Design-to-time Real-Time Scheduling [1]

Alan Garvey Victor Lesser

Department of Computer Science
University of Massachusetts
Amherst, Massachusetts 01003
CSNET: GARVEY@CS.UMASS.EDU

April 14, 1994

Abstract

Design-to-time is an approach to problem-solving in resource-constrained domains where: multiple solution methods are available for tasks, those solution methods make tradeoffs in solution quality versus time, and satisficing solutions are acceptable. Design-to-time involves designing a solution to a problem that uses all available resources to maximize the solution quality within the available time. This paper defines the design-to-time approach in detail, contrasting it to the anytime algorithm approach, and presents a heuristic algorithm for design-to-time real-time scheduling.

Our blackboard architecture that implements the design-to-time approach is discussed and an example problem and solution from the Distributed Vehicle Monitoring Testbed (DVMT) is described in detail. Experimental results, generated using a simulation, show the effects of various parameters on scheduler performance. Finally we discuss future research goals and plans.

[1]This work was partly supported by the Office of Naval Research under a University Research Initiative grant, number N00014-86-K-0764, NSF contract CDA 8922572, and ONR contract N00014-89-J-1877. This work is sponsored by the Department of the Navy, Office of the Chief of Naval Research, and 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 to Design-to-time

Design-to-time (a generalization of what we have previously called approximate processing[1]) is an approach to solving problems in domains where

- there is not necessarily enough time available for all processing,
- there are soft and hard real-time deadlines,
- multiple solution methods (which make tradeoffs in solution quality and timeliness) are available for tasks,
- solutions not completely satisfying optimal solution criteria (so called *satisficing* solutions) are acceptable in over-constrained situations, and
- the predictability of deadlines and task resource requirements is reasonable, although the predictability of new task arrival times may be low.

How predictable deadlines and durations need to be is based on a complex set of factors that is based in part on the system load and the amount of time required to recognize when a task will not meet its deadline and react to that event appropriately. A system can tolerate uncertainty in its predictions if

- monitoring can be done quickly and accurately, so that when a task will not meet its deadline enough time remains to execute a faster method, or
- intermediate results can be shared among methods, so that when it is necessary to switch to a faster method the intermediate results generated by the previous method can be used, or
- there exists a fall back method that quickly generates an minimally acceptable solution.

The methodology is known as design-to-time because it advocates the use of all available time to generate the best solutions possible. It is a problem-solving method of the type described by D'Ambrosio[2] as those which "given a time bound, dynamically construct and execute a problem solving procedure which will (probably) produce a reasonable answer within (approximately) the time available."[1]

This form of problem-solving is related to (but distinct from) the use of *anytime algorithms*[4, 5, 6]. Anytime algorithms as described by Dean and Boddy are interruptible procedures that always have a result available and that are expected to produce better results as they are given additional time. Russell and Zilberstein make a distinction between two kinds of anytime algorithms: *interruptible* algorithms, which can be interrupted at any time and always produce a result, and *contract* algorithms, which must be given a time allocation in advance, and produce better results with increased time allocations, but may produce no useful results if interrupted before their allocated time. Design-to-time differs from contract anytime algorithms in that we have a predefined set of solution methods with discrete duration and quality values. Contract anytime algorithms can generate a solution method of any duration; that duration just has to be specified before task execution begins. Figure 1 is a graph showing

[1] It appears that Bonissone and Halverson [3] were the first to use the term "design-to-time" to refer to systems of the form described by D'Ambrosio.

what an example of the difference in solution quality versus time tradeoffs in design-to-time and anytime algorithms might look like. Note that because of situation-specific tradeoffs in the applicability of problem-solving methods this graph is dynamic. The design-to-time methods appear above the anytime curve, both because it is assumed that the application of a single method with no attempt to generate useful intermediate results should take less time than the anytime approach to produce the same level of quality, and because if the anytime method is faster then the design-to-time approach can use it.

Other research that addresses real-time scheduling concerns by providing multiple solution methods includes the work on *imprecise computation*[7], the GARTL real-time programming language[8], and the Flex language[9].

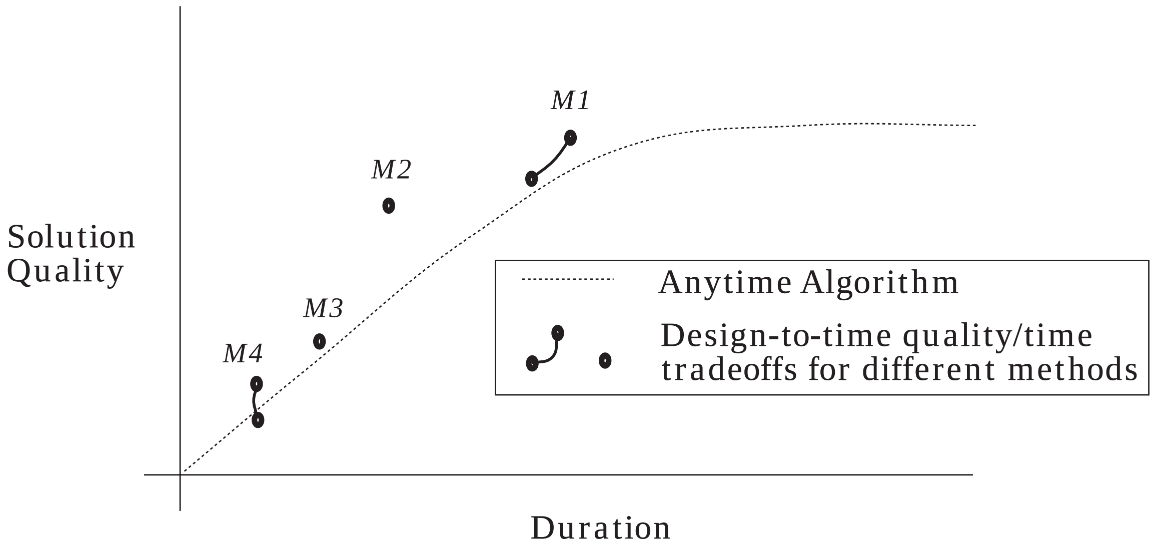

Figure 1: Anytime algorithm versus design-to-time tradeoffs.

In this paper we investigate the implications of design-to-time on the real-time scheduling and controlling of task execution for tasks with soft and hard deadlines[2]. We describe a controller architecture for allocating resources to tasks at a high level and micro-scheduling the execution of tasks at a low-level. The task of the controller is to design a high-level solution to a problem using all of the available resources as efficiently as possible.

In our approach a controller plans at a high level which tasks to perform, what solution method to use for each task, and when to work on each task, based on expected time to process each task, quality versus time tradeoffs between alternative methods, and the distribution of deadlines. Each task is broken up into multiple problem-solving steps that may have order constraints among them or depend on the availability of data. A lower level execution subsystem then directs the actual execution of low-level problem-solving steps, potentially interleaving steps from several tasks. Feedback from the execution subsystem to the controller allows the rescheduling of tasks when necessary because of inaccurate predictions or unexpected events. A high-level picture of this architecture is shown in Figure 2.

[2] In previous work [10, 11] we examined the effects of the design-to-time approach on other aspects of a problem-solving architecture.

