Design-to-time Scheduling with Uncertainty [1]

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

UMass Computer Science Technical Report 95–03
January 9, 1995

Abstract

Design-to-time real-time scheduling is an approach to solving time-sensitive problems where multiple methods are available for many subproblems. This paper examines design-to-time scheduling problems where the value (quality) and duration of methods is uncertain. We first describe our basic approach to scheduling design-to-time problems, describe the necessary extensions for scheduling uncertain tasks, and describe the effects of uncertainty on the difficulty of scheduling and on the quality of the schedules produced.

[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

Design-to-time[Garvey and Lesser, 1993, Garvey *et al.*, 1993, Garvey *et al.*, 1994] is an approach to problem solving that involves designing a solution plan dynamically at runtime that uses all of the time available to find as good a solution as it can. Because the problems it is solving are generally intractable and because time spent finding solution plans is time that could otherwise be spent solving the actual problem, it is a satisficing approach. In our design-to-time work, problem solving is modeled as a set of interrelated computational tasks, with alternative ways of accomplishing the overall task and not a single "right" answer, but a range answers of different qualities, where the overall quality of a problem solution is a function of the quality of individual subtasks. This scheduling of such prespecified task structures that contain alternatives represents a simplified version of the scheduling problem in which a planner for dynamically generated process plans is tightly integrated with a scheduler as described by Smith[Smith, 1993]. Our approach is quite different from his, but the basic idea of problems that require both deciding "what" to do and deciding "when" to do it is shared. Another major focus of our work on design-to-time is on taking interactions among subproblems into account when building solution plans, both "hard" interactions that must be heeded to find correct solutions (e.g., hard precedence constraints), and "soft" interactions that can improve (or hinder) performance (e.g., facilitates constraints[Decker and Lesser, 1993]).

Previous work on design-to-time scheduling has examined scheduling in an actual application[Garvey and Lesser, 1993], in a simulation of an application with particularly circumscribed interactions among tasks[Garvey *et al.*, 1993], and in a distributed simulation[Garvey *et al.*, 1994]. In the work on the actual application (the Distributed Vehicle Monitoring Testbed) the focus was on scheduling in situations where approximations were available for some of the tasks of the system, and interactions among tasks were assumed to be minimal. The solution involved building periodic schedules, monitoring task execution (because tasks did not always perform as expected, partially because predictions were only approximate and partially because interactions *did* exist even if it was assumed they did not), and dynamically rescheduling as necessary. The initial simulation work looked at a slightly simplified model of the actual application with a focus on the interactions among subtasks. Both hard and soft interactions were allowed. In this work an optimal algorithm was found when interactions were limited to particular parts of the overall problem. Most recently, design-to-time research has looked at distributed scheduling. In this work no limitations were placed on interactions (thus necessitating heuristic scheduling) and problems were divided (usually with some overlap) among multiple agents. Coordination between agents was handled using the Generalized Partial Global Planning (GPGP) approach[Decker and Lesser, 1992]. At the request of the coordination algorithm, agents would make *commitments* to completing particular methods by particular times. The heuristic scheduling in this work was done using an early version of the basic algorithm described in this paper.

An example of a problem to be solved by the design-to-time scheduling algorithm is given in Figure 1. This representation of a task structure is based on the TÆMS modeling framework[Decker and Lesser, 1993]. In a TÆMS task structure the leaves of the graph represent executable computations (known as *methods*) and the nonleaf nodes represent tasks that achieve quality as a function of the quality of their subtasks. Each separate graph is known as a *task group* and represents a single independent problem to be solved. Each task group has

a deadline by which all computation on that task group must be completed. Non-parent-child connections between tasks and methods represent interactions, such as *enables* (Task A must have quality greater than a threshold before Method B can correctly begin execution) and *facilitates* (if Task A has quality greater than a threshold, then Method B will have reduced duration and/or increased quality).

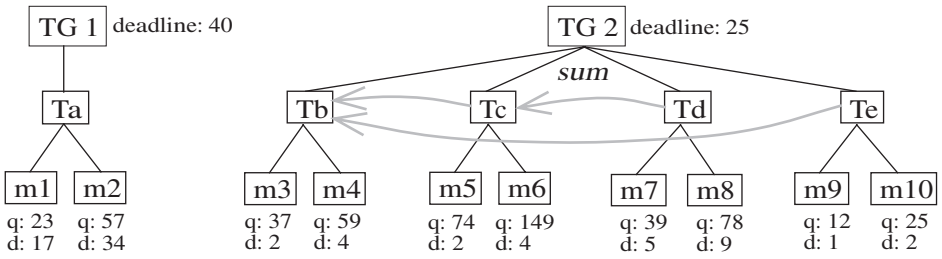

Figure 1: An example of TÆMS task structure to be scheduled. The black lines represent task/subtask connections, while the gray lines represent facilitates relationships.

