A Negotiation-based Interface Between a Real-time Scheduler and a Decision-Maker [1]

Alan Garvey and Keith Decker and Victor Lesser
Department of Computer Science
University of Massachusetts

UMass Computer Science Technical Report 94–08
March 17, 1994

Abstract

In our work on real-time problem solving we have found that the interface between the decision-maker and the real-time scheduler needs to be complex and bidirectional. We argue that this interface can usefully be modeled as a negotiation process. In this paper we present the details of the interface, as well as our scheduler that is capable of scheduling real-time tasks and providing the information required by the interface.

[1] This material is based upon work supported by the National Science Foundation under Grant No. IRI-9208920, NSF contract CDA 8922572, DARPA under ONR contract N00014-92-J-1698 and ONR contract N00014-92-J-1450. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

1 Introduction

In previous work on real-time scheduling, the interface between the scheduler and the application has been very simple. It is usually assumed that the application passes tasks on to the scheduler for scheduling and does not react when the scheduler is unable to schedule some tasks before their deadlines or only able to provide low quality solutions. A more complex interface is proposed in [Stankovic *et al.*, 1989] that allows the application to ask what-if questions of the scheduler and modulate the behavior of the scheduler, however these ideas have not been implemented to date. In our work on a complex, real-time, multi-agent problem-solver we have found that a more bidirectional, negotiation-based interface is useful. Whereas negotiation is usually seen as a tool for distributed AI applications, we feel that it is also useful in complex, single-agent problem-solving where there are many ways to solve a problem and also multiple criteria with which to judge the potential solutions. This paper presents our interface between higher-level decision-making and lower-level scheduling and acting, described as a negotiation process between the scheduler and the decision-maker. We will discuss why each part of the interface is useful, whether it can be implemented efficiently in a scheduler, and the effects of omitting that part. This paper will also describe a scheduler that is capable of effectively scheduling real-time tasks and providing the kind of information required by the interface.

As an example of what we mean by such an interface, consider a situation where multiple agents are working on a problem. Agent A has a method (Method A1) that enables the execution of an important method at another agent B. At this point the decision-maker at Agent A realizes that it should try to get the scheduler to schedule Method A1. It can do this by associating a *do commitment* with Method A1, meaning that it requests that the scheduler try to build schedules that execute Method A1. Agent A's scheduler returns a schedule that completes executing Method A1 at time 7. The decision-maker at Agent A tells other decision-makers that it can commit to giving them the result of Method A1 at time 8 (allowing time for communication to occur). At the second agent B, the decision-maker receives this message and passes it along to the scheduler, which reports back that time 8 is too late—the result is needed by time 6. Agent A's decision maker is informed of this feedback, and Agent A again invokes its scheduler, now with a *deadline commitment* to complete Method A1 by time 5 (to allow time for communication). Agent A's scheduler returns a schedule that commits to completing Method A1 by time 5 and Agent A communicates this information to the other agent, which is now able to complete its method by the deadline.

In this example, the decision-maker and scheduler at each agent are semi-autonomous subsystems, communicating bidirectionally and sometimes requiring more than one step to arrive at a satisfactory solution. As mentioned above, this paper focuses on the interface between each coordination module and local scheduler, as well as on the scheduling algorithm used by the local scheduler. A related paper [Decker and Lesser, 1994] describes how the coordination modules communicate with one another.

Another layered architecture approach is the *subsumption architecture* [Brooks, 1986]. In this architecture modules at a higher level can modulate the behavior of lower-level modules by overriding their inputs and outputs. Our approach to modulation is more complex since our modulations can be bidirectional and take the form of constraints on inputs and outputs rather than simple overrides. A layered approach is described in [Hudlická and Lesser, 1987] where a diagnosis system watches and informs the scheduling process. Another layered architecture

(briefly mentioned above) that has been proposed for real-time AI tasks is the extensions to the Spring real-time operating system described in [Stankovic *et al.*, 1989]. In this work, elements of the interface between the AI level and the scheduler include *time planners* that allow the AI level to ask what-if scheduling questions without disrupting the low level scheduling. The work on the Spring operating system also supports guarantees and endorsements, stronger versions of what we call commitments.