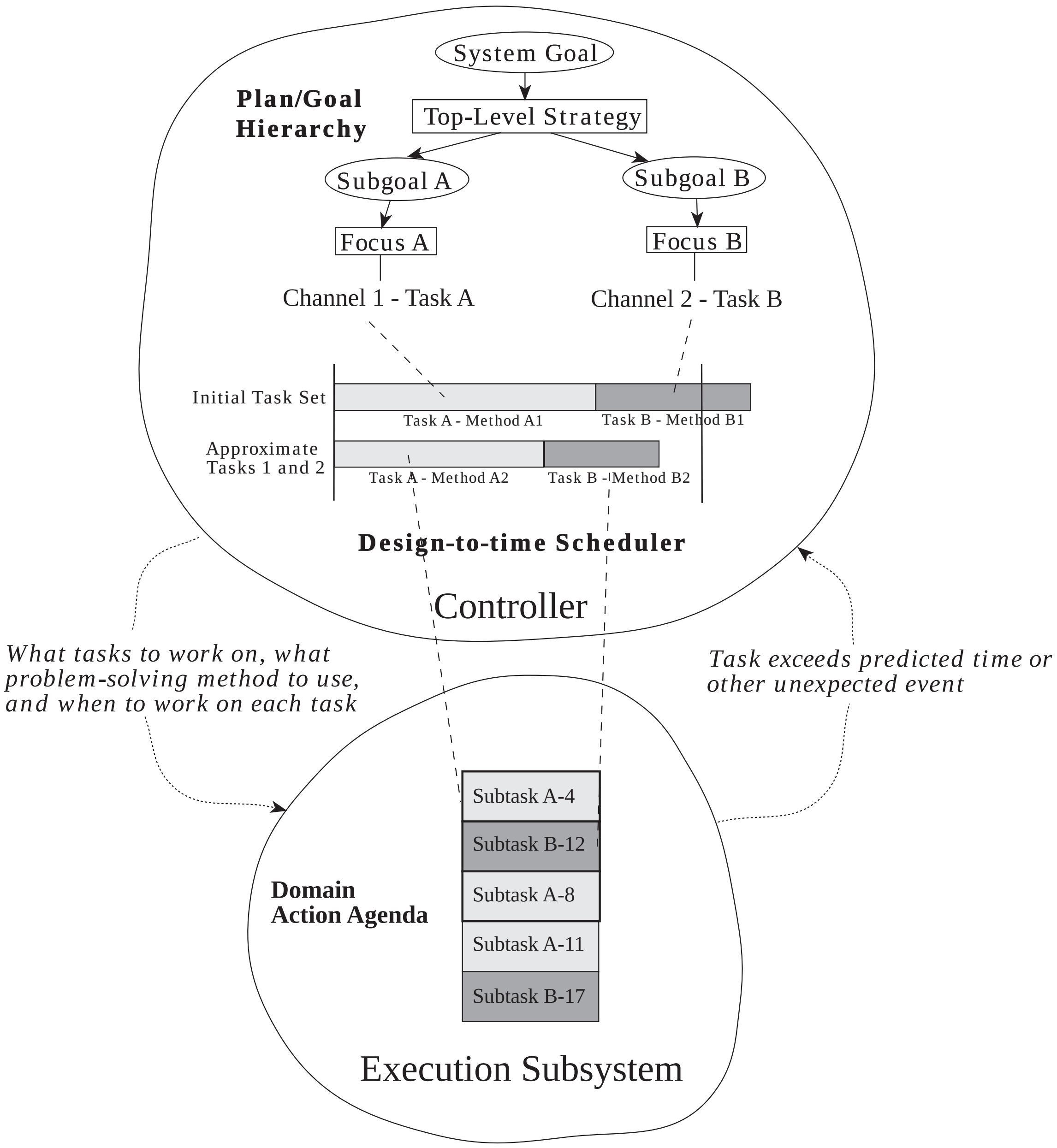

Figure 2: High-level view of our design-to-time architecture.

The controller makes decisions about resource allocation for the current time and for discrete times in the future. In the terminology of this paper, each task is assigned a *channel*, which is a separate thread of problem-solving with its own goal. The controller assigns specific resources to each channel for discrete amounts of time. In our current work the controller creates plans with resource allocation updates at periodic intervals, but other update schedules are possible and are probably more appropriate for particular problems (e.g., updates in reaction to the arrival of bursts of tasks).

The controller has two major methods available to it for adjusting resource allocations when there are not enough resources available to meet all demands: modifying tasks to use approximations, and postponing tasks to later times when, presumably, resources are available. Approximations reduce problem-solving time at the expense of a degradation in other aspects of solution quality[1, 10]. Approximate processing is a useful reconfiguration method when the effects on solution quality are within the bounds allowed by the goal associated with the task. Each approximate method has its own situation-specific tradeoff between solution quality and time, which allows the tradeoffs of each approximation to be taken into account when choosing a problem-solving method.

Another reallocation method is the postponing of tasks. Postponing a task is a useful reallocation method when the postponement will not cause the task to miss a future deadline and when the future workload is expected to be less than the current workload. When postponing a task it is necessary to take any deadlines associated with that task into account. Postponing a task beyond its deadline, or postponing a task that is required to allow a related task to meet its deadline are undesirable behaviors and should be avoided. If the projected future workload does not contain enough resources to allow the completion of the task, then postponing the task merely pushes the problem into the future. Approximations and postponement are used only when necessary to meet specific deadlines. The design-to-time method advocates using all of the available time to generate the best solution possible.

Approximate processing requires the problem solver to be very flexible in its ability to represent and efficiently implement a variety of processing strategies. With minimal overhead, the problem solver should dynamically respond to the current situation by altering its operators and state space abstraction to produce a range of acceptable answers[10, 1]. It should also be possible for different problem-solving methods to share intermediate results (i.e., for some of the intermediate results generated by one problem-solving method to reduce the computation time or increase the quality produced by another method.)

The execution subsystem assures that the planned resource allocation is adhered to by micro-scheduling the work for each channel within a particular resource allocation plan. In particular the execution subsystem can interleave work from multiple tasks to take advantage of interactions among tasks, (e.g., results which are useful to multiple tasks can be generated early to allow them to be shared.) When tasks exceed their predicted resource requirements or other unexpected events occur, the execution subsystem signals the controller, which can adjust current and future resource allocations to meet the new requirements.

An algorithm for design-to-time scheduling is given in Section 2. Section 3 describes the details of our design-to-time architecture to implement this algorithm. At this point the paper goes in two directions. First we introduce the Distributed Vehicle Monitoring Testbed (DVMT), a real application in which our design-to-time ideas have been implemented, and describe how our design-to-time problem-solver solves a real-time DVMT problem. Then Section 5 describes a series of experiments run in a simulation environment where we are able to more systematically investigate the ability of the system to react when it recognizes that a task will not meet its deadline and how that reaction is affected by parameters such as amount of shared intermediate results, monitoring rate, monitoring accuracy, and the time distribution of methods for tasks. Finally, Section 6 discusses results and future work.

2 A Design-to-time Scheduling Algorithm

This section describes the design-to-time scheduling algorithm that is used in the DVMT. Some aspects of this algorithm are specifically tailored to the periodic nature of the application. Also this algorithm takes advantage of the specific task/subtask relationship of DVMT tasks.

The algorithm is given a set of tasks, each of which has a start time, a deadline, and an estimated processing time. Associated with each task is a set of methods with their associated processing times and solution qualities. Each method consists of a set of subtasks, each of which can have a distinct earliest start time (for example, because of data arrival times or dependencies on earlier subtasks), and each of which has an estimated processing time[3].

The goal of this algorithm is to maximize the overall solution quality for the set of tasks in the given amount of time, while missing as few deadlines as possible. This problem is analogous to the basic scheduling problem of scheduling a set of tasks with different start times, end times and deadlines, which is known to be NP-Complete[12]. Because of this our design-to-time scheduling algorithm does heuristic scheduling, potentially using both application-specific and general control heuristics.

This algorithm is reapplied every time the execution subsystem signals the controller that an unpredicted event has occurred (i.e., a new task has appeared or an existing task has not behaved as predicted). *Quality* is a heuristically-derived value that incorporates both solution quality (i.e., certainty, completeness, and precision), and the importance of the task to the satisfaction of the system goal. Only approximations that do not cause predicted quality to go below that required to meet known deadlines will be considered. Figure 3 shows an example of the scheduling algorithm in action.

1. Start with the previously calculated schedule. If no previous schedule exists then assume the most complete processing method will be used for each task.

2. Examine the schedule as a whole to make sure the total times required by each task together do not exceed the time available before the tasks' deadline[4]. (In the example, the total times required for Tasks A, B, and C exceed the total available time as shown in part (a) of Figure 3.) If this constraint is violated, then keep switching a heuristically-chosen task to a faster approximation until

[3] The total estimated processing time for a method for a task is the sum of the estimated processing times of all of its subtasks.

[4] In these examples we assume that all tasks share the same deadline (or have no deadline at all). In the general case the algorithm works backward from the latest deadline applying the techniques described between each pair of deadlines.

the constraint is met. A general heuristic that is useful is to choose the task that has the lowest reduction in quality over reduction in time ratio. The result is a high level schedule that should allow all tasks to meet their deadlines unless there are lower-level task constraints that are violated. (In the example, the scheduler decides to use an approximation for Task A to bring the total required time below the available time limit as shown in part (b).)

3. Layout the schedule at the subtask level, with each distinct subtask earliest start time defining a different period. By default, place each subtask in the earliest possible period. Check each period in the schedule to make sure that it is not overloaded (i.e., make sure that it does not have more work scheduled than time available). If a period is found that does have too much work, attempt to postpone work for a heuristically-chosen task from that period to a later period. Keep doing this until no periods are overloaded. If this is not possible, go back to step 2 and choose a faster approximation for one of the tasks involved in the overloaded period. (In the example, two possible layouts of subtasks are shown with different constraints on when Task A's second subtask can start[5]. In part (c) *Period 2* is overloaded, but it is possible to postpone part of the work for Task A until *Period 3*, as shown in part (d), resulting in a feasible schedule. In part (c′), however, *Period 3* is overloaded and no postponements are possible because of the deadline. In this case the algorithm goes back to step 2 (as shown in part (d′)) and decides to use a faster approximation for Task B. This results in the feasible schedule shown in part (e′).)

4. Pass the scheduled tasks on to the execution subsystem, which will monitor task execution to ensure that the schedule is adhered to. (For the example, the execution subsystem will micro-schedule problem-solving steps for Tasks A, B, and C during the first period, interleaving the work for the tasks and ensuring that none of the tasks uses more than its scheduled time.)

This algorithm is stated as if exact values are known for the quality and time associated with each solution method for a task. More realistically a probabilistic range of values is known. The only change to the algorithm is in the heuristics used to choose which tasks to approximate and postpone. A conservative approach could calculate the worst possible change within the ranges. A liberal approach could calculate the best possible change. Another approach could use some kind of average, such as the centers of each range. Different approaches are needed for different parts of the problem and at different times[13].

In fact, variability in solution methods can lead to a need for careful monitoring of tasks. Every task should have a fall back method that produces a minimal quality result in a predictable amount of time[6]. When the decision is made to go with a higher quality method (presumably with a higher variance in time and quality), the system has to monitor the performance of the task and dynamically revise its estimate of when the task will complete. If necessary it has to decide to switch to the fall back

[5] Note that these are not two possible options that the scheduler considers, but two illustrative examples of what such a layout looks like.

[6] We have found that predictability can be increased, at least for situation assessment problems, when the character of the search space or the way it is searched is changed as a result of lowering the criteria for solution acceptability[10].