Given task structures of this form, the job of the design-to-time scheduling algorithm is to dynamically build schedules with a preference for schedules that (in order of importance) achieve nonzero quality for all task groups, maximize the sum of the qualities of all task groups, and minimize the total duration of method executions. The result of this scheduling algorithm is a schedule that specifies what methods to execute, when to execute them, and what values are expected from that execution. Figure 2 shows a schedule for the task structure in Figure 1.

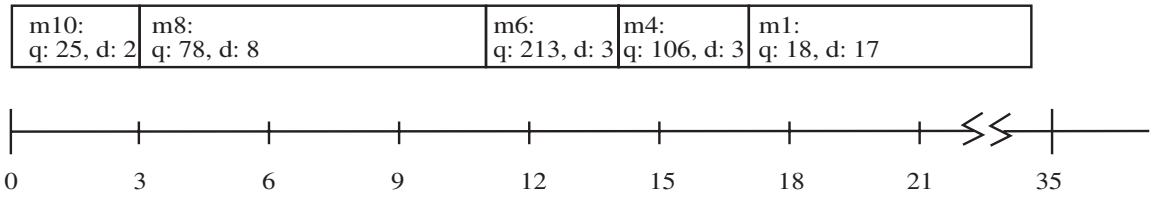

Figure 2: A schedule that solves the problem from Figure 1. The runtimes and qualities for $m6$ and $m4$ are not as indicated in Figure 1, because both methods are facilitated, thus reducing duration and increasing quality.

In the task structure shown here, it is assumed that the quality and duration of each method is known with certainty and the effect of interactions among tasks is completely predictable. Most systems-oriented work in real-time shares this assumption that the worst-case execution time of each method is known and is relatively close to the expected execution time. For most AI applications this is not a reasonable assumption because of the heuristic algorithms used. The performance of AI tasks is often difficult to predict and worst-case execution time (when known at all) is often much longer than expected (average case) execution time. For this reason it is useful to extend our problem representation to allow uncertainty in the quality and duration of methods, and in the effects of interactions. This of course introduces significant additional difficulties in the building of schedules and in their execution. In particular, monitoring of method execution becomes necessary. Along with an ordered list of methods to execute (with expected start and finish times) a schedule now also needs to contain what we call

monitoring points, scheduled times to interrupt method execution and check on progress, as well as specifications of what progress to require and what to do if those requirements are not expected to be met.

Design-to-time is related to *anytime algorithm*[Dean and Boddy, 1988, Russell and Zilberstein, 1991] and *imprecise computation*[Liu *et al.*, 1991] approaches to problem solving. In the anytime algorithm approach a procedure is available for each subproblem that achieves increasing value as it is given increasing runtime. In early anytime algorithm work by Dean and Boddy[Dean and Boddy, 1988, Boddy and Dean, 1989] each of these procedures was assumed to be independent and the function mapping value to time (known as the *performance profile* was assumed to be static. Dean and Boddy derived an algorithm for optimally allocating time to anytime algorithms under these assumptions[1]. More recent work in anytime algorithm scheduling has been done by Zilberstein and Russell[Russell and Zilberstein, 1991, Zilberstein and Russell, 1992]. They extended the paradigm to allow performance profiles to be dependent on inputs. The focus of their work has been on compilation, that is, finding time allocations for sets of anytime algorithms that are related by sharing inputs and outputs. They are able to find optimal solutions for problems where the graph of anytime algorithms connected by shared inputs/outputs is a directed acyclic graph, that is, for problems where each subproblem provides outputs for exactly one other problem. For problems with shared subproblems they describe heuristic algorithms that should generally perform quite well. Anytime algorithm work differs from design-to-time scheduling in the assumption that all methods are anytime algorithms with well behaved performance profiles and in the limitation of interactions to those possible through the direct sharing of inputs/outputs. More detail on anytime algorithms and other AI approaches to real-time are described in Garvey and Lesser[Garvey and Lesser, 1994]. Imprecise computation is a systems-oriented approach that relies on methods that have mandatory and optional parts. Solutions to problems involve finding schedules that execute all mandatory parts and get as much value as possible out of optional parts (usually just by spending as much time executing optional parts as possible). Optional parts can be either anytime algorithms (usually with linear performance profiles) or 0/1 algorithms that either must be executed completely or not at all. The only kind of interaction between methods that has been explored in imprecise computation is hard precedence constraints. Research in imprecise computation has focussed on finding polynomial time algorithms for well constrained subsets of the general problem[Liu *et al.*, 1991], or showing that particular problems are NP-Complete[Ho *et al.*, 1992, Leung *et al.*, 1992]. Imprecise computation differs from design-to-time in its limiting of interactions to precedence constraints, its interest in severely limiting problem representation to allow polynomial time solutions to be found, and its exclusive focus on optimal (rather than heuristic, satisficing) solutions.

In this paper we first describe the kinds of problems we are interested in solving (as represented in the TÆMS task modeling paradigm), then describe the basic mechanisms used by our heuristic scheduling algorithm. Next we describe extensions to that algorithm to allow it to schedule when there is uncertainty in the duration and quality of methods and the effect of interactions. Following that we describe experiments that explore the effect of uncertainty on the difficulty of scheduling and the quality of schedules produced. We conclude with a

[1] A more general and more efficient algorithm for solving this problem has since been designed by Dey, Kurose and Towsley[Dey *et al.*, 1993].

summary of the approach and a discussion of future research questions.

