

Reasoning about Executorial Uncertainty to Strengthen Schedules

Laura M. Hiatt[†] and Terry L. Zimmerman[‡] and Stephen F. Smith[‡] and Reid Simmons^{†‡}

[†] Computer Science Department

[‡] Robotics Institute

Carnegie Mellon University

Pittsburgh, PA 15213

Abstract

We consider a scheduling problem where the goal is to maximize the reward obtained by a team of agents in an execution environment with duration and activity outcome uncertainty. To address scalability issues, we take as our starting point a deterministic, partial-order schedule and attempt to hedge against activity failure by adding redundant backup activities into the agent schedules.

There is a basic trade-off in adding such backup activities, since they reduce overall temporal slack and can increase the risk that other activities will fail. We approach this trade-off by explicitly analyzing the executorial uncertainty of the deterministic schedule to identify those activities most likely to fail to contribute to scheduled reward, and target the introduction of additional backup activities accordingly. The analysis also determines when the schedule has become saturated and further activity additions would be counterproductive. Experiments in simulation demonstrate that schedules that are strengthened in this manner earn higher reward than both the original schedules and those produced by less focused mechanisms.

Introduction

The general problem of scheduling under uncertainty remains a difficult challenge. Research that has opted to build schedules (or scheduling policies) by reasoning with explicit models of uncertainty has developed some promising mechanisms for coping with specific types of domain uncertainties (e.g., (McKay et al. 2000; Beck and Wilson 2007)), but in general has produced solutions that have difficulty scaling (Bryce, Mausam, and Yoon 2008). Other research in robust scheduling has alternatively emphasized the use of expected models and deterministic scheduling techniques, but with the goal of constructing a flexible schedule (or a set of schedules) that can absorb deviations at execution time (Drummond, Bresina, and Swanson 1994; Policella et al. 2006). These approaches are much more scalable, but either result in overly conservative schedules at the expense of performance (e.g., (Morris, Muscettola, and Vidal 2001)) or ignore the potential leverage that can be provided by an explicit uncertainty model.

In this paper, we adopt a composite approach that attempts to couple the strengths of both of these research streams. Like (Policella et al. 2006), we assume as our starting point a partial-order schedule, which is constructed (based on deterministic modeling assumptions) to optimize the performance objective at hand, but retains temporal flexibility where permitted by domain constraints as a basic hedge against uncertainty. We then perform an explicit analysis of the executorial uncertainty of this baseline schedule to identify its weak points and take appropriate schedule fortification actions (e.g. adding redundant backup actions that can achieve the same results as the potentially failing action).

The general problem that motivates our work is that of coordinating the activities of a team of collaborative agents as they execute a joint schedule in an uncertain environment. In the class of problems we focus on, various activities in the joint schedule contribute differentially to the reward (referred to in this paper as quality) that is achieved by executing the schedule, and the overall goal is to maximize accumulated quality. A given agent can execute one activity at a time, an activity succeeds and accumulates its designated quality if it is executed within its allowable time window, and the execution of a given activity may enable (or disable) other activities by the same or other agents. Activities are organized hierarchically, and lower quality backup activities are often available as alternatives if time is short. As execution proceeds, all activities are subject to both durational and outcome uncertainty (characterized as discrete probability distributions), making the challenge one of constructing a robust, quality-maximizing schedule.

One way to increase schedule robustness in this context is to protect against activity failure by adding redundant, backup activities. However, there is a basic trade-off in trying to fortify the schedule in this way: the addition of new activities will reduce overall temporal slack in the agents' schedules, and this in turn can increase the risk that other activities will fail to execute within their specified time windows.

We argue that reasoning about the uncertainty associated with scheduled activities provides a basis for effectively reconciling this trade-off. A probability of failure analysis is developed for targeting the most profitable activities to bol-

ster, and an analysis of overall expected quality is defined to determine when the schedule has become saturated and further introduction of backup methods would be counter-productive. We present a schedule execution experiment, performed in simulation, which demonstrates that schedules strengthened in this manner achieve higher quality than both the original schedules and those produced by less focused mechanisms.