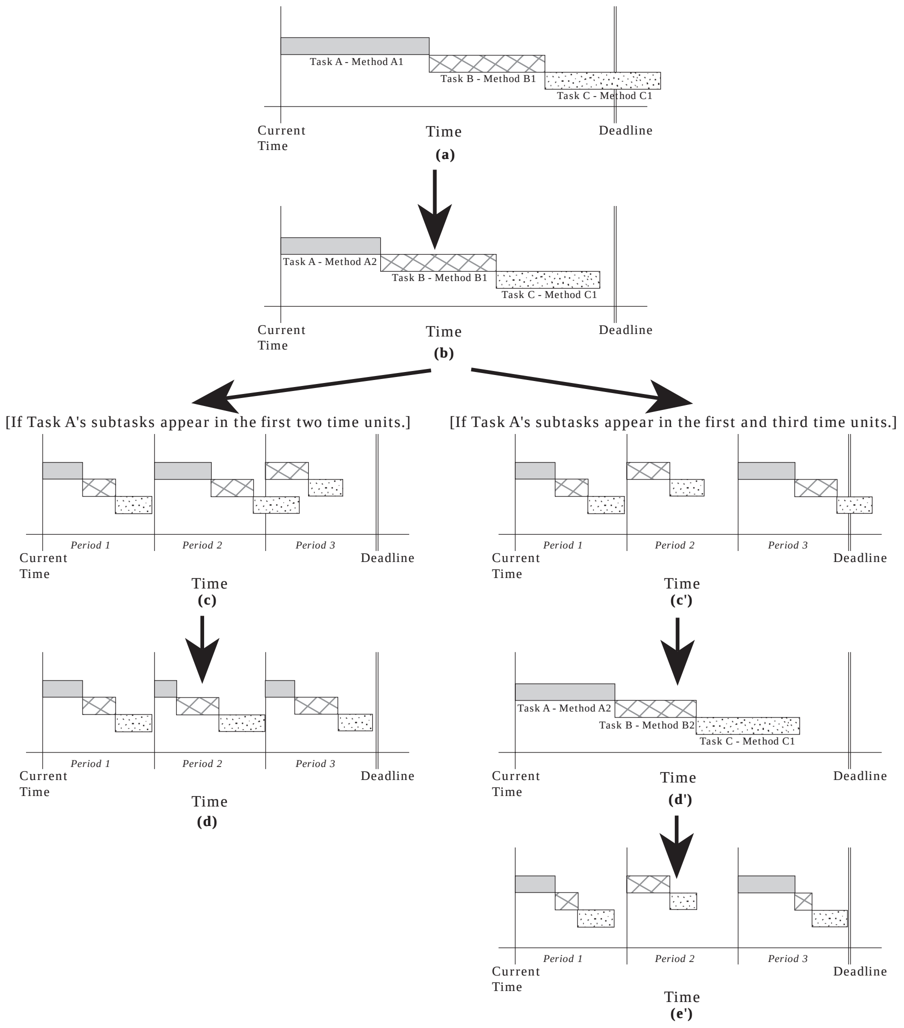

Figure 3:
Steps in design-to-time scheduling. In this example, Tasks A, B, and C each have a deadline at the time indicated by the double line labelled "Deadline". In part (a) the scheduler lays out the entire schedule to make sure the total time required does not exceed the available time. In this case the three tasks together require more time than is available, so the scheduler decides to use an approximation for Task A, as shown in part (b). Once a high-level view of the schedule is completed, the scheduler lays out the schedule for each subtask. Two examples of what such a layout might look like are given. In the left column (parts (c) and (d)) *Period 2* has more work scheduled than time available. As shown in part (d), the scheduler fixes this problem by postponing part of Task A to *Period 3*. In the right column (parts (c′) (d′) and (e′)) *Period 3* is overloaded. In this case no postponement can result in an acceptable schedule (because all tasks must complete by the deadline), so the scheduler goes back to the high level schedule and decides to use an approximation for Task B (as shown in part (d′)). It again lays out the schedule for each subtask (as shown in part (e′)) and this time the entire schedule is acceptable.

method in time to allow the fall back method to complete its processing. Note that the amount of time required by the fall back method can be dynamic and depend on how many of the intermediate results of the higher quality method it can use. Additionally the ability of a monitor to accurately predict the overall performance of a method from partial results needs to be taken into account when choosing that method, e.g., if a high quality method produces no intermediate results until processing is nearly completed and there is a high variance in how long processing takes, then that method should not be scheduled unless enough time is available for the high end of the time estimate range.

3 Our Design-to-time Architecture

Any architecture that implements the design-to-time scheduling algorithm given above must have the ability to:

- divide the problem up into tasks
- divide tasks up into subtasks that have distinct earliest start times
- control the execution of each task separately so that different tasks (including different instances of the same kind of task) can use different approximations
- estimate the duration and solution quality for a task given a particular problem solving method
- predict, where possible, the arrival of future tasks, so as to facilitate early recognition of scheduling overload situations
- monitor problem solving to notice the arrival of new tasks and notice unexpected behavior in existing tasks

This section describes how our blackboard implementation of the design-to-time architecture meets these requirements. Our design-to-time blackboard architecture implementation consists of two main components: a controller that decides which tasks to perform, what problem-solving methods to use for those tasks, and when each task should be worked on; and an execution subsystem that micro-schedules the execution of steps of tasks and ensures that tasks perform as predicted. This architecture (especially details about the execution subsystem) is described in more detail in [14].

3.1 The Controller

A design-to-time problem-solver has a *system goal*, which is the high-level goal that the system is working to achieve. Dynamically, at runtime, a plan/goal hierarchy is constructed as shown in Figure 4. This hierarchy consists of control goals, which describe subproblems that need to be solved; and a BB1-style control plan[15], which embodies the plan the agent intends to use to meet the control goals. At the lowest level of the control plan are foci which control specific execution *channels*. Multiple execution channels allow explicit, detailed control over each task separately, where a task is a unit of work that might at some point need to have some aspect of its behavior controlled separately from other units of work. Low-level control of each channel is provided by a parameterized, low-level control loop that is described in detail in [14].

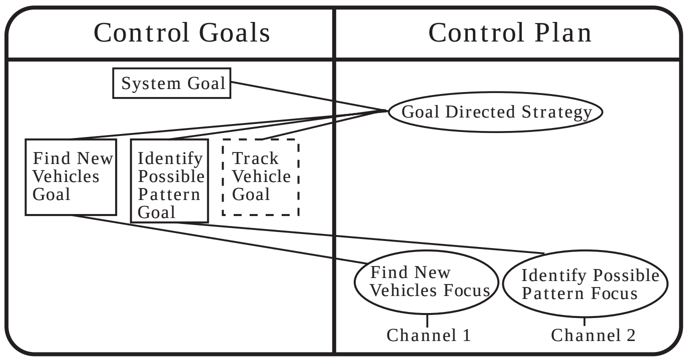

Figure 4: An example of a plan/goal hierarchy from the DVMT. The rectangles represent control goals and the ellipses represent strategies and foci. A dashed line rectangle represents a future control goal.

Channels are created and modified by the controller in response to dynamically created control goals. Channels are modified to use various approximations by the controller as required to meet timing constraints. A channel is made to use a particular approximate processing technique through the modification of the parameters of its low-level control loop.

Associated with each channel is an estimate of the amount of processing time required to perform the work for that channel. In our current research these estimates are derived from a combination of analytic techniques and the results of previous runs of the system. Estimates of the time required for work that has been partially completed is the the time already spent plus the time for problem solving steps already given to the execution subsystem. Estimates of the time required for work in the future are calculated based on the number of plausible interpretations of the data that are being considered. These time estimates change dynamically as work progresses on a task.

3.2 The Execution Subsystem

The execution subsystem controls the detailed execution of tasks. At the lowest level tasks are made up of knowledge sources (KSs). At runtime a single agenda of executable knowledge source instantiations (KSIs) is maintained and the execution subsystem chooses which action to perform next using heuristic rating criteria associated with the channels. The execution subsystem can interleave the execution of KSIs from different channels to take advantage of any useful interactions among channels. It monitors the execution of each task and signals the controller when a task takes longer than predicted. In the example below, if the processing of the data for any task takes longer than estimated, the execution subsystem signals the controller, which recalculates its resource allocations to take this into account. A more detailed description of the execution subsystem is available in Decker et al [14].

4 An Example of Design-to-time Problem-Solving

This section first describes our application environment, then shows how the system works when enough resources are available for all processing, and finally illustrates how problem solving changes when resources become more scarce.

4.1 The Distributed Vehicle Monitoring Testbed

The Distributed Vehicle Monitoring Testbed (DVMT)[16] simulates a network of vehicle monitoring nodes, where each node is a design-to-time problem-solver that analyzes acoustically-sensed data in an attempt to identify, locate, and track patterns of vehicles moving through a two-dimensional space. The DVMT is implemented as a blackboard system, and domain knowledge sources perform the basic problem-solving tasks of extending and refining partial solutions, or *hypotheses*. To solve a problem, the system must choose from among several different general strategies and fine tune them, including the choice of different strategies for different kinds of data and different strategies at different stages of processing. Also available in the DVMT are approximations for various problem-solving tasks[10]. Data from the sensors arrives periodically, and the time between two data arrivals is known as a *sensor cycle*.