2 Problem Specification

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), maximum (OR-like), or sum). At the leaves of the DAG are *executable methods* M representing actual computations that can be performed. An 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, $\mathsf{facilitates}(T, M, \phi_d, \phi_q)$ 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 $\mathsf{hinders}(T, M, \phi_d, \phi_q)$ (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.

3 Heuristic Design-to-time Scheduling Algorithm

This section describes the basic algorithm used to schedule TÆMS task structures. The algorithm consists of three main components corresponding to the three subproblems of planning (deciding "what" tasks to do), scheduling (deciding "when" to do those tasks), and iterative repair (trying to improve schedules). In our work these components are initial alternative generation, method ordering (aka scheduling), and schedule analysis. The two basic data structures used by the algorithm are *alternatives* and *schedules*. An alternative is an annotated set of methods that, if they were all to achieve their expected quality, would achieve quality for their task. Alternatives are generated for each task and task group as described below. They are combined together to form alternatives for the entire set of task groups to be scheduled. A schedule is an ordered set of methods with expected start times, finish times and qualities. A graph showing the interconnections between the components and data structures is shown in Figure 3.

3.1 Initial Alternative Generation

Initial alternative generation involves analyzing a task structure and deciding on a set of alternatives for each task group. This initial analysis is done without reference to task relationships or deadlines, so it is possible that a given method set could not possibly achieve quality either

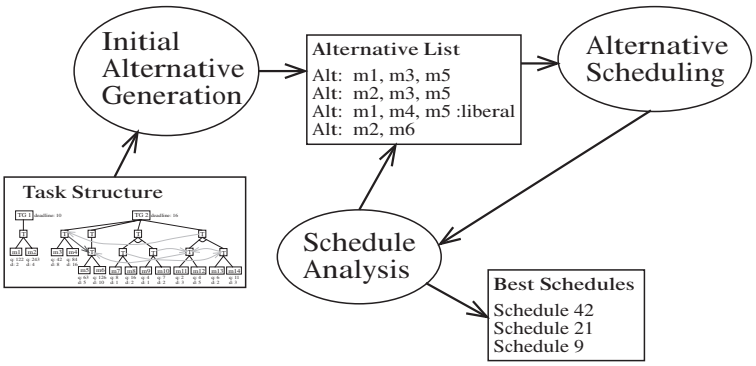

Figure 3: A chart showing the interconnections between components of the scheduling algorithm.

because of interacting relationships (e.g., some methods not being enabled by any methods in the method set) or deadlines. To help alleviate some of the uncertainty introduced by ignoring deadlines and relationships, a broad range of alternative generators are used. Different generators will be applicable in different problem solving circumstances (e.g., extreme time pressure, no time pressure, ...).

Alternatives are generated by recursively generating alternatives for each task in the task structure starting at the leaves. There is exactly one alternative for each method that consists of just that method. For tasks there is a small set of alternative generators that generate alternatives using the quality accumulation function for the task and the alternatives for each child task. At this point the complete set of initial alternative generators[2] (not all of which are applicable for all tasks) consists of:

- *highest quality* – This generates the highest (expected) quality alternative with the minimum (expected) duration. For the sum quality accumulation function this is just the highest expected quality alternative for each subtask combined together. For minimum quality accumulation we find the highest quality alternative for each subtask, take the one that has the lowest quality, and combine it with the fastest alternative for each other subtask that achieves at least that (lowest) quality. This is because when quality is combined using minimum it is not possible to achieve higher than this lowest quality, so there is no point in achieving higher quality for other subtasks. For maximum quality accumulation this is the alternative for the one subtask that has the highest expected quality with the highest value. For example, in the task structure in Figure 1 this the *highest quality* alternative for Tc is $\{m6\}$ and for $TG2$ it is $\{m4, m6, m8, m10\}$.

- *minimum duration* – This generates the minimum (expected) duration alternative that achieves nonzero quality. For minimum quality accumulation this is the combination of the minimum duration alternatives for each subtask. For maximum and sum quality

[2]Note that in these alternative generators there is a distinction made between *expected* quality/duration and *maximum possible* quality and *minimum possible* duration. Expected quality/duration is the quality/duration a method would achieve if no relationships had any effect on it. Maximum possible quality is the quality that a method would achieve if all positive relationships had their most positive effect and no negative relationships had any effect. Minimum possible duration is similar.

accumulation this is the alternative for the one subtask that has the minimum expected duration with the minimum value. From the example, the *minimum duration* alternative for Tc is $\{m5\}$ and for $TG2$ it is $\{m9\}$.

- *maximum possible quality* – This is identical to highest quality, except it uses maximum possible quality rather than expected quality.