Given that significant communication in both directions is required in our layered approach, an obvious question is why have separate subsystems. There are many reasons why a distinct separation should exist between such a scheduler and decision-maker, including at least modularity, efficiency and reusability. Modularity suggests that separate functionality should be kept in separate modules with clearly defined interfaces. In general it is difficult for all current problem solving criteria to be encapsulated into an evaluation function and transmitted to the scheduler, because deciding what to do is a evolving computational *process*. From the scheduler's perspective, transmitting *all* potentially useful information about a schedule is also difficult and inefficient. Another reason why a separation should exist is that the subsystems work at different levels of abstraction. One of the roles of the decision maker is to constrain the search done by the scheduler, for example, by using commitments to tell the scheduler what parts of the task structure to focus on. While it is possible for the scheduler to use all available information to make such decisions itself, for efficiency reasons it is useful to have the decision maker constrain the search space for the scheduler. Another reason for separating scheduling and decision-making is that scheduling is a more generic activity and it should be possible to reuse schedulers in multiple applications. It is undesirable to reproduce a scheduler each time a new problem area is investigated.

We believe that the communication between the decision-maker and the scheduler can best be modeled as a process of *negotiation*. Negotiation is coordinated communication with the goal of enabling or improving problem solving. [Lâasri *et al.*, 1992] describe the information exchanged in negotiation in terms of *proposals*, *critiques*, and *explanations*. The description of the interface in this paper is arranged around these kinds of information. We first describe the basic input/output behavior of the scheduler in terms of proposals and explanations. We then examine feedback in the form of critiques, either of the schedule as produced by the scheduler or of the input specification. The paper concludes with a discussion of future work.

2 Proposals and Explanations

Problem-solving begins when a problem to be solved arrives at the decision-maker. In our work, problems are presented to the decision-maker as TÆMS task structures. The form of such task structures is described in more detail in [Decker and Lesser, 1993]. Briefly, a problem episode $\mathbf{E}$ consists of a set of independent *task groups* $\mathbf{E} = \langle \mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_n \rangle$, each with a hard deadline $\mathrm{D}(\mathcal{T})$ and containing interrelated *tasks* $T \in \mathcal{T}$. Within a task group, tasks form a directed acyclic graph through the `subtask` relationship. The *quality* or value of a task T at a particular time t (notated Q(T,t)) is a function of the quality of its subtasks (in this paper, the function is one of minimum (AND-like) or maximum (OR-like)). At the leaves of the DAG are *executable methods* M representing actual computations that can be performed. A single agent may have multiple methods for a task that trade-off time and quality. Besides the `subtask` relationship tasks can have other relationships to methods representing the interactions among

tasks. Such relationships include $\mathsf{enables}(T, M, \theta)$ meaning that the enabling task T must have quality above a threshold θ before the enabled method M can execute, $\mathsf{facilitates}(T, M, \phi_{\mathrm{d}}, \phi_{\mathrm{q}})$ meaning that if the facilitating task T has quality above a threshold then the facilitated method M can execute more quickly (proportional to ϕ_{d}) and/or achieve higher quality (proportional to ϕ_{q}), and $\mathsf{hinders}(T, M, \phi_{\mathrm{d}}, \phi_{\mathrm{q}})$ (the opposite of facilitates) where if the hindering task T has quality, then the hindered method M will achieve reduced quality and/or increased duration if it is executed. Note that these relationships occur from a task or method to a method. A relationship from Task A to Task B is translated to relationships from Task A to all methods below Task B.

In general, the decision-maker is faced with a single problem or set of problems, and either multiple ways to solve those problems, or multiple, changing criteria on the solutions, or both. Throughout the rest of this paper we will refer to two example scenarios to ground our interface and scheduling discussions. The multi-agent scenario is fully implemented and works as described. The real-time scenario, while plausible and similar to systems under development, does not exist at this time.