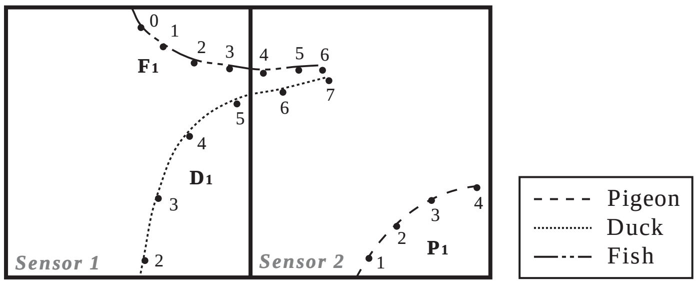

Figure 5: Real-time example environment.

Figure 5 shows an example DVMT environment for a single DVMT agent[7] The large dots along the lines represent the location of the object at the sensor time given by the adjoining number. The two rectangles labelled *Sensor 1* and *Sensor 2* represent the ranges of the two fixed sensors associated with this DVMT node. The system is able to predict fairly accurately when a known object will move outside of sensor range, based on its current heading and velocity. The system knows about three kinds of patterns among its objects: a duck attacking a fish, and a duck or pigeon meandering.

Associated with this environment is a system goal, in which is encoded the priorities and parameters of the system. Some of what is encoded includes:

- If at all possible try to identify and track all objects that pass through the range of the sensors.
- Ducks attacking fish are more important than meandering objects.

[7] All of our DVMT design-to-time research thus far has used a single-agent DVMT. However we have begun investigating multi-agent design-to-time scheduling in our simulator[17, 18, 19].

- There is a deadline that fish must be warned that they are part of a duck attacking fish pattern in at most six sensor-cycles from when the later of the two vehicles comes within sensor range.

4.2 An example problem with adequate resources

Before problem solving actually begins, control knowledge sources post the system goal, which triggers the posting of a top-level strategy for meeting that goal. In this example a *goal-directed* top-level strategy is posted[8]. Additional control knowledge sources elaborate this strategy into default heuristics for controlling the execution of control knowledge sources and an initial control goal of finding any new vehicles that appear. This leads to the creation of a *find-new-vehicles* channel that is constantly looking for new vehicles that are not already being worked on by an existing channel. At the beginning of problem solving this is the only active channel and it is projected to require no resources (because there is no known work for it to do.)

In the example environment of Figure 4 three objects appear: a fish at sensor time 0, a pigeon at sensor time 1, and a duck at sensor time 2. Figures 6 and 7 show the scheduling process when adequate resources are available. At the beginning of sensor cycle 0 the *find-new-vehicles* channel receives the signal level data associated with the fish. The appearance of this data causes domain KSIs to appear on the domain KSI queue. Together these KSIs (and the KSIs that they will trigger to continue processing the data from sensor cycle 0 up to the track level) make up the work for the *find-new-vehicles* channel during sensor cycle 0. The appearance of the domain KSIs causes the execution subsystem to project a change in the workload of the *find-new-vehicles* channel, which in turn causes the design-to-time scheduling algorithm to be invoked. This algorithm projects the total time for the *find-new-vehicles* channel (which is well below the total available time), then checks that no sensor cycle is overloaded (which they are not), then passes the KSIs associated with the new schedule to the execution subsystem. At this point the execution subsystem begins scheduling domain KSIs for immediate execution. Every KSI has a dynamically calculated expected duration. After each KSI execution a monitor compares the actual progress of the system to the expected progress, using these expected duration. If the performance of the system goes outside allowable tolerances, the design-to-time scheduler is reinvoked to reschedule as necessary.

When the domain KSIs have processed the data up to the track level, this satisfies the control goal of the *find-new-vehicles* channel (which is to recognize when new vehicles appear and process their data for one sensor cycle). Generic control KSs notice when control goals are satisfied by regularly monitoring each active control goal's satisfaction function. Control KSs associated with the top-level *goal-directed* strategy then post the next part of the control plan, which is a control goal to identify any possible patterns the new vehicle might be involved in. The posting of this control goal triggers a control KS that creates a new *identify-possible-patterns* channel to identify any possible patterns the fish might be involved in. At this point the processing of data from sensor cycle 0 is complete.

Sensor cycle 1 contains data from two objects, the fish that has already been tentatively identified and a newly arrived pigeon. The *identify-possible-patterns* channel

[8]Although we do not discuss them in this paper, other approximate processing strategies could be used in this example including clustering of noisy data and the skipping of data from every other sensor cycle.

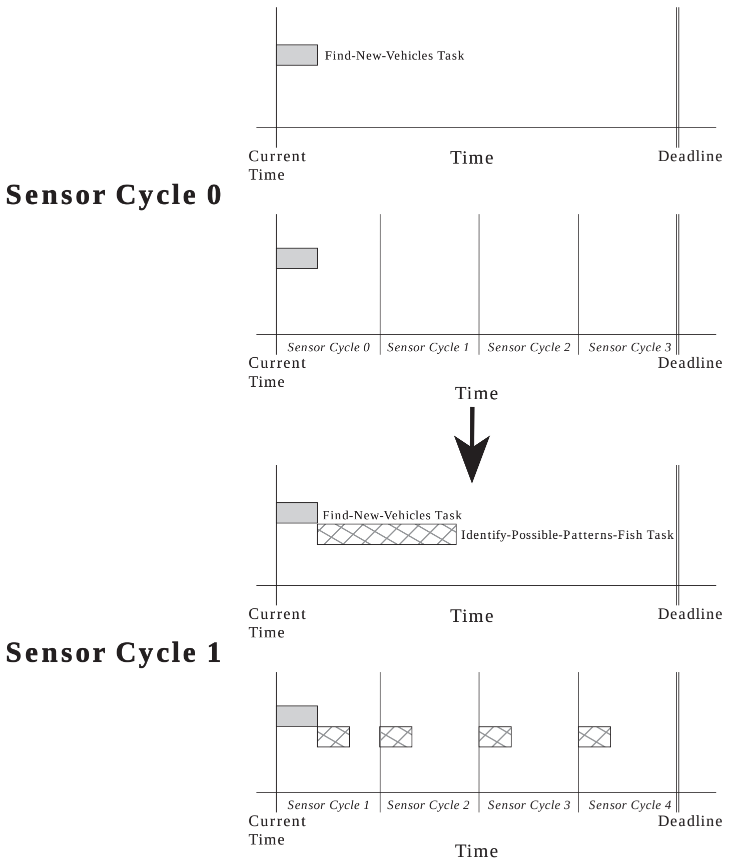

Figure 6: The design-to-time algorithm scheduling sensor cycles 0 and 1.

accepts the fish signals, because they are spatially close to the previous fish signals and because they are of a type that can be associated with fish. However, this channel does not accept the pigeon data, which is then picked up by the *find-new-vehicles* channel. The design-to-time scheduler is invoked because of the work associated with the unexpected new pigeon data. It has no problem scheduling the tasks, both at a high level and for each discrete sensor cycle. Both channels process their respective data in the same way as in the previous cycle, with the processing of the pigeon data resulting in a new *identify-possible-patterns* channel being created to identify any patterns that the pigeon might be involved in.

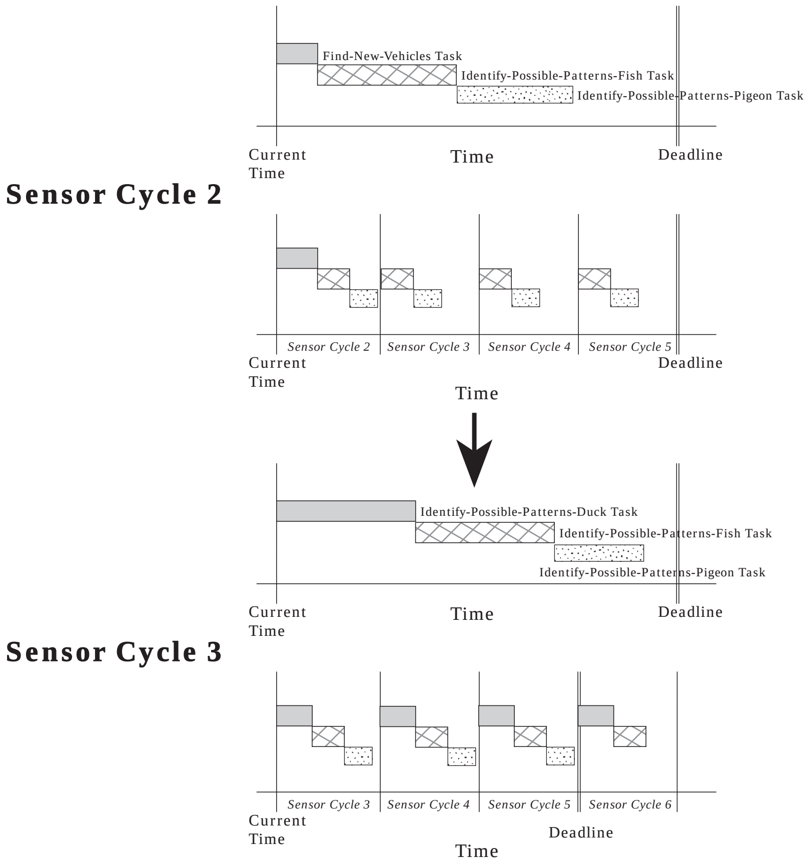

Figure 7: The design-to-time algorithm scheduling sensor cycles 2 and 3 when adequate time is available.