- *quality/duration tradeoff* – This generates a fixed number of extra alternatives for tasks with sum and minimum quality accumulation. The idea is to allow the exploration of a few of the exponential number of possible combinations of subtasks for sum and minimum quality tasks. At this point a parameterized number (5 by default) of extra alternatives are generated that combine alternatives with the highest quality to duration ratios. In some cases these alternatives are either identical to existing alternatives or achieve lower or equal quality to existing alternatives in higher or equal duration. These inferior alternatives are pruned. From the example, a *quality/duration tradeoff* alternative for $TG2$ is $\{m3, m5, m7, m10\}$ which has an expected quality of about half the *highest quality* alternative, and a duration that is about 50% less. In tight deadline situations this alternative might be preferable to the longer duration alternative.

- *nonlocal* – This and the next generator are only useful in multiagent situations. This generator looks to see if a nonlocal result is expected for the task (i.e., if another agent has committed to transmitting the result for this task by a particular time). If such a result is expected, then an alternative that relies on that result is generated with the expected quality of the result and a duration of zero.

- *minimize nonlocal* – This generator is similar to highest quality, but tries to find the highest quality alternative with minimum nonlocal result usage, rather than minimum duration. Associated with each alternative is a list of nonlocal results that it is relying on to achieve its expected quality. When alternatives are combined to generate new alternatives, their nonlocal results are combined as well. This generator tries to find an alternative that has the shortest list of nonlocal results.

When two different alternative generators generate identical alternatives (consisting of the same method set) those alternatives are combined.

The result of this process is a set of alternatives for each task group. The next step is to combine these task group alternatives together to form complete alternatives for the entire task structure. Since the order in which this combination occurs does not matter (although the order of methods in schedules certainly does matter) the number of such combinations is simply the number of alternatives for each task group multiplied together. The number of initial alternatives for a task group is usually between 5 and 10, although sometimes the number is lower because of the combining of identical alternatives. Thus the number of possible combinations of alternatives is approximately $O(a^n)$ where a is the semiconstant number of alternatives for each task group and n is the number of task groups. Clearly for all but a small number of task groups this number is too large to consider. At this point all such possible combinations of alternatives are actually generated, but a better approach would be to heuristically generate a parameterized number of them. In the example, 2 alternatives are generated for $TG1$ and 3 alternatives are generated for $TG2$, leading to 6 initial alternatives.

3.2 Scheduling of Alternatives

Once an initial set of alternatives for the entire set of task groups has been generated, we begin to take these alternatives and schedule the methods they contain. The first step in the scheduling of these alternatives is choosing which alternative to attempt to schedule. A simple heuristic technique is used to make this choice. Initially it chooses the fastest acceptable alternatives, with a goal of getting a minimal acceptable schedule. If the schedule analysis module has suggested alternatives with conservative or liberal biases (as explained below) these are given preference. Then it begins looking at the alternatives with the highest expected quality value. These heuristics for ordering the alternatives contribute to the anytime character of the scheduling algorithm by finding a minimal acceptable schedule as soon as possible and finding better schedules as more alternatives are considered. When an alternative is chosen, the scheduler first checks to see if the schedule to be produced by this alternative could possibly be better than the best schedule we already have (by comparing the maximum quality the schedule could possibly produce to the expected quality of the best existing schedule). If this best possible quality is less than our best existing quality, then scheduling is terminated (since the chosen alternative is the best still unscheduled). Scheduling is also terminated when a threshold number of alternatives have been considered. Otherwise the best alternative is returned.

Once an alternative has been chosen, it is scheduled in a simple, nonbacktracking, heuristic manner. The basic procedure is extremely simple.

1. Start with an empty schedule that does nothing.

2. Rate each method in the chosen alternative against a set of heuristics (taking the current schedule into account.)

3. Add the method with the highest nonnegative rating to the end of the schedule. If all methods have been scheduled, then return the current schedule, else if all ratings are negative, then consider adding idle time to the end of the schedule (followed by a repeat of step 2 or return), else a repeat of step 2.

A negative rating for a method means that it is inappropriate to add the method to the end of the current schedule, probably because it is not enabled or would violate a deadline or earliest start time constraint. A zero rating means it would probably be better not to add the method, but it is not an error to do so. A positive rating assesses the method's utility relative to other positively rated method's utilities.

The standard set of heuristics is:

- *enforce enables* – Gives a negative rating to unenabled methods and a 0 to all other methods.

- *enforce hard deadline* – Gives a negative rating to any method that would exceed its deadline if the method was added to the end of the schedule and a 0 to all other methods.

- *enforce earliest start time* – Gives a negative rating to any method that would violate an earliest start time constraint if the method was added to the end of the schedule and a 0 to all other methods.

- *prefer facilitators* – Increases the rating of all methods that allow other methods to transition from unfacilitated to facilitated if the facilitating method is added to the end of the schedule. The increase in value is proportional to the number of methods that are newly facilitated. Does not change the current rating of nonfacilitating methods.

- *delay facilitatees* – Gives a 0 rating to all methods that could be facilitated by tasks that have not yet achieved nonzero quality (meaning that the facilitation would not take place if the method was added to the end of the schedule). Allows the current rating of all other methods to be unchanged.