Multi-agent scenarios. In the multiagent scenarios we are working with, each executable method is executable by exactly one agent, however several agents may have identical methods for achieving quality for the same task. The goal of the decision-makers is to work together to produce the highest possible quality for as many task groups as possible, i.e., each attempts to maximize the *global* utility measure $\mathrm{U}(\mathbf{E}) = \sum_{\mathcal{T} \in \mathbf{E}} Q(\mathcal{T}, \mathrm{D}(\mathcal{T}))$. This is not straightforward, because each agent sees only some part of the total task structure, and it may be the case that no agent sees the entire structure. Thus the decision maker cannot simply ask the scheduler to optimize this global criteria. One kind of information that agents can communicate to one another is information about the task structures that they see. The decision-makers are responsible for coordinating their activity so as to avoid redundant method execution and allow relationships that extend across agents to be exploited or avoided as appropriate. The role of the scheduler is to schedule execution of local methods according to criteria provided by the decision-maker.

Real-time scenarios. In the real-time scenarios that we are working with, task groups arrive continuously, and require the use of multiple reusable physical resources (motorized tables, robot arms) and consumable resources. There are often not enough resources to complete all task groups, but the decision-maker still attempts to maximize its total utility. Each task group may have a different maximum payoff $I(\mathcal{T})$, and the decision-maker tries to maximize $\mathrm{U}(\mathbf{E}) = \sum_{\mathcal{T} \in \mathbf{E}} I(\mathcal{T}) Q(\mathcal{T}, \mathrm{D}(\mathcal{T}))$. Here the criteria are known but the best mix of problems to solve is not. The role of the scheduler here is to *endorse* the execution of time-critical control codes on multiple hardware platforms, using shared, private, and consumable resources. Some task groups, representing periodic, maintenance, or operating system activities, will be constantly present and their execution will be absolutely guaranteed.

2.1 Scheduler Inputs

The decision-maker *proposes* to the scheduler that a solution to each newly arriving task group be added to the schedule of methods to be executed. A basic request to the scheduler (its input) consists of four things: the task structures to be scheduled, a set of commitments, a set of non-local commitments, and a runtime indication.

The task structures to be scheduled $\mathbf{E_S}$, should include some indication of what aspects of those structures have changed since the scheduler was last invoked. If we write $B_A^t(X)$ to indicate what agent A believes at time t about X, then the scheduler at agent A at time t has access to $B_A^t(\mathbf{E})$, and $B_A^t(\mathbf{E}) \setminus B_A^{t-1}(\mathbf{E})$.

The set of commitments $\mathbf{C}$ are constraints that the invoker would like the scheduler to try to satisfy. We have defined three types of commitments:

- $C(\mathsf{Do}(T, q))$ is a commitment to 'do' (achieve quality for) T and is satisfied at the time t when $Q(T, t) \geq q$. A 'don't' commitment is also possible.
- $C(\mathsf{DL}(T, q, t_{dl}))$ is a 'deadline' commitment to do T by time t_{dl} and is satisfied at the time t when $[Q(T, t) \geq q] \wedge [t \leq t_{dl}]$. A $C(\mathsf{Do}(T, q))$ is really shorthand for $C(\mathsf{DL}(T, q, \mathsf{D}(T)))$.
- $C(\mathsf{EST}(T, q, t_{est}))$ is a 'earliest start time' commitment[1] to not begin work on T before time t_{est} and is satisfied at the time t_{est} iff $\forall t \leq t_{est},\ Q(T, t) \leq q$.

The importance of local commitments such as these are as *soft* constraints on the possible solutions. Any scheduler that can schedule real-time method executions can already deal with hard constraints such as deadlines and earliest start times. Hard commitments can be used to provide *guarantees* [Cheng *et al.*, 1988] by requiring commitments to be satisfied in all valid schedules.