The remainder of the paper is organized as follows. We first briefly review other related work in scheduling under uncertainty. Next we summarize the multi-agent scheduling problem of interest and the assumptions that underlie generation of the initial deterministic schedule for this problem domain. The core mechanisms for strengthening this initial schedule are then presented, followed by our experimental analysis. We end with a discussion of our broader research objectives in this area.

Related Work

Robust Scheduling

One way to increase schedule robustness is to use a partial-order schedule representation, which flexibly allows activity execution intervals to drift within a range of feasible start and end times. Policella et al. (2006) generate robust partial-order plans by expanding a fixed-time schedule to be partial-order, showing improvements with respect to fewer precedence orderings and more temporal slack over a more top-down planning approach.

Dynamic controllability specifies whether a solution strategy exists for an uncertain temporal planning problem; run-time decisions such as assigning a time to an executable timepoint can be made based only on observations available at the current point of execution. Morris, Muscettola, and Vidal (2001) have shown that determining dynamic controllability of simple temporal networks with uncertainty (STNUs) can be done in time polynomial to the size of the STNU; Rossi, Venable, and Yorke-Smith (2006) showed that this property is also true for STNUs that express preferences. In contrast with these approaches, however, the problem we consider solicits a solution that will also maximize quality earned during execution.

Probabilistic Planning

An alternate approach to improving schedule robustness is explicitly taking probabilistic information into account while generating the schedule. Conditional planners create branching plans, and assume observations will allow the executor to choose which branch of the plan to follow (Bertoli et al. 2001; Blum and Langford 1999; Little and Thiébaux 2006). Conformant planners such as (Smith and Weld 1998;

Onder, Whelan, and Li 2006) deal with uncertainty by developing plan that are robust without assuming that the results of actions are observable, and in some ways are similar to our approach. These types of planning are very difficult, however, and none of the above planners handle durative actions. Contingency planning such as (Younes and Simmons 2004a) is more scalable, but is typically best applied only for critical situations. Techniques have also been developed for scheduling with durational uncertainty (Beck and Wilson 2007) and unreliable resources (McKay et al. 2000) in the specific context of job shop scheduling.

Recent extensions to the MDP model have allowed MDP planners to take time into account in a more explicit way, and handle concurrency (Younes and Simmons 2004b; Mausam and Weld 2006). Other MDP work (Musliner et al. 2007) has attempted to solve an uncertain, distributed, multi-agent domain in real time. While both these approaches in general develop very good solutions, their computational complexity prevents them from scaling and performing well as run-time algorithms.

Problem of Interest

We focus on a multi-agent schedule management problem that is concerned with collaborative execution of a joint mission by a team of agents in a highly dynamic environment. Missions are formulated as a network of activities in a version of the *TAEMS* language (Task Analysis, Environment Modeling and Simulation) (Decker 1996) called *C-TAEMS* (Boddy et al. 2005). A simple example of such a hierarchical activity network is shown in Figure 1. The root of the tree is the overall mission task, called the “taskgroup” (TG), for which agents seek to accrue as much quality as possible. On successive levels, interior nodes constitute aggregate activities that can be decomposed into sets of sub-activities and/or primitive activities (leaf nodes), which are directly executable in the world. Hereafter we refer to interior nodes as “tasks” and leaf nodes as “methods”; the term “activity” will be used to refer to either type of node.

Each method can be executed only by a specified agent (denoted by *agent: Name* in Figure 1) and each agent can execute at most one activity at a time. Method durations are specified as discrete probability distributions, and hence known with certainty only after they have been executed. The notation in Figure 1 for duration, for example, *dur: ((5 0.5) (6 0.5))*, states that the method will have duration 5 with a 50% probability and duration 6 with a 50% probability. Method qualities, shown as *qual: X*, are deterministic. Methods may also have a specified failure distribution and, if so, have an associated likelihood of producing zero quality when executed (denoted by *fail: X%* in Figure 1).