- *prefer increased quality* – Increases the rating of all methods that would directly increase the quality of their task group if they were to be added to the end of the schedule. The rating increase is proportional to the increase in quality for the task group. Allows the current rating to be unchanged otherwise.

- *avoid violating commitments* – Reduces the rating of methods that would cause commitments to be newly violated. The reduction in rating is proportional to the penalty associated with violating the commitments. Allows the current rating to be unchanged otherwise.

- *prefer satisfying commitments* – Increases the rating of methods that cause commitments to be newly satisfied. The increase in rating is proportional to the value of satisfying the commitment. The rating is unchanged otherwise.

- *prefer earlier deadlines* – Increases the rating of methods proportional to the earliness of their deadlines. The increase is ratings is substantial and causes all positively rated methods to be rated in an earliest deadline first manner with all other ratings breaking ties among methods with the same deadline.

These heuristics are evaluated in the order given and as soon as any heuristic assigns a negative rating, the process is aborted and the negative rating is returned.

Besides the standard heuristic set given above there are two minor modifications that bias the scheduler to be more conservative or more liberal. An alternative can have a suggested scheduling bias associated with it that indicates the preference of the originator of the alternative. The suggestion of such nonstandard biases is done by the Schedule Analysis component as described below. The intent of these biases is to change the way the scheduler orders methods in over constrained situations (where there is not enough time available to schedule all desired methods) or less constrained situations (where extra time is available to improve quality). In over-constrained situations it might be better to jettison work on task groups that we have not been able to generate quality for. In less constrained situations it might be useful to dampen some of the best-first character of the heuristics to allow longer duration combinations of methods to be considered.

The liberal scheduling heuristics are identical to the standard, except that *delay facilitatees* is replaced by *help circular facilitates* which looks for cases where two methods that both cause the facilitation of another method are schedulable. Normally, the one that generates higher quality would be scheduled (because of the ratings of other heuristics). This heuristic gives a higher rating to the lower quality method in the pair, thus allowing the facilitation to occur, and still

allowing the other (higher quality) member of the pair to be facilitated later thus achieving higher overall quality. The *prefer increased quality* heuristic is also removed from the liberal set, under the assumption that enough time is available to schedule all methods, so scheduling highest quality first has no useful purpose and might interfere with other ordering heuristics.

The conservative scheduling heuristics are also identical to the standard, except the additional heuristic *avoid working on hopeless task groups* is added to the very end. This heuristic assigns a rating of 0 to methods whose task group is on a list associated with the alternative. This is a list of task groups that the originator of the conservative bias for the alternative believes cannot possibly achieve nonzero quality. This heuristic has the effect of not scheduling work on methods from task groups on that list unless no other methods are executable. The intention is to allow more time for other task groups, which might allow them to achieve higher quality than they would otherwise achieve.

For example, in the example task structure from Figure 1, one alternative for the set of both task groups is $\{m1, m4, m6, m8, m10\}$, combining the fastest alternative for $TG1$ with the highest quality alternative for $TG2$. When this alternative is scheduled, the *prefer increased quality* heuristic dominates resulting in a schedule that orders the methods $\{m6, m4, m8, m10, m1\}$. This schedule generates a total quality of 358.5 (18 for $TG1$ and 340.5 for $TG2$), but does not take advantage of the facilitates from Td to Tc and from Te to Tb. The reason the heuristics initially rate the methods this way is because this ordering returns the most quality quickly. The heuristics are greedy because no backtracking is done when schedules are built, so it is always preferred to get as much quality as possible as quickly as possible. However, in this case the liberal scheduling improvement (described below) will notice that all methods were successfully scheduled, but one or more methods did not achieve their maximum quality (in this case methods $m4$ and $m6$.) When the methods in the alternative are rated using the liberal scheduling heuristics, the ordering is $\{m10, m8, m6, m4, m1\}$ (as shown in Figure 2) and the resulting total quality is 438.8 (18 for $TG1$ and 420.8 for $TG2$).

3.3 Schedule Analysis

Each schedule that is returned by the Alternative Scheduling component undergoes some level of analysis by this component. First, the schedule is evaluated against all evaluation functions to determine if it is better than the best current schedule against any of them. If it is, then it is pushed on the list of good schedules to be remembered. The current evaluation function prefers schedules that achieve the highest quality. From the set of schedules that achieve the highest quality it prefers the schedules that have the shortest duration. An extra reward is given to schedules that achieve nonzero quality for all task groups. In the work on uncertainty mentioned below, a consideration of the probability that the schedule will achieve its expected quality is also done.

Following this basic evaluation, a series of local improvements are attempted on the schedule. These improvements are a form of iterative repair[Zweben *et al.*, 1993], except that each successful improvement adds new alternatives for the entire set of task groups to those being considered (i.e., new plans), rather than actual complete schedules. There are parameters that control both how many total alternatives can be generated and how many alternatives can be generated as possible ways to improve any particular schedule. In practice no more than about 4 improvements can be suggested for any one schedule given limitations of the current set of

local improvements, but this number will increase as new, more specialized improvements are added.