Soft commitments are needed to handle the coordination of multiple agents where there is more than one way to solve a task or where there are soft coordination relationships such as facilitates. They are also useful in real-time systems, as shown by SPRING's use of *endorsements* [Cheng *et al.*, 1988] to indicate commitments that may only be violated when more important tasks arrive. When invoking the scheduler in a query mode, the decision-maker may also supply the symbolic values 'early' for a deadline commitment and 'late' for an earliest start time commitment, which indicates to the scheduler that it should attempt to satisfy the commitment as early or late as possible.

If a scheduler does not provide the ability to specify soft commitments, it is possible in some situations for the decision-maker to achieve the same results by repeated execution of scheduler queries using hard commitments (even changing the task structure, if need be). We believe it will always be more efficient to add the ability to interpret soft constraints to the scheduler than to play guessing games by invoking the scheduler multiple times.

The set of non-local commitments $\mathbf{NLC}$ are commitments that the scheduler can assume will be satisfied. These are of the form of the commitments mentioned above and tell the scheduler to expect to receive the indicated results at the indicated time.

In multi-agent problems, non-local commitments can be used to communicate work that will be done by other agents. This component is necessary for achieving coordinated behavior in complex domains. These non-local commitments might be created at run time by the decision-makers, or they might be derived from pre-defined 'social laws' [Shoham and Tennenholtz, 1992] that all agents agree to, or are constructed to, satisfy. Another effect of NLCs in multi-agent problems is the triggering of non-local effects (coordination relationships); each non-local deadline commitment, for example, implies an earliest start time on the 'affected' end of any

[1] In fact this is more general than a standard earliest start time constraint, in that it allows some nonzero amount of work to be done on T as long as quality does not go above the threshold. Standard earliest start times can be modeled with a q value of 0.

relationships. For hard relationships like enables this implies a hard earliest start time; for soft relationships it actually expands the search space (since each affected task can be started either before or after the earliest start time with different results).

In real-time problems as well, non-local commitments are important. In our example domain not all of the hardware is under the control of the real-time operating system scheduler, in particular the robot arms run on separate hardware with a separate specialized execution controller. The only way for the RT scheduler to function is to allow the robot hardware to make non-local commitments (in this case, worse-case execution time guarantees) about certain physical activities that are not directly under the control of the real-time scheduler.

Another potential use for non-local commitments is to allow the decision-maker to direct the search of the scheduler. The decision-maker can use non-local commitments to ask questions such as, assuming quality is achieved in this part of the task structure, how could we take advantage of that in other parts of the task structure. This could be useful in situations where the cost of the information gathering associated with expanding a task structure is potentially large. It could also be used in situations where the scheduler has successfully produced a schedule to satisfy one part of a task structure and the decision-maker now wants to focus the schedulers attention on another part that can begin execution when the previously scheduled work is completed. In time-constrained situations such non-local commitments can reduce search for the scheduler by allowing it to prune committed portions of the task structure.

Various mechanisms for controlling the runtime of the scheduler can include: a hard deadline by which the scheduler should complete; a satisficing value for a schedule (the scheduler completes when a schedule of at least this value is found); or a decision-theoretic tradeoff function that indicates the added value of spending time finding better schedules versus executing the first element of the current schedule[Russell and Wefald, 1991]. In non-real-time scenarios, this might not be particularly important as long as the runtime of the scheduler is small compared to the grain size of application tasks. In real-time scenarios it is crucial to at least be able to predict the worst-case performance of the scheduler.

2.2 Basic Scheduling Algorithm

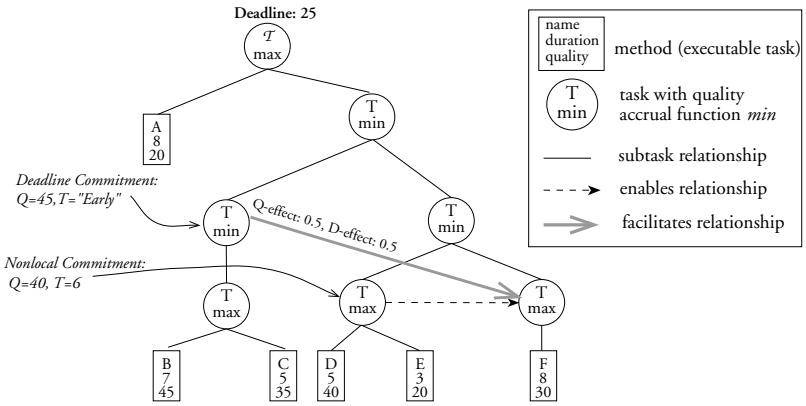

Figure 1: An example of a complete input specification to the scheduler.