Processing during sensor cycle 2 proceeds similarly, with processing of fish and pigeon data continuing in their respective *identify-possible-pattern's* channels and the *find-new-vehicles* channel noticing the appearance of the duck, leading to the creation of a third *identify-possible-patterns* channel for the duck. The appearance of the duck causes a deadline to be created to warn the fish if it is involved in a duck-attacking-fish pattern by sensor cycle 7 (because the system goal specifies that the fish must be warned within 6 sensor cycles of both objects in the pattern coming within sensor range.) The continued processing of the data for the pigeon allows the scheduler to predict that the pigeon will leave sensor range about sensor cycle 4, based on current direction and velocity.

The system goal specifies that four sensor cycles of data are required to confirm the involvement of vehicles in a pattern. During sensor cycle 5 enough data has been processed to confirm that the duck and fish are involved in a duck-attacking-fish pattern. This is noticed by a control KS, which issues a warning to the fish. Processing in all channels continues until all available data has been processed.

4.3 How the system reacts when resources are scarce

The way that we experiment with reducing system resources is to reduce the sensor cycle length, i.e., the time available to process data between the arrivals of new data. As the sensor cycle length is reduced the controller has to take action, because not enough time is available to completely perform all tasks. It first notices a problem during sensor cycle 2 when the appearance of the duck causes the workload for sensor cycle 2 to exceed the available time. Figure 8 shows how the design-to-time algorithm reacts when not enough resources are available in sensor cycles 2 and 3. Note that the high-level part of the scheduler does not notice a problem with the total time, because of the projected exit of the pigeon after sensor cycle 4. Because we are less concerned about pigeons and because there is no deadline associated with the pigeon tracking, the scheduler decides to postpone part of the work of tracking the pigeon for each of sensor cycles 2, 3, and 4 to later sensor cycles. This allows it to devote adequate processing to meet the deadline to warn the fish about the attacking duck by sensor cycle 5. In this case we are able to solve the real-time scheduling problem with one fix. In general, we may need to iterate through the scheduling algorithm several times to find a satisfactory schedule.

If sensor cycle length is reduced even more, the high level part of the scheduler will notice that the total time required exceeds the time available and decide to use faster approximations for some or all of the *identify-possible-patterns* channels. One such approximation for the DVMT, known as *level-hopping*, reduces duration by hopping over levels of abstraction in the sensor interpretation. This approximation has the effect of decreasing the certainty and precision of the resulting interpretation, thus reducing quality. As a last resort, if the sensor cycle length is reduced to a very short amount of time, the controller turns off the *find-new-vehicles* channel. This will have the effect of completely ignoring the appearance of any new vehicles, but allows enough time for the the deadline associated with the known vehicle data to be processed.

This rescheduling solves the real-time problem because it reduces the workload in each sensor cycle until it can be performed in the time available, and it meets the required deadline. In this example we see the system designing and refining a solution to a real-time problem. It designs a solution that takes advantage of all of the resources available, and refines that solution as the expected workload changes when new vehicles appear. It combines the use of approximations and postponements as appropriate to best use available resources.

5 Experimental Results

When building a design-to-time problem-solver several questions must be answered. The answers to these questions can be thought of either as constraints on how to best configure the problem or criteria for determining whether design-to-time is the most

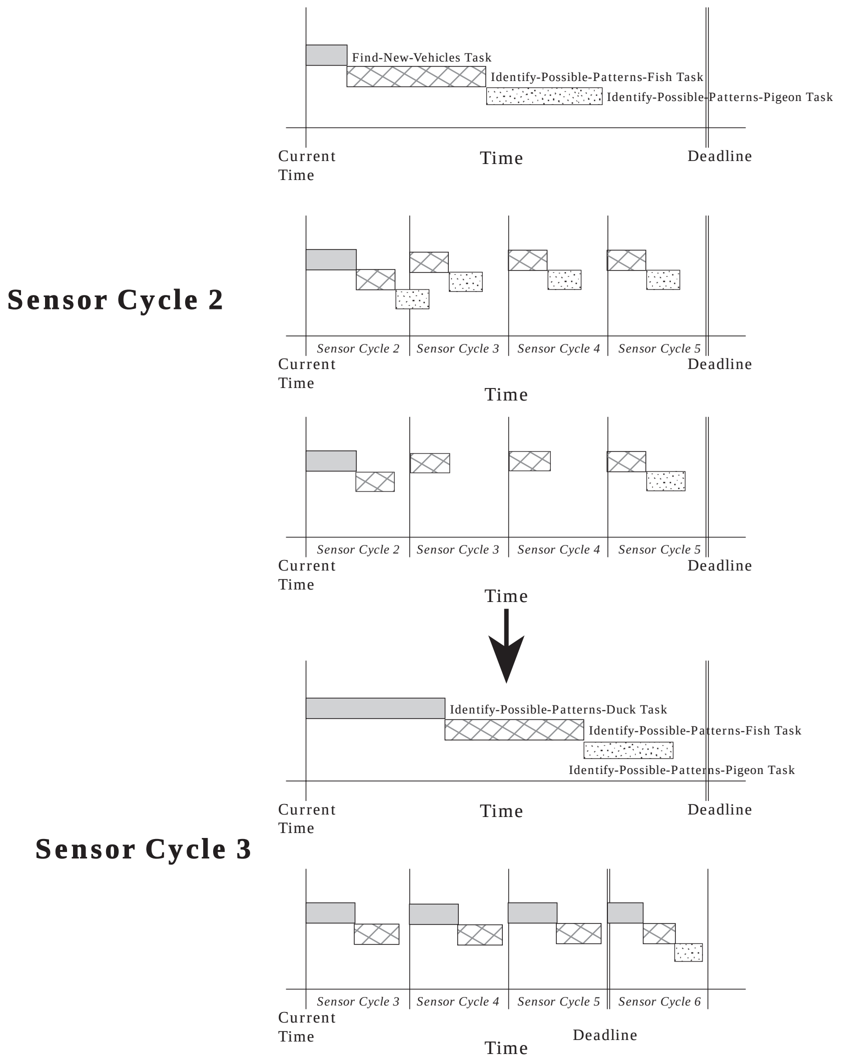

Figure 8: The design-to-time algorithm scheduling sensor cycles 2 and 3 when resources are scarce.

appropriate problem-solving approach for the problem.

- How do you choose a set of solution methods?
  - How many approximations are best?
  - How much variance in duration and quality estimates is tolerable?
  - How fast a fall back method is necessary to reduce missed deadlines to a tolerable level?
- How frequently should you monitor task execution?
  - What is the cost/accuracy tradeoff in monitoring?
  - What is the effect of the amount of sharable intermediate results among solution methods on the system performance?
- What do you do if the duration/quality variance is too high?
  - How is monitoring frequency/accuracy affected by biases in estimated duration?

This section describes a series of experiments run on a simulator. In these experiments a parameterized simulation environment is used to generate sets of tasks for a design-to-time scheduler. The actual duration and quality values for each method for each task are randomly generated from normal distributions with means equal to the estimated value and variances as specified for the method/task combination. Inter-arrival times for each task are generated by an exponential distribution whose mean varies by task type.

All of these experiments were run in an environment consisting of two task types, one with slightly longer mean duration and higher mean quality solution methods. The expected qualities and durations for each solution method for each of these task types is shown in Figure 9. As is shown, the distribution of quality/time points is roughly linear. The distribution of arrivals is weighted so that about 60% of arriving tasks are of type 1 and 40% are of type 2. Unless otherwise stated, the variance in duration and quality for individual tasks were about 50% and 10% of the mean respectively, 50% of the intermediate results were usable when methods were switched, and monitoring was done at 25%, 50%, and 75% of the expected task duration. Each monitoring observation is a random variable drawn from a normal distribution. The mean of the distribution is the actual simulated result so far and the variance decreases linearly with remaining task duration and exponentially with solution method quality. The monitoring observation is used to estimate a total duration for the method, which is compared to the available time for the method as determined by the scheduler. If the available time is exceeded, monitoring recommends switching to a faster method. Task execution is non-interruptible, except for monitoring (i.e., the scheduler does not even notice the arrival of new tasks until the execution of the current task is completed.) Task deadlines in each experiment were generated randomly from a distribution consisting of deadlines varying from 1.3 to 5 times the duration of the highest quality solution method for the task. These deadlines were relative to the arrival of the task to the system, not the time that the scheduler notices the arrival (which could be much later because the scheduler only notices new task arrivals between task executions).

Low, medium, and high loads are discussed in several of the experiments. They are controlled by varying the expected arrival rate of each task type. Low, medium, and high correspond roughly to a total utilization in which the use of the best method for each task would result in utilizations of 75%, 150% and 300% respectively. For the simulation length used in these experiments this corresponds to task sets roughly of size 75, 150, and 300 respectively. Each experiment was run on at least 5 distinct task sets.

The purpose of these experiments is to help answer the above questions. The goal is to understand some of the relationships among the various parameters of the system. The experiments produce empirical correlations among these parameters. We expect in the future to be able to understand these correlations analytically.

One potential concern in using a simulator is that crucial aspects of real problems will be abstracted away in the simulation. We have tried to address that concern by including as many aspects of real DVMT design-to-time tasks as possible. One issue that is only partially addressed in this simulation is subtask interactions. DVMT tasks are broken up into subtasks that can have interactions with other subtasks both from their parent tasks and other tasks. Examples of these interactions include shared subproblems (which only have to be solved once for a set of subtasks), order constraints (where one subtask has to be completed before another can begin), and overlapping problems (where one subtask can provide results that improve the quality of another subtask's solution). These interactions are used by the DVMT scheduler to order the execution of subtasks[20]. Another aspect of real problems that is not completely handled is penalties associated with missing deadlines. The effect of missing particular deadlines is domain-specific and can range from catastrophic to merely inconvenient. Our current scheduler does not have a model of the cost of missing particular deadlines; it just assumes that missing any deadline is to be avoided as much as possible. For this reason our experimental results are reported in terms of both average quality per task and percent of deadlines missed, with no attempt to integrate the two pieces of information (as would need to be done for any actual application.)