The first set of improvements check to see if the schedule could potentially benefit from a scheduling bias. These biases adjust the set of heuristics used to order the methods in a schedule. The intent is consider different ways of ordering the methods, either because methods achieved zero quality or missed deadlines (calling for a conservative bias) or all methods were scheduled but some achieved less than their maximum possible quality (calling for a liberal bias).

The conservative bias is biased against particular task groups (indicated as part of indicating the conservative bias.) Methods associated with these task groups will not be scheduled unless nothing else can be scheduled. The intent is to allow consideration of methods that might be able to achieve quality but haven't because time is being wasted on task groups that do not end up achieving quality. The *consider conservative bias* improvement notices if the schedule was unbiased or had a conservative bias but still had zero quality for some task group not in the conservative bias list for the alternative, and if so generates an alternative that has the same methods as the schedule's alternative, and a conservative bias with a list of task groups that includes the previous schedule's list (if any), plus the earliest deadline zero quality task group (not already in the list) in the schedule.

The liberal bias is biased in favor of taking chances that might achieve higher quality, but might achieve less or no quality if they fail. Normally the scheduling heuristics are greedy and try to achieve quality as quickly as possible, even if it is not the best possible quality. The liberal bias changes this to favor orderings that might achieve higher quality, but might not complete by a deadline. The *consider liberal bias* improvement notices if the schedule was unbiased, all methods in the alternative were scheduled and at least one task group has less than its maximum possible quality, and if so generates an alternative that has the same methods as the schedule's alternative, and a liberal bias.

The next set of improvements try to add additional methods to the schedule's alternative to allow unenabled or unfacilitated methods to be enabled or facilitated. The *unenabled method* improvement looks for methods in schedule that were not enabled, and combines an alternative for the task that enables the unenabled method (from the alternatives originally generated for that task) with the schedule's alternative. The *nonmax quality* improvement looks for methods in the schedule that are not achieving their maximum possible quality (because of facilitates that are not taken advantage of) and combines an alternative for the facilitators of these methods (again from the original alternatives generated for those facilitating tasks) with the schedule's alternative. If the facilitators already achieve nonzero quality, then it is probably a circular facilitates problem of some sort, so a different alternative (different than the one already included in the schedule alternative) for the facilitator is added to the schedule. The *nonmax quality circular* improvement is very similar. It combines another alternative for methods that do not achieve their maximum quality with the schedule's alternative. This also is meant to help with circular facilitates in the case where the nonmax quality method has to be executed too early to be facilitated (presumably because it enables or facilitates something else) and allows another alternative to be scheduled. Both of these last two improvements work best with liberal scheduling biases, but they are first tried with the standard bias to ensure that no obvious schedules are missed. They will be considered for a liberal bias after the standard version is actually scheduled.

Once all of the improvements have been considered, control passes back to the Alternative

Scheduling component. This component can either decide to schedule another alternative or stop scheduling.

3.4 Anytime Behavior

Because of the heuristic approach that is used to choose which alternatives to schedule, a minimal value schedule is usually found quickly and the schedules produced tend to improve in value as more alternatives are scheduled. This gives the scheduling algorithm an anytime algorithm character.

Figure 4 shows the sum quality of the best schedule found (so far) as more alternatives are considered. In this case, the first alternative considered for the set of task groups is the alternative that combines the *minimum duration* alternative for each task group (consisting of $\{m1, m9\}$). The schedule generated from this alternative achieves a sum quality of 30. This is followed by a liberal-biased version of this alternative (because $m9$ does not achieve its maximum quality) that generates the same schedule. The next alternative considered is the combination of the *highest quality* alternatives for each task group (consisting of $\{m2, m4, m6, m8, m10\}$) and the resulting schedule achieves a sum quality of 298, but 0 quality for $TG1$ (because it is unable to complete execution of $m2$ before its deadline of 40). This is followed by several improvements that try to increase the qualities for $m4$ and $m6$, as well as achieve nonzero quality for $m2$. These improvements are unable to improve the sum quality. Next the alternative that combines the *minimum duration* alternative for $TG1$ with the *highest quality* alternative for $TG2$ is considered. As described at the end of Section 3.2, this initially results in an ordering that achieves a sum quality of 358.5. Rescheduling with a liberal bias improves this sum quality to 430.8. This is followed by 10 more improvements that try to achieve higher quality for $TG1$, but are unsuccessful. At this point scheduling terminates, because no alternative is left that could possibly achieve a higher sum quality. Alternatives not considered include, for example, the *highest quality* for $TG1$ combined with *minimum duration* for $TG2$, because it could not achieve a sum quality of more than 69 (57 for $m2$ plus 12 for $m9$).

This algorithm has a few parameters that can externally control the behavior of the algorithm. Judicious use of these parameters can result in anytime performance by the algorithm. Unfortunately it would probably be quite difficult to calculate an *a priori* performance profile for the scheduling of a particular task structure. It should (in principle) be possible to predict the bounds on runtime for the algorithm given a particular task structure, but it is very difficult (probably impossible) to predict the expected quality and duration of the resulting schedule without actually doing a minimum amount of actual scheduling.