In general, these scheduling problems are NP-Hard. For that reason, heuristic scheduling is necessary for all but the smallest problem instances. [Garvey *et al.*, 1993] describes an optimal algorithm for a simplified version of this kind of scheduling problem. The algorithm we present

here is an implemented version of the design-to-time scheduling paradigm discussed in [Garvey and Lesser, 1993].

Figure 1 shows an example of a complete input specification to the scheduler. Given a problem of the form described above, the scheduler attempts to produce schedules that adequately solve the problem. The scheduler consists of two main components: an *alternative generator* that chooses sets of methods from a task structure that should return quality for the task structure using just the task/subtask relationship, and a *method orderer* (aka scheduler) that takes a set of methods and generates a schedule. Currently there are a small fixed set of alternative generators including *highest quality*, *minimum duration*, *minimum nonlocal reliance*, and each of the above augmented to try to satisfy otherwise unsatisfiable commitments. In the example given above the *highest quality* alternative generator would choose the methods B, D, F then remove D because an adequate value for D is provided by the nonlocal commitment early enough to be useful.

The method orderer takes each set of methods returned by an alternative generator and tries to schedule those methods. It uses a simple greedy heuristic algorithm that rates each method against a number of heuristics, adds the one with the highest rating to the end of the schedule and continues. If it schedules all methods this way, then it is done, otherwise it tries adding slack time to the schedule, hoping that unscheduled methods will be schedulable at a later time (for example, because of later earliest start times or nonlocal enablement.) Some of the heuristics used by the method orderer include never execute a method that is not enabled or will finish past its deadline, prefer methods that facilitate or enable other methods and prefer methods with earlier deadlines. Currently the method orderer does no backtracking; if it is not able to schedule all methods, it returns the partial schedule it is able to construct. In the example the method orderer would take the set of methods chosen by the *highest quality* alternative generator (B, F) and choose to do B first, both because it contributes quality to a task that facilitates F and because F is not enabled until the non-local commitment result is received at time 6. If, even after scheduling B, the non-local commitment result would not be available, then the method orderer would schedule idle time to allow the enabling result to be received.

2.3 Scheduler Output

The output from the scheduler after an invocation should include at least one valid schedule, a list of satisfied commitments, a list of violated commitments with alternatives, an indication of tasks that should be scheduled but are not, and an indication of the value of each returned schedule with respect to some fixed set of criteria.

A set of valid schedules $\mathbf{S}$ is returned that do a satisfactory job of satisfying the problem given to the scheduler. An individual schedule $S \in \mathbf{S}$ consists of at least a set of methods and start times: $S = \{\langle M_1, t_1\rangle, \langle M_2, t_2\rangle, \ldots, \langle M_n, t_n\rangle\}$. This output is of course necessary, and forms the initial *proposal* in the negotiation process. The remaining items provide an explanation of this proposal.

The next three items returned (satisfied commitments, violated commitments with alternatives, and multicriteria schedule values) are not *necessary* for the scheduler to provide, because they can all be derived mathematically from the schedule itself and the set of non-local commitments. However, for practical implementations, the scheduler often has this information at hand, or can collect it during schedule generation, and it would be expensive to recompute.

The set of input commitments that are satisfied in a schedule $\mathrm{Satisfied}(S)$ is returned ($\forall S \in \mathbf{S}, \mathrm{Satisfied}(S) \subset \mathbf{C}$). If the scheduler supports symbolic local commitments like 'early' deadline commitments and 'late' earliest start time, then it must also supply an indication of when the commitment is expected to be satisfied in the schedule $\mathrm{SatTime}(C, S)$. For example, if $C_1 = \mathsf{DL}(T, q, \text{'early'})$ and $C_1 \in \mathrm{Satisfied}(S)$ then $\mathrm{SatTime}(C, S) = \min t \leq \mathrm{D}(T)$ s.t. $Q_{\mathrm{est}}(T, t, S) \geq q$.