5.1 Availability of fast fall back methods

This experiment investigates the importance of the availability of fast fall back methods on the performance of the scheduler. In this experiment the set of methods available to the scheduler for each task type is systematically reduced by removing the fastest method (i.e., the system is run with all 8 methods for each task as shown in Figure 9, then with the slowest 7 methods, then with the slowest 6 methods, . . .). Figure 10 shows the effect of removing faster methods from the method set on the average quality produced per task and on the percentage of deadlines missed.

As expected, this experiment shows that the lack of fast fall back methods results in a significant increase in the percentage of missed deadlines. It also shows that the average quality produced per task increases for awhile until it is overwhelmed by the zero qualities associated with all of the tasks that missed deadlines. In future experiments we would like to understand how the quality results in this experiment are affected by different distributions of methods, for example concave or convex (rather than linear) quality/duration tradeoffs.

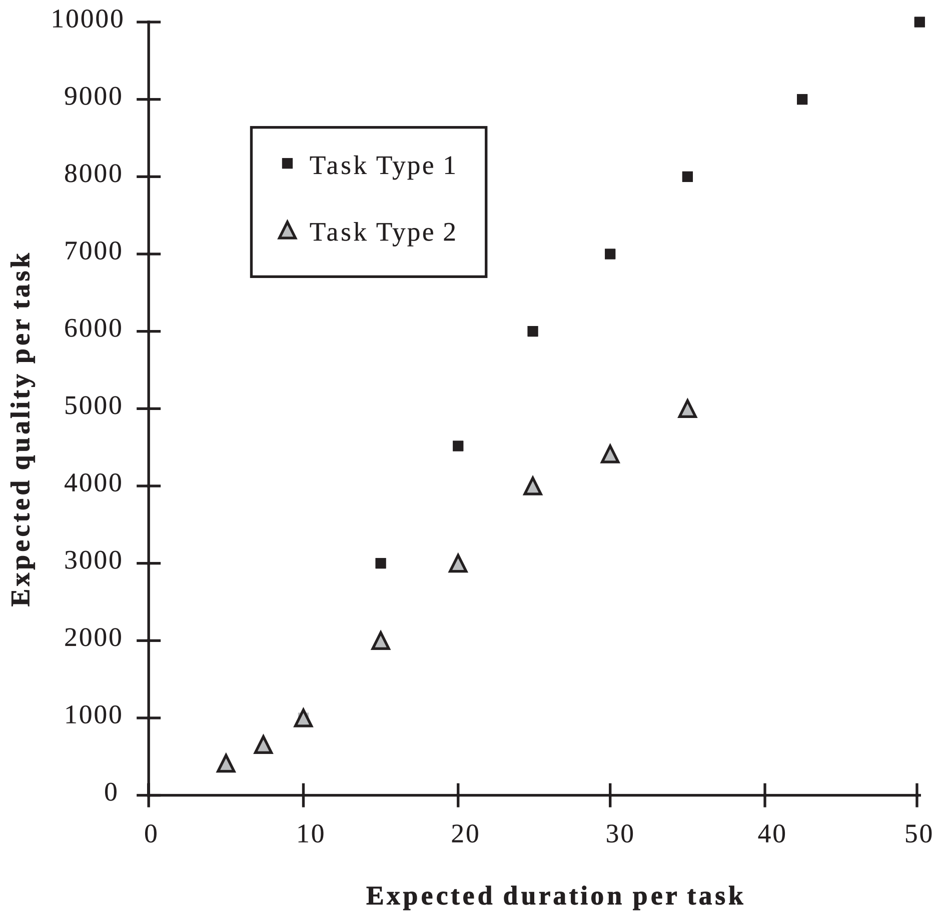

Figure 9: The expected quality and duration values for the task types used in these experiments.

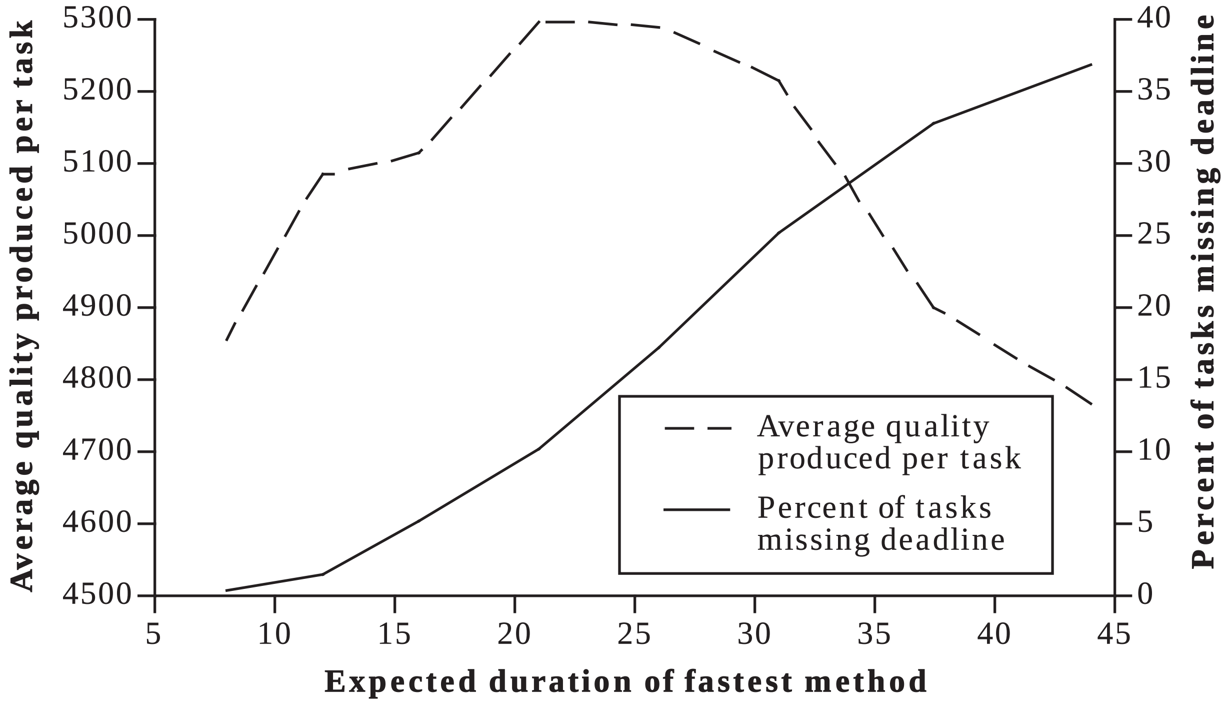

Figure 10: Quality produced and percentage of deadlines missed as the expected duration of the fastest method increases.

5.2 Frequency of monitoring and bias in duration estimates

In these experiments the rate at which tasks are monitored is varied, as well as the bias in the duration estimates. The rate of monitoring is measured as the number of times a task execution is monitored. These monitorings are evenly distributed along the expected duration of a task (e.g., a task with two monitoring points is monitored at 33% and 67% of expected duration.) No cost is associated with monitoring in this experiment. We also introduce a bias in how the expected duration for a method relates to the actual duration. In all of our experiments the actual duration is calculated from a normal distribution with a mean of the expected duration and a parameterized variance which is usually set to 50% of expected duration. Bias is introduced by uniformly varying the expected duration seen by the scheduler (by multiplying it by a duration bias parameter.)

Figures 11 and 12 show the average quality produced and percentage of deadlines missed, respectively, as both the rate of monitoring and the duration bias parameter are varied. When the bias is low the scheduler tends to underestimate the duration of tasks; when the bias is high the scheduler tends to overestimate the duration of tasks. In this experiment the term small is used to denote a bias of 20% and the term large is used to denote a bias of 40%.

These results suggest that under all situations at least a little monitoring results in significantly fewer missed deadlines. As the system is biased toward underestimating the actual duration of methods, monitoring becomes especially useful. In this situation increased rates of monitoring tend to result in increased average task quality. On the other hand, when the actual durations are overestimated, increased rates of monitoring