4 Extensions for Uncertainty

Unfortunately, in many real applications the precise quality and duration values used in the algorithm described above might not be available, either because it is too expensive or difficult to calculate them or because of inherent uncertainty in the problem domain. In those situations, it is desirable to extend the problem solving model to include uncertainty. This has the effect of significantly increasing the difficulty of the scheduling problem, but schedulers that take uncertainty into account are likely to create schedules that achieve much better results in

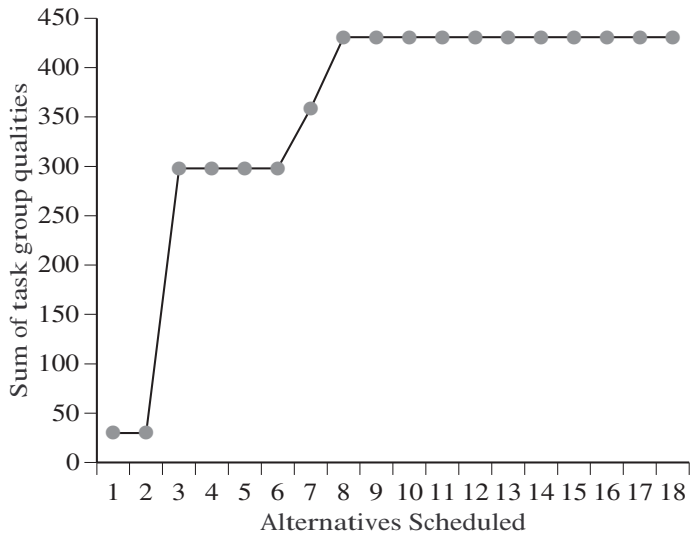

Figure 4: The sum quality of the best schedule as a function of the number of alternatives considered for the task structure from Figure 1.

uncertain situations both because they can take the likelihood of achieving results into account and because rescheduling overhead can be reduced (as described below) by doing contingency scheduling.

Uncertainty is introduced to the system as uncertain duration and qualities for methods and uncertain parameter values for relationships. In this work the form of these uncertainties is discrete distributions (e.g., there is a 20% chance the value is 12, a 50% chance the value is 13, and a 30% chance the value is 14). Each method has independent discrete distributions for its quality and duration. Each relationship has a discrete distribution for its parameters (e.g., the duration and quality power parameters for facilitates relationships).

In extending the heuristic scheduler described above to work with uncertain information two major changes are required: modifying the scheduler to maintain and propagate uncertain information as schedules are constructed, and annotating schedules with monitoring information and using that information at runtime to monitor method execution.

When constructing a schedule it is necessary to propagate uncertainty information through the schedule. For example, the start time distribution for a method in a schedule is the finish time distribution of the previous method in the schedule, and the finish time distribution for a method is its start time distribution plus its duration distribution. For those times in the finish time distribution that are later than the deadline for the method, the quality distribution has to be updated to be 0.

When a method is added to a schedule, the scheduler looks for a lower quality, but faster method that could replace the added method. If it can find such an alternate, the schedule is annotated to remember the alternate and switching to the alternate is one of the options considered when monitoring is done. (In the future, we would like to explore more complex versions of this contingency scheduling that might have alternates for larger sections of a schedule or entire contingent schedules that continue on from a monitoring point.)

Once a schedule has been constructed, it is evaluated (and possibly improved) as before. As

mentioned above, one of the factors taken into account in this evaluation is the likelihood that this schedule will produce the quality that is expected. The importance of this likelihood could be varied, depending on the tolerance for a schedule occasionally not completing as expected and on the amount of monitoring.

When a schedule is executed, as each method begins execution *monitoring points* are assigned to it. These are times at which execution will be stopped and progress on the method execution evaluated. Associated with each monitoring point are minimum quality and maximum duration values. If these expectations are not being met, the monitor has the option of reinvoking the scheduler or switching to an alternate method (if one is available). In our current (somewhat simple) implementation, monitoring is always done at the latest point that the alternate could still begin execution (to ensure that we allow the option of switching to the alternate), and periodically (with an experimentally controlled period) if no alternate is available.

5 Experiments

The intent of our experiments is to understand the effects of uncertainty on the quality of the results produced by our scheduling algorithm. In the first experiment we compare our approach of using expected values from the quality and duration distributions with the more traditional systems-oriented approach of using worst-case values. As the scheduler is propagating quality and duration distributions during schedule construction, it is often useful to distill the distribution down to a single number. Normally the scheduler uses the median value (50th percentile) of the distribution. In this experiment we modified the scheduler to use a time percentile for time-related distributions (e.g., duration) and a quality percentile for quality-related distributions. When these values are set to 1 for time and 0 for quality the scheduler will make worst-case assumptions about the expected quality and duration of the distribution in question.