The set of input commitments that are violated in a schedule $\mathrm{Violated}(S)$ is returned. For each violated commitment, a proposed modification to the commitment that the scheduler *is* able to satisfy ($\mathrm{Alt}(C, S)$) is also returned. For earliest start time and deadline commitments this involves a proposed new time and/or minimum quality. For do/don't commitments this involves a recommended retraction or a reduced minimum quality value. For example, for a violated deadline commitment $C(\mathsf{DL}(T, q, t_{dl})) \in \mathrm{Violated}(S)$ the function $\mathrm{Alt}(C, S)$ returns an alternative commitment $C(\mathsf{DL}(T, q, t^*_{dl}))$ where $t^*_{dl} = \min t$ such that $Q(T, t) \geq q$ if such a t exists, or NIL otherwise.

The knowledge that certain commitments are satisfied or violated is absolutely necessary to the decision-maker that uses commitments, regardless of the domain.

An indication of the "value" of the schedules that were returned according to several objective criteria. Some of the objective functions that can be measured include the total quality for all scheduled task groups,[2] the number of task groups that do/do not complete before their deadline, the amount of slack time available in the schedule to allow easy scheduling of new tasks and/or allow time for tasks to take longer than expected to run, and the number (or weighted value) of the commitments that are not satisfied in the schedule.

Complex real problems invariably involve multiple evaluation criteria that must be balanced with one another; we view this balancing as the role of the decision-maker, and the scheduler attempts to maximize the current criteria, often returning multiple schedules (e.g., one that best satisfies each of the current criteria.) While the ability to evaluate a schedule with respect to certain criteria could be implemented outside of the scheduler, the ability to attempt to optimize certain criteria can only be placed in the scheduler.

A minimal list of tasks in the task structure that the schedule is not providing quality for but would need to have quality to allow their task group to achieve non-zero quality. Such tasks can result from the scheduler not having any local methods to generate quality for the tasks (either because the task structure is distributed across agents and those tasks have methods at some other agent(s), or because the agent has not yet done the information gathering necessary to determine what methods are available for the task.) Such tasks can also result from the scheduler not being able to schedule the execution of all methods known to it because of deadlines or other constraints.

A summary of the output of the scheduler for the example problem given above is shown in Figure 2. In this example three schedules are returned. The bottom one is the schedule generated by the *minimum duration* generator and produces a fast, low quality result, violating the given deadline commitment because no quality is ever generated for the committed task. The middle schedule is generated by the *highest quality* generator and produces the highest possible quality in the fastest possible time, satisfying the deadline commitment at time 7. The top schedule is generated by the *minimum nonlocal reliance* generator and produces the

[2] This could be a weighted sum if task group importance varies, or some more complex function if desired.

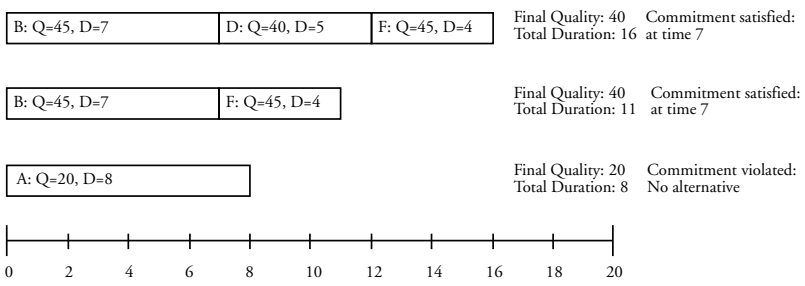

Figure 2: An example of the output of the scheduler for the example problem given above.

highest quality possible completely locally, not relying on the given nonlocal commitment, also satisfying the deadline commitment at time 7. Which of these schedules is chosen by the decision-maker depends on the current evaluation criteria. If the fastest possible, acceptable result is desired, perhaps because of a large workload of other tasks, then the bottom schedule is chosen. If the best possible result in the minimum possible time is desired, the the middle schedule is chosen. If the best possible result that does not rely on other agents is desired (possibly because of other work that those agents need to do or a concern about the other agent's reliability) then the top schedule is chosen.