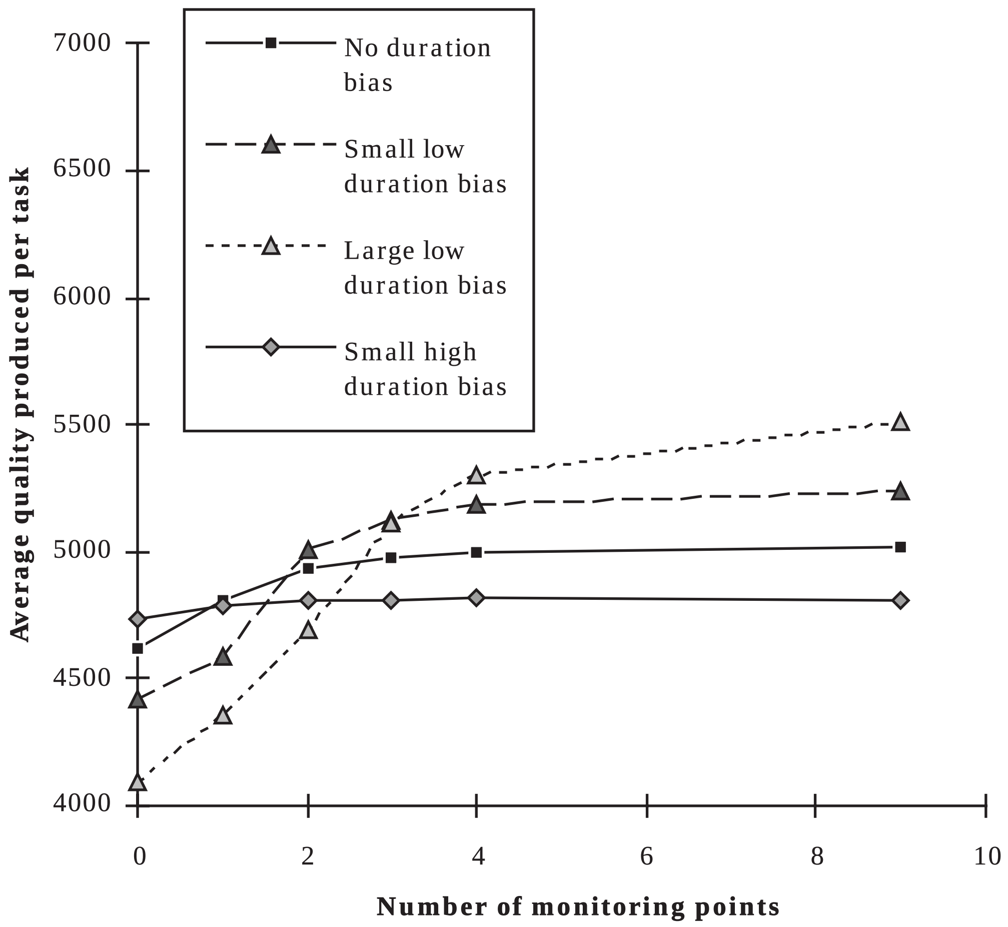

Figure 11: Average quality produced per task as the rate of monitoring is varied for four settings of the duration bias parameter.

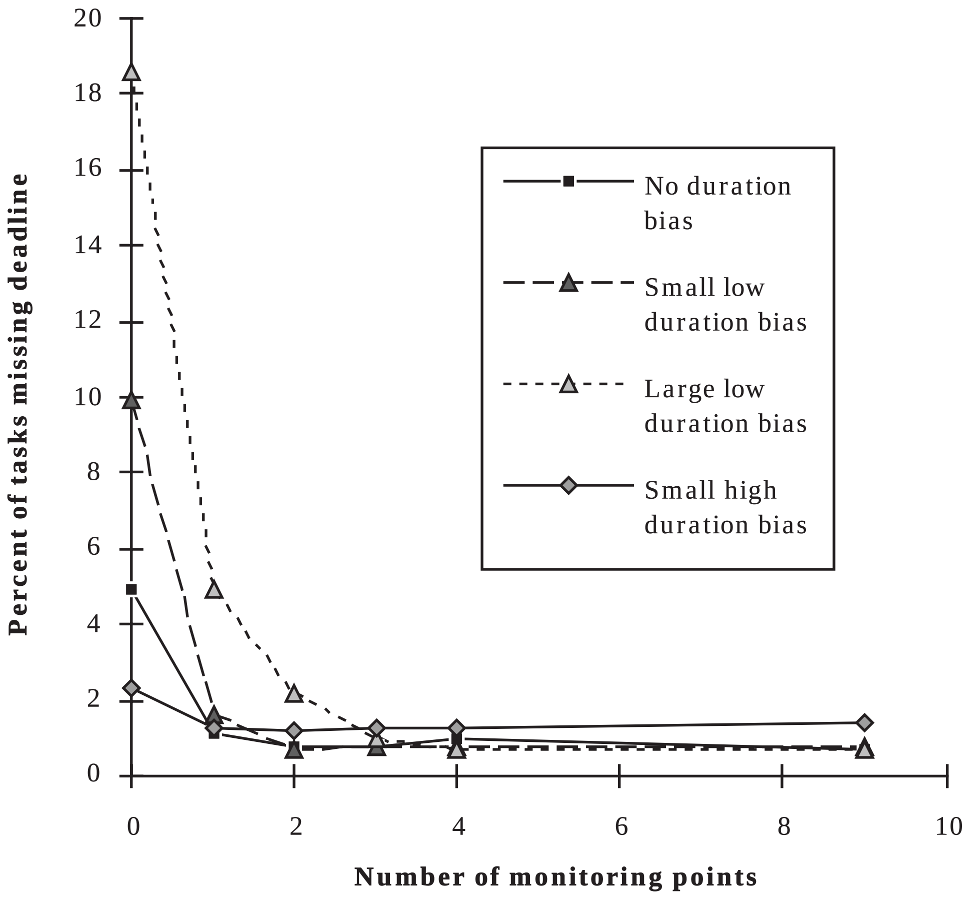

Figure 12: Percentage of tasks missing deadlines as the rate of monitoring is varied for four settings of the duration bias parameter.

have almost no effect.

This experiment shows that in many situations monitoring has a beneficial effect. These results were generated at a moderate load for the system. A somewhat unexpected result (not shown in these graphs) is that monitoring seems to be most useful at moderate loads. At low loads monitoring's usefulness declines—even if methods have much longer durations than expected—because there is usually enough slack time available to allow these deviations. Monitoring also has a reduced usefulness at high loads because—especially during highly overloaded bursts of task arrivals—most tasks are already executing their fastest method, so the only way that monitoring can improve performance is by noticing when tasks will exceed their deadline, even with their fastest method, so that time is not wasted executing them. This is not to suggest that monitoring at relatively low and high loads has no benefit, only that the most noticeable benefit seems to happen at medium loads.

5.3 Amount of shared intermediate results

In this experiment we investigate the usefulness of shared intermediate results. Whenever monitoring decides that the current method for a task is performing inadequately and the task should switch to a faster method, some amount of intermediate results may be available for the new method. We measure the amount of shared intermediate results as a percentage of the duration spent on previous methods that can be deducted from the duration of the new method. In this experiment that percentage was varied from 0% to 100%.

Figures 13 and 14 show the average quality produced and percentage of deadlines missed respectively as the rate of monitoring is varied for shared intermediate result percentages of 0%, 50% and 100%. These results suggest that the availability of shared intermediate results leads to improved performance in terms of increased quality, but has little effect on the percentage of missed deadlines, however the magnitude of the improvement is relatively small. Shared intermediate results appear to be useful exactly in those situations where monitoring is most effective—when the load is neither too low to force enough method changes nor too high to preclude the use of better methods, at least initially.

6 Discussion and Future Directions

This paper describes the design-to-time approach to real-time problem solving, demonstrates its feasibility in a complex real-time application, and describes simulation experiments that vary design-to-time scheduling parameters.

The simulator results are just a beginning, but they begin to suggest how such simulations might be useful, both for confirming intuitions about what system parameters are important for scheduling (e.g., showing that monitoring almost always provides a reduction in missed deadlines), and for occasionally surprising us with unexpected interactions (e.g., monitoring may only have major benefit at medium loads, possibly suggesting that monitoring rates should be reduced at high and low loads.)

Our intent with the simulation work is to develop a theory of the characteristics of the design-to-time approach that answers the questions about required predictability outlined in Section 1. These simulation experiments indicate that our description of

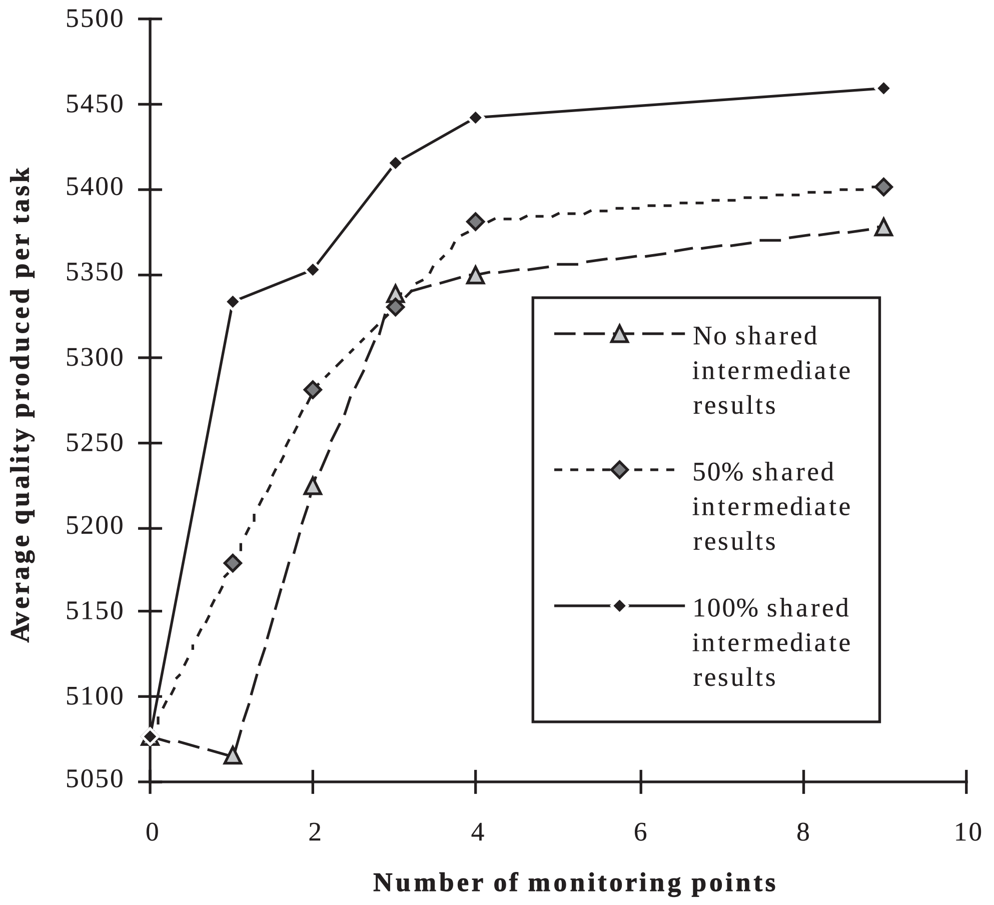

Figure 13: Average solution quality produced as the rate of monitoring increases for three amounts of shared intermediate results.

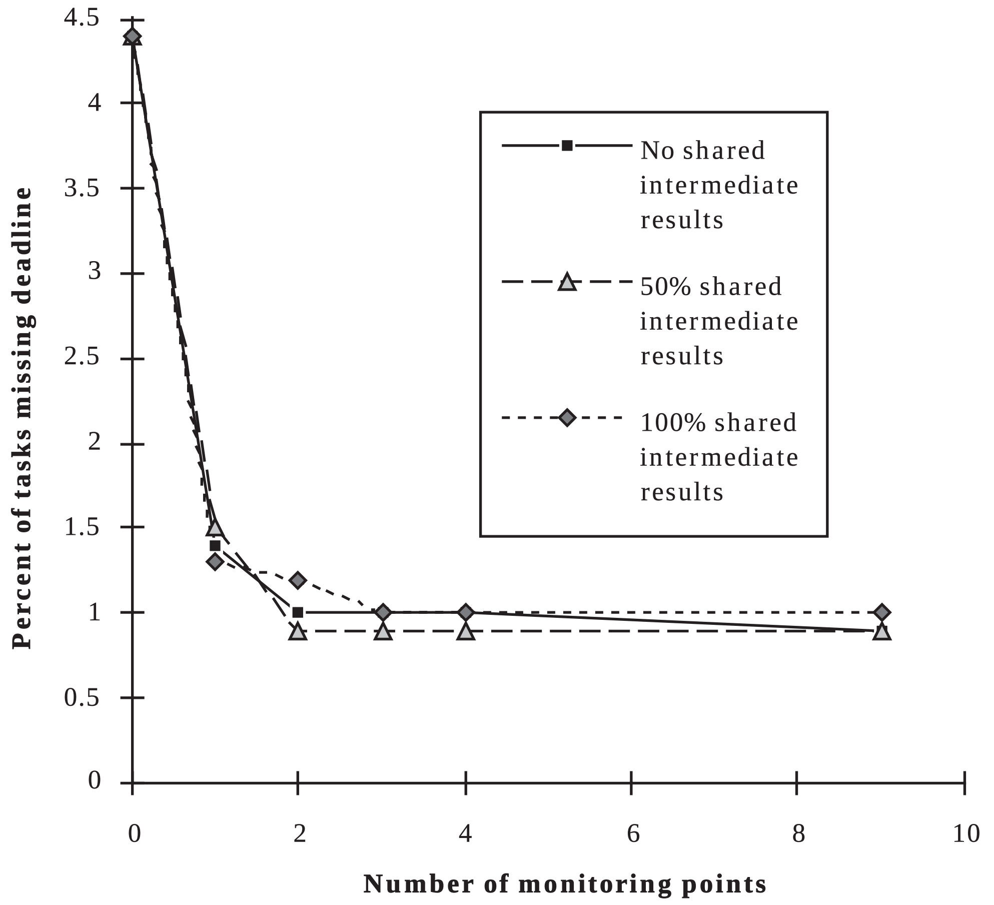

Figure 14: Percentage of tasks missing deadlines as the rate of monitoring increases for three amounts of shared intermediate results.

the techniques that can be used to react to unpredictability in task durations and deadlines have promise. It appears that—at least under some circumstances—monitoring, sharing intermediate results, and having fast fall back methods make it possible to overcome unpredictability.

The space of possible experiments is very large and in this paper we present only a very limited subset. In the future we would like to investigate the effect of associating a *cost* with monitoring. We also want to explore more fully the tradeoffs between scheduling time and the quality of the schedule produced. Just as with monitoring, we might want to have a range of schedulers and understand what the effect of various parameters is on the usefulness of a particular scheduler.

In the future we plan to extend our simulation work extensively. We have recently began work on a much more detailed simulator that represents the complex interactions that can exist among tasks [21, 22]. This increased sophistication in our simulator will both allow and encourage increased sophistication in our scheduler. Eventually we hope to take the knowledge about scheduling that we acquire in this process and integrate it back into the DVMT or another complex AI application.

Acknowledgments

We would like to thank Keith Decker, Krithi Ramamritham and Jack Stankovic for helpful comments on the paper. We would also like to thank Keith Decker and Marty Humphrey for their work on the DVMT.

References

[1] V. R. Lesser, J. Pavlin, and E. Durfee, “Approximate processing in real-time problem solving,” *AI Magazine*, vol. 9, pp. 49–61, Spring 1988.

[2] B. D’Ambrosio, “Resource bounded-agents in an uncertain world,” in *Proceedings of the Workshop on Real-Time Artificial Intelligence Problems*, (IJCAI-89, Detroit), Aug. 1989.

[3] P. P. Bonissone and P. C. Halverson, “Time-constrained reasoning under uncertainty,” *The Journal of Real-Time Systems*, vol. 2, no. 1/2, pp. 25–45, 1990.

[4] M. Boddy and T. Dean, “Solving time-dependent planning problems,” in *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, (Detroit, MI), Aug. 1989.

[5] T. Dean and M. Boddy, “An analysis of time-dependent planning,” in *Proceedings of the Seventh National Conference on Artificial Intelligence*, (St. Paul, Minnesota), pp. 49–54, Aug. 1988.

[6] S. J. Russell and S. Zilberstein, “Composing real-time systems,” in *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, (Sydney, Australia), pp. 212–217, Aug. 1991.

[7] J. W. S. Liu, K. J. Lin, W. K. Shih, A. C. Yu, J. Y. Chung, and W. Zhao, “Algorithms for scheduling imprecise computations,” *IEEE Computer*, vol. 24, pp. 58–68, May 1991.

[8] C. Marlin, W. Zhao, G. Doherty, and A. Bohonis, “GARTL: A real-time programming language based on multi-version computation,” in *Proceedings of the International Conference on Computer Languages*, (New Orleans, LA), pp. 107–115, Mar. 1990.

[9] K. B. Kenny and K.-J. Lin, “Building flexible real-time systems using the Flex language,” *IEEE Computer*, vol. 24, pp. 70–78, May 1991.

[10] K. S. Decker, V. R. Lesser, and R. C. Whitehair, “Extending a blackboard architecture for approximate processing,” *The Journal of Real-Time Systems*, vol. 2, no. 1/2, pp. 47–79, 1990.

[11] K. S. Decker, A. J. Garvey, M. A. Humphrey, and V. R. Lesser, “Control heuristics for scheduling in a parallel blackboard system,” *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 7, no. 2, pp. 243–264, 1993.

[12] R. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. R. Kan, “Optimization and approximation in deterministic sequencing and scheduling: A survey,” in *Discrete Optimization II* (P. L. Hammer, E. L. Johnson, and B. H. Korte, eds.), North-Holland Publishing Company, 1979.

[13] P. P. Bonissone, S. S. Gans, and K. S. Decker, “RUM: A layered architecture for reasoning with uncertainty,” in *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, Aug. 1987.

[14] K. S. Decker, A. J. Garvey, M. A. Humphrey, and V. R. Lesser, "A real-time control architecture for an approximate processing blackboard system," *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 7, no. 2, pp. 265–284, 1993.

[15] B. Hayes-Roth, "A blackboard architecture for control," *Artificial Intelligence*, vol. 26, pp. 251–321, 1985.

[16] V. R. Lesser and D. D. Corkill, "The distributed vehicle monitoring testbed," *AI Magazine*, vol. 4, pp. 63–109, Fall 1983.

[17] K. S. Decker and V. R. Lesser, "An approach to analyzing the need for meta-level communication," in *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence,* (Chambéry, France), Aug. 1993.

[18] K. S. Decker and V. R. Lesser, "A one-shot dynamic coordination algorithm for distributed sensor networks," in *Proceedings of the Eleventh National Conference on Artificial Intelligence,* (Washington), pp. 210–216, July 1993.

[19] K. S. Decker and V. R. Lesser, "Quantitative modeling of complex computational task environments," in *Proceedings of the Eleventh National Conference on Artificial Intelligence*, (Washington), pp. 217–224, July 1993.

[20] K. Decker, A. Garvey, M. Humphrey, and V. Lesser, "Effects of parallelism on blackboard system scheduling," in *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence,* (Sydney, Australia), Aug. 1991.

[21] K. S. Decker, A. J. Garvey, V. R. Lesser, and M. A. Humphrey, "An approach to modeling environment and task characteristics for coordination," in *Enterprise Integration Modeling: Proceedings of the First International Conference* (C. J. Petrie, Jr., ed.), pp. 379–388, MIT Press, 1992.

[22] A. Garvey, M. Humphrey, and V. Lesser, "Task interdependencies in design-to-time real-time scheduling," in *Proceedings of the Eleventh National Conference on Artificial Intelligence,* (Washington, D.C.), pp. 580–585, July 1993.