We ran the scheduler on 100 randomly generated problem episodes (generated with environmental parameters that result in task structures with at least the size and complexity of the one in Figure 1) with each of the two possible values for the two percentiles that are varied. Because the variance between these randomly generated episodes is so great, we took advantage of the paired response nature of the data to run a non-parametric Wilcoxon matched-pairs signed-ranks test [Daniel, 1978] to compare the worst-case values (0 for quality, 1 for time) with the median values (0.5 for each). The null hypothesis is that there is no difference in the sum of quality for all task groups in each pair of runs. We were able to disprove this hypothesis ($p = 0.003$) and, on average, the runs using the worst-case values achieved 3.6 percent less quality. We were not able to disprove the null hypothesis that there is no difference between the number of deadlines missed using the two different values (meaning no significant difference was detected between the number of deadlines missed in the two runs.)

We next extended this experiment to look at the effect of deadline tightness on the quality of schedules produced. As above, we compared using worst-case quality and duration values to using median values. It might be imagined that as deadlines become tighter, the usefulness of using worst-case values increases. Our results suggest this is not true, in fact as deadlines become tighter, the value of using median values increases. Again we used a non-parametric Wilcoxon matched-pairs signed-ranks test to compare worst-case values with median values for

deadlines that were 0.5 and 0.75 times the original deadlines. The null hypothesis is that there is no difference in the sum quality for all task groups in each pair of runs. We were able to disprove this hypothesis ($p = 0.00001$ for 0.5 times deadline and $p = 0.0001$ for 0.75 times deadline) and, on average, the runs using the worst-case values achieved 4.7% and 5.8% less quality respectively. Again no significant difference was detected in the number of deadlines missed.

These results suggest that there is potential benefit from taking uncertainty into account when scheduling, rather than just using worst-case values, and that this benefit can be gained without missing any more deadlines than are missed when worst-case assumptions are made.

6 Future Work

There are several directions in which this research can be extended. Some of these directions include extending the kinds of task structures that can be scheduled (such as by adding the ability to schedule methods that are anytime algorithms, increasing the set of relationships that are handled, and adding the ability to schedule tasks that require non-CPU resources), improving the heuristic performance of the scheduling algorithm (such as by adding more schedule improvement modules and making the scheduler more aware of its own use of resources), and testing the ideas in a real application. Work in each of these areas is planned as part of the continued development of the design-to-time scheduling approach.

References

[Boddy and Dean, 1989] Mark Boddy and Thomas Dean. Solving time-dependent planning problems. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Detroit, MI, August 1989.

[Daniel, 1978] W. W. Daniel. *Applied Nonparametric Statistics*. Houghton-Mifflin, Boston, 1978.

[Dean and Boddy, 1988] T. Dean and M. Boddy. An analysis of time-dependent planning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 49–54, St. Paul, Minnesota, August 1988.

[Decker and Lesser, 1992] Keith S. Decker and Victor R. Lesser. Generalizing the partial global planning algorithm. *International Journal of Intelligent and Cooperative Information Systems*, 1(2):319–346, June 1992.

[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.

[Dey *et al.*, 1993] J.K. Dey, James Kurose, and Don Towsley. On-line processor scheduling for a class of IRIS (increasing reward with increasing time) real-time tasks. CS Technical Report 93–09, University of Massachusetts, 1993.

[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 and Lesser, 1994] Alan Garvey and Victor Lesser. A survey of research in deliberative real-time artificial intelligence. *The Journal of Real-Time Systems*, 6, 1994.

[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.

[Garvey *et al.*, 1994] Alan Garvey, Keith Decker, and Victor Lesser. A negotiation-based interface between a real-time scheduler and a decision-maker. CS Technical Report 94–08, University of Massachusetts, 1994.

[Ho *et al.*, 1992] Kevin I-J. Ho, Joseph Y-T. Leung, and W-D. Wei. Scheduling imprecise computation tasks with 0/1-constraint. Technical Report UNL–CSE–92–16, University of Nebraska-Lincoln, 1992.

[Leung *et al.*, 1992] Joseph Y-T. Leung, Vincent K.M. Yu, and W-D. Wei. Minimizing the weighted number of tardy task units. Technical report, University of Nebraska-Lincoln, 1992.

[Liu *et al.*, 1991] 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*, 24(5):58–68, May 1991.

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

[Smith, 1993] Stephen F. Smith. Integrating planning and scheduling: Towards effective coordination in complex, resource-constrained domains. In *Keynote Address at the Italian Planning Workshop*, Rome, Italy, September 1993.

[Zilberstein and Russell, 1992] Shlomo Zilberstein and Stuart J. Russell. Efficient resource-bounded reasoning in AT-RALPH. In *Proceedings of the First International Conference on AI Planning Systems*, College Park, Maryland, June 1992.

[Zweben *et al.*, 1993] Monte Zweben, Eugene Davis, Brian Daun, and Michael J. Deale. Scheduling and rescheduling with iterative repair. *IEEE Transactions on Systems, Man and Cybernetics*, 23(6):1588–1596, 1993.