3 Critiques and Repair

Critiques are expressions of dissatisfaction with particular parts of a proposal. They can be made internally by the scheduler as part of the schedule creation process or externally by the decision-maker. In both cases the scheduler attempts to respond to the critique by *repairing* the schedule. Repair consists of deciding what part of the schedule to modify (not necessarily the exact part criticized by the critique) and deciding what modification to make.

Examples of the kinds of critiques that can be made about a schedule:

- Methods that did not get scheduled.
- Facilitates relationships not taking effect.
- Hinders relationships not avoided.
- Parts of the task structure without quality (presumably that would lead to quality for the root task).
- Commitments that were violated.

There are many possible scheduling actions to take in response to critiques including adding idle time, changing method order, switching to faster alternatives, and adding additional methods to the schedule.

At this point in our research, critiques are only generated internally by the scheduler. After the scheduler has generated a new schedule it critiques it looking for a few kinds of specific problems. If these problems are detected it attempts to repair them by adding new methods or idle time to the schedule. In the future we intend to add many more critiques and repairs to the scheduler, including critiques from outside the scheduler. We also intend to study the tradeoffs associated with critiques, comparing the improvement in performance with the added runtime cost.

4 Discussion

We will first return to the main reasons for a bidirectional interface separating scheduling from decision making—reusability, modularity, and efficiency.

Of the various reasons for developing a complex bidirectional interface, the reusability argument is the most clear. A scheduling component that attempts to find optimal method execution times for arbitrary task structures and evaluation criteria is useful in many domains, regardless of whether they have real-time or distributed problem solving components. A more important related question is whether the extra capabilities required by this interface can be provided cheaply by an otherwise efficient, reusable scheduler. Examining the input characteristics, we find that the input task structures are not much more than the specification of the problem to be solved, and that handling these constraints would not be an additional burden to a standalone scheduler. Commitments, amounting to preferred (soft) deadlines and earliest start times are also part of standard real-time scheduling specifications. So too are the mechanisms for controlling the runtime of the scheduler, for any scheduler that can schedule real-time tasks with deadlines. The only potentially unique feature is the input of non-local commitments. If non-local commitments are always taken at face value (no lies) then they can be used for making more efficient searches, as discussed earlier.

Examining the reusable scheduler's output characteristics for efficient implementation, we find the first two—the schedules themselves and which commitments are satisfied when—to be non-controversial parts of almost any real-time scheduler. So too is the list of violated commitments, but perhaps not the generated alternatives. Generating alternatives to violated commitments on executable methods is trivial—just look in the generated schedule and return the actual execution if found, or suggest retraction of the commitment otherwise. Generating alternatives for high-level task commitments may be somewhat more complex, depending on the internal structure of the scheduler and what information is efficiently available. As we mentioned earlier, this output characteristic was assigned to the scheduler because it is *usually* more efficient to compute there, but it can be (inefficiently) computed by the decision maker from the schedule and non-local commitments themselves if necessary. The same thing is true of schedule evaluations under multiple criteria—it is our experience that the scheduler can more efficiently calculate these evaluations that the decision maker. The scheduler usually has already computed these evaluations as part of its search process. The most unique and potentially expensive output characteristic is the production of a task list for which the scheduler desires quality but cannot produce. This behavior enables several sophisticated responses in multi-agent systems, such as contracting behavior on the part of the decision maker on behalf of the scheduler, but is not part of the standard definition of scheduling problems and undoubtedly causes extra overhead. We plan to analyze how this behavior can be efficiently provided and under what circumstances it is useful in future work.