Different types of tasks in an activity network accrue quality in different ways. Each task has a specified *quality accumulation function (qaf)* defining when and how a task accumu-

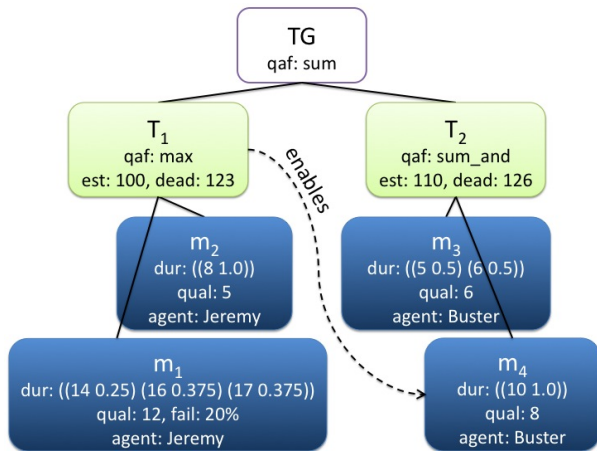


Figure 1: A simple 2-agent C_TAEMS problem.

lates quality as its children are completed. For example, a *sum* task accrues quality as soon as one child executes with positive quality, and its final value is the total earned quality of all children. Among the other qaf functions supported are *max*, which earns the maximum quality of its executed children, and *sum_and*, which earns the sum of the qualities of its executed children as long as *all* children successfully execute. Both *max* and *sum* qafs are considered *or* qafs, in that only one child has to successfully execute for the task to accrue quality. The *sum_and* qaf is an *and* qaf because the task only accrues quality when all of its children do.

While not required, a naturally emergent structural feature of these task networks is to have *max* tasks as the direct parents of the leaf node methods. These methods represent alternatives for achieving the parent *max* task, are typically of different durations and qualities, and often belong to different agents. For example, a task network for a search and rescue domain might model “Transport Medical Supplies” as a lowest-level *max* task with executable child methods for each agent that could transport the supplies. If a less comprehensive transport was an option, such as shuttling fewer supplies from a closer hospital, then additional methods would be present having lower quality and shorter duration. Hereafter we refer to *max* tasks that are parents of methods as “max-leaves.” They play a key role in the schedule strengthening strategy described below.

A rich set of inter-dependencies between activities in the problem can be modeled via non-local effects (NLEs), including “hard” and “soft” constraints. Hard NLEs express causal preconditions: for example, the *enables* NLE in Figure 1 stipulates that the target activity m_4 cannot be executed until the source activity T_1 accumulates quality. The NLE *disables* has the opposite effect; once the source activity accrues quality the target will not accrue positive quality if it is executed. Soft constraints include *facilitates* and *hinders*. The source activity for soft constraints does not have to execute for the target to accrue quality; however, if the source

earns quality before the target begins to be executed, it amplifies (or dampens) the quality and duration distribution of the target activity (see (Boddy et al. 2005)).

Additional constraints in the network include an earliest start time and a deadline (denoted by *est: X* and *dead: X* respectively in Figure 1), that can be specified for any node. Each descendant of a task inherits these constraints from its ancestors, and its effective execution window is defined by the tightest of these constraints. An executed activity earns zero quality if any of these constraints are violated.

Approach

Core Scheduler

Our scheduling approach is rooted in an incremental flexible-times scheduling framework (Smith et al. 2007). We adopt a partial-order schedule (*POS*) representation where, as previously indicated, the execution intervals associated with scheduled activities are not fixed, but instead are allowed to float within imposed time and activity sequencing constraints. This allows the explicit use of slack as a hedge against simple forms of executional uncertainty (e.g., method durations). Its underlying implementation as a Simple Temporal Network (STN) model provides efficient updating and consistency enforcement mechanisms.

To produce an initial deterministic schedule, we compute and assume expected duration values for methods in the input problem activity network and focus on producing a *POS* for the set of agents that maximizes overall problem quality. Given the complexity of solving this problem optimally and our interest in scalability, we adopt a heuristic approach. Specifically, a schedule is developed via an iterative two-phase process. In the first phase a relaxed version of the problem is solved optimally to determine the set of “contributor” methods, or methods that would maximize overall quality if there were no capacity constraints. In the second phase, an attempt is made to sequentially allocate these methods to activity timelines. If a contributor being allocated depends on one or more enabling activities that are not already scheduled, these are scheduled first. Should a method fail to be inserted at any point (due to capacity constraints), the first step is re-invoked with the problematic method excluded. More detail on this can be found in previous publications by the authors.

It should be noted here that while this deterministic *POS* schedule provides a degree of resiliency to uncertainties in the duration of activity execution, the core scheduler does not explicitly reason about either the impact of these uncertainties or failure likelihood. The next section describes an approach that post-processes agent schedules to partially ameliorate this shortcoming.

Strengthening by Backfilling

Given a set of agent schedules (as represented implicitly by the execution intervals associated with methods in the underlying STN), our interest is in fortifying the schedules to boost the likelihood that the projected taskgroup quality will in fact be achieved in actual execution. Possible tactics to achieve this include:

- substituting methods with lower failure likelihoods for scheduled methods with equivalent roles (generally siblings in the task network)
- scheduling additional methods under the source side of an enables NLE that supports a high-quality target task to reduce the odds that the entire enabling structure earns no quality because one child fails to accrue quality
- scheduling redundant methods under scheduled max-leaf tasks to improve their chances of accruing quality (recall that *max* tasks accrues only the quality of its highest quality child)

We limit our investigation here to the last strengthening tactic, which we term “backfilling,” in part because the deterministic scheduler tends to generate solutions for which most max-leaf nodes have a single (ideally maximum quality, but not always) child scheduled under them, regardless of failure likelihood. These backfill opportunities are readily identifiable and provide a sufficient number of instances to support exploration of the cost and tradeoffs of the approach. Our approach extends naturally to the other cases, which will be discussed further in the future work section of this paper.

Beyond the modest additional computation time, the primary trade-off in adding methods to bolster schedule robustness is the additional crowding of agent timelines, which may increase the risk that other methods will miss their deadline and earn no quality. To explore this trade-off we constructed a backfilling algorithm, *targeted backfilling*, which performs backfilling on initial schedules generated by the deterministic scheduler. The targeted backfilling algorithm uses the probabilistic analysis described below together with an assessment of the change in the problem’s expected quality to focus the backfilling effort.

Probabilistic Analysis of Deterministic Schedules

To estimate the expected quality of a schedule and identify those nodes in most need of redundant allocation, we developed an algorithm that Probabilistically Analyzes Deterministic Schedules generated by the above scheduler, (PADS). At a high level, PADS begins by finding the probability that methods will accrue quality based on missing a deadline or failing. It also tracks the finish time distribution of activities in order to find the finish time distributions (and accordingly the probabilities of missing a deadline) of later methods. It combines method probabilities and finish

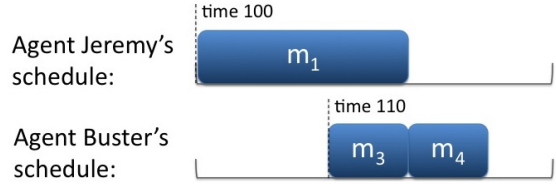


Figure 2: A schedule with three methods scheduled across two agents for the problem shown in Figure 1.

time distributions to determine both the probability of parent tasks accruing quality and their finish time distributions. The probability and finish time distributions at the task level depends on the type of node (e.g. *or*, *and*). Probability and finish time distributions are also adjusted if activities are targets of NLEs, such as a disabler increasing an activity’s likelihood of earning zero quality.

While calculating probability profiles for each activity, PADS finds the probability that each method will earn quality *given that it executes* (ignoring, for example, the fact that it may be disabled), or $pqe(m)$. Relevant methods’ pqe values can then be combined together as needed to find each method and task’s probability of earning quality overall, or $PQ(a)$. “Relevant” