actions, so to account for this time we need just an accurate model and a component capable performing these actions in response to a method execution. Given this, we can use our existing TÆMS processing components to correctly account for this time. The task structure from our running example, seen in figure 4B, models both negotiation and communication activities. The duration of a negotiation task is relatively deterministic, or at least can be described within some bounds, so creating the task structures was a matter of learning the characteristics of our negotiation scheme. An additional benefit of describing these activities in TÆMS is that it permits the planning component to reason about the selection of negotiation schemes. Consider a system where one had several different ways to negotiate over a particular commitment, each with different quality, cost and duration expectations. By describing these in TÆMS , we can simply pass the structure the generic DTC planning component, which will determine the most appropriate negotiation scheme for the current environmental conditions. Furthermore, once a given scheme is selected, it can potentially 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. These topics are more complicated due to their non-deterministic nature, i.e. the agent does not necessarily know a priori how long it will take to schedule an arbitrary set of interdependent actions. 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.

3.2 Plan Caching

A second issue affecting the agent's real time performance is that meta-level tasks take significant time in the first place. 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. Thus, as part of developing this control architecture, we have looked into optimizing the meta-level performance of our components.

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 1. 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, 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 will be covered in section 4.

3.3 Learning

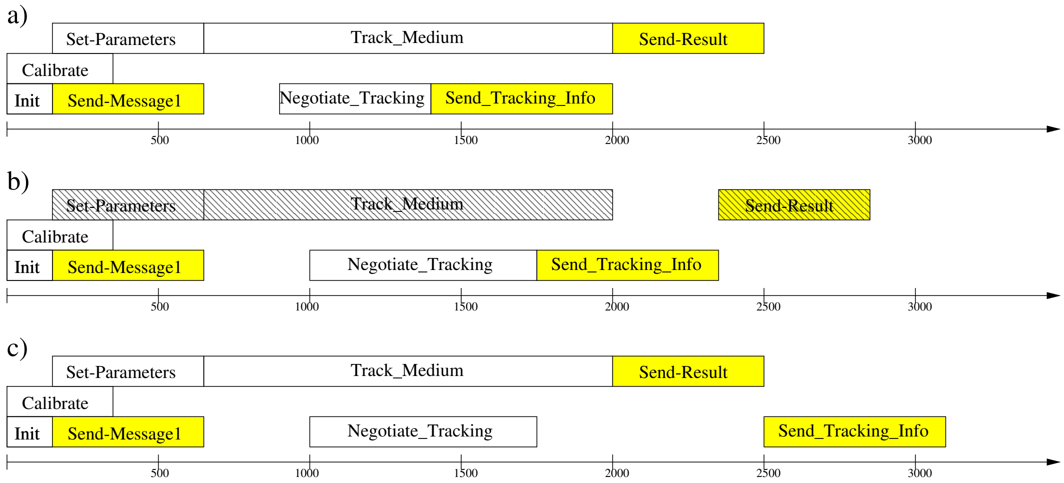


Figure 5: A) Initial schedule produced after all the goals have been received, B) Invalid schedule showing the deadline constraint broken by a method execution taking too long, C) Corrected schedule respecting all constraints.

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 [8], some quantitative and structural elements of TÆMS structures have been shown to be learnable using off-line analysis of a large corpus of results. While this technique would work to a certain extent for our application, we are more interested in using a lightweight runtime learning component to give the agent the capability to dynamically adapt to changing conditions.

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

3.4 Commitment Windows

Despite the improvements listed above, it is still difficult to model and execute actions to a millisecond resolution in a complex agent running on top of an uncontrolled environment (e.g. Java, generic kernel with competing local processes). Thus, our techniques must be able to tolerate the variability in execution time inherent in the real time environment.

We have previously discussed the importance of the partial order scheduler, which facilitates the efficient shifting of scheduled methods as dictated by temporal needs. Wherever possible, we offer similar flexibility in our commitment structures by specifying negotiated tasks with windows of execution time, rather than just fixed deadlines. This corresponds well to our problem domain, as commitments over scanning and tracking tasks naturally have windows of time where they may be successfully performed. As actions are scheduled or shifted, the window provides some leeway in when a time-constrained method can be performed, giving the agent the potential to locally resolve conflicts arising from failed expectations and new commitments. In addition, the notion of an execution window provides a clear metric usable for the relaxation or tightening of constraints occurring during negotiation[2].

Our commitments are typically delivered in the form of a *periodic task*, an action which should be performed repeatedly over some specified period. Again, this notion maps well to our sensor environment, where a given node may be needed to perform a series of measurements for scanning or tracking. This has the quality of reducing coordination overhead, important in our communication-limited environment. More germane to the topic at hand, however, is the periodic task's predictive capacity. By explicitly specifying a series of desired actions, the agent is able to schedule the periodic task out to an arbitrary point on its time horizon. This in turn better equips the agent to reason about and negotiate over potential actions and commitments in the future.

[2]Implementation of this scheme was done by Jiaying Shen and Roger Mailler.

4. PERFORMANCE

We implement this architecture to be able to track a vehicle moving using distributed radar sensors. In this section we will describe the real time environment our system operates in, and present performance results.

The goal of this application was to track one or more targets moving through the sensor environment. The radar sensor measurements consist of only amplitude and frequency values, so no one sensor has the ability to precisely determine the location of a target by itself. The sensors must therefore be organized and coordinated in a manner that permits their measurements to be used for triangulation. In the abstract, this situation is analogous to a distributed resource allocation problem, where the sensors represent resources which must be allocated to particular tasks at particular times, in order for the tasks to be effectively coordinated.

The available sensor platforms have three scanning regions, each a 120 degree arc encircling the sensor. Only one of these regions can be used to perform measurements at a time. The communication medium uses a low-speed, unreliable, radio-frequency (RF) system over eight separate channels. Messages cannot be both transmitted and received simultaneously regardless of channel assignment, and no two agents can transmit on a single channel at the same time without causing interference. The sensor platforms are capable of locally hosting one or more processes, which share a common CPU. Our solution populates each sensor platform with a single agent process. Targets move through the environment in an arbitrary pattern as the scenario progresses. We assume that agents have basic knowledge of themselves (i.e. position, orientation, capabilities, etc.). Unless specified, all other information must be communicated by other agents over the RF medium.

The need to triangulate a target's position requires frequent, closely coordinated actions amongst the agents - ideally three or more sen-

Component	Num. Calls	Execution Time
DTC Scheduler	72.14	300 ms
DTC Caching	31.12	74 ms
Partial Order Scheduler	531.03	36 ms

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

sors performing their measurements at the same time. In order to produce an accurate track, the sensors must therefore minimize the amount of time between measurements during triangulation, and maximize the number of triangulated positions.

To test our architecture, we used both a simulated environment, RADISM[3], and a physical hardware platform. More details on these experiments can be found in [16]. The configuration we used had four sensors, using four PCs running Linux as sensor's hosts, and one single target. The target was moving at 1 foot/sec. RMS error, which calculates the difference between the measured and actual tracks, is used as a rough metric to determine the effectiveness of the agent's synchronization (several other factors affect RMS as well). In the simulator, we were able to achieve an RMS error of 1.68 feet averaged over 1077 runs with reliable communication. In this case, the agents were able to synchronize their measurements within an average window size of 58 ms. To put this value into perspective, the average execution time for those measurements is about 1500 ms. Table 1 shows other timing measures used to determine how effective our optimization techniques were. The caching system in these tests was able to avoid calling DTC 43.13% of the time. It is also interesting to note that our agent calls the partial order scheduler almost at every cycle, to update and maintain the current schedule. With an average number of call 531.03 per run, we are calling this component every 338 ms.

5. FUTURE 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 [11], we cannot make performance guarantees [15] about agent control. However, in contrast to CIRCA, the approach presented here operates on multiple distributed agents and the statistically "fast enough" model addresses the requirements of this application. In 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. It is also unclear for other applications, e.g., information gathering via the net, if hard real-time guarantees are useful or needed.

There are number of technical directions that we think are important in developing this framework further. One involves developing a better understanding of how to make choices at the DTC level, given that in some cases the primitive methods can be executed in parallel if resources are available. We currently can do some of this reasoning at the DTC level [18] but it is does not use information from the detailed resource modeler, and for that reason we don't currently exploit this capability of DTC. Thus, the decisions at the DTC level are overly conservative about what plan alternative is most appropriate for accommodate the given deadline. We would thus like to create additional mechanisms to make better predictions. Along this line, we would also like to develop an efficient meta-meta reasoning component in order to decide how much effort should we allocate to the DTC component in the current situation given the ability of the DTC component to work in an anytime mode. Another role for this new component is to decide where and how much slack to put in the schedule to accommodate unexpected meta-control activities. As we discussed early in the paper, we don't explicitly allocate slack time for unexpected meta-control events such as planning and scheduling for new goals or revisions to existing schedules. Another aspect of the framework that we would like to extend is the conflict resolution component. One direction is to make it more sophisticated in its understanding exactly what previously scheduled event(s) is responsible for the conflict.

[3]RADISM was designed and built by Rome Labs.

6. REFERENCES

[1] Ella M. Atkins, Edmund H. Durfee, and Kang G. Shin. Detecting and Reacting to Unplanned-for World States. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, July 1997.

[2] 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.

[3] 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".

[4] John R. Graham and Keith S. Decker. Towards a distributed, environment-centered agent framework. In *Proceedings of the Sixth International Workshop on Agent Theories, Architectures, and Languages (ATAL)*, Florida, jul 1999.

[5] Bryan Horling et al. The tæms white paper, 1999. http://mas.cs.umass.edu/res-earch/taems/white/.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] David J. Musliner, James A. Hendler, Ashok K. Agrawala,

Edmund H. Durfee, Jay K. Strosnider, and C. J. Paul. The Challenge of Real-Time Artificial Intelligence. *Computer*, 28(1):58–66, January 1995.

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

[14] 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.

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

[16] Régis Vincent, Bryan Horling, Roger Mailler, Jiaying Shen, Raphen Becker, Kyle Rawlins, and Victor Lesser. Distributed sensor network for real time tracking. Submitted to Autonomous Agents 2001.

[17] 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.

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

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

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

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

[22] 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.