The other two reasons for a complex bidirectional interface, efficiency and modularity, are closely tied. Efficiency is primarily the ability to search on, or attempt to optimize, specific criteria effectively. Defining a carefully delimited scheduling problem, and even limited search criteria, allows for the construction of a efficient scheduler. In fact, multiple schedulers might be constructed, each optimized to search (perhaps in parallel) under a different criteria (best quality, least violated commitments, earliest finish time). This leads directly to the modularity argument—that in general it is difficult to encapsulate all problem solving criteria into a single

evaluation function, and deciding on the correct criteria is an evolving computational process (to be handled by a separate decision maker, we argue).

5 Conclusions

We have presented the details of a complex, bidirectional interface between a decision-maker and a scheduler. We have argued that it is useful to separate scheduling from decision-making at least for reasons of modularity, efficiency and reusability. We have also described a scheduler that is capable of both scheduling the kinds of task structures presented, and responding to and providing the kinds of information required by the interface.

The major ideas presented here, including the details of the interface, have been implemented in a multi-agent problem-solver. The interface was developed in response to the real requirements of building this complex problem-solver to solve randomly generated task structures in the TÆMS environment. We have found the schedules produced and runtime of our scheduling algorithm to be acceptable for this problem-solver, but other applications might require more complex schedulers or faster schedulers.

In the future we would like to extend the system in a few directions. One interesting area to explore is doing asynchronous, concurrent decision-making and scheduling. Both activities could go on simultaneously with both systems evaluating what requests to respond to first and how to respond to changing environments. Another area we intend to investigate is the effect of uncertainty in the duration and quality of methods. Such uncertainty increases the difficulty of scheduling and potentially reduces the reliability of commitments. Guaranteeing the satisfaction of commitments would require the same kind of worst-case execution time scheduling currently done by systems-oriented real-time operating systems [Cheng *et al.*, 1988], however, a more probabilistic scheduler could try to produce schedules that improve performance in most cases, but occasionally fail due to worst-case task performance.

References

[Brooks, 1986] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2(1):14–23, March 1986.

[Cheng *et al.*, 1988] S. Cheng, J. Stankovic, and K. Ramamritham. Scheduling algorithms for hard real-time systems. In *Hard Real-Time Systems*. IEEE Press, 1988.

[Decker and Lesser, 1993] Keith S. Decker and Victor R. Lesser. Quantitative modeling of complex computational task environments. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 217–224, Washington, July 1993.

[Decker and Lesser, 1994] Keith S. Decker and Victor R. Lesser. Designing a family of coordination algorithms. Submitted to AAAI-94, 1994.

[Garvey and Lesser, 1993] Alan Garvey and Victor Lesser. Design-to-time real-time scheduling. *IEEE Transactions on Systems, Man and Cybernetics*, 23(6):1491–1502, 1993.

[Garvey *et al.*, 1993] Alan Garvey, Marty Humphrey, and Victor Lesser. Task interdependencies in design-to-time real-time scheduling. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 580–585, Washington, D.C., July 1993.

[Hudlická and Lesser, 1987] E. Hudlická and V. R. Lesser. Modeling and diagnosing problem-solving system behavior. *IEEE Transactions on Systems, Man, and Cybernetics*, 17(3):407–419, May/June 1987.

[Lâasri *et al.*, 1992] Brigitte Lâasri, Hassan Lâasri, Susan Lander, and Victor Lesser. A generic model for intelligent negotiating agents. *International Journal on Intelligent Cooperative Information Systems*, 1(2):291–317, 1992.

[Russell and Wefald, 1991] Stuart Russell and Eric Wefald. *Do the Right Thing: Studies in Limited Rationality*. MIT Press, Cambridge, MA, 1991.

[Shoham and Tennenholtz, 1992] Y. Shoham and M. Tennenholtz. On the synthesis of useful social laws for artificial agnet societies (preliminary report). In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 276–281, San Jose, July 1992.

[Stankovic *et al.*, 1989] J. A. Stankovic, K. Ramamritham, and D. Niehaus. On using the Spring kernel to support real-time AI applications. In *Proceedings of the EuroMicro Workshop on Real-time Systems*, 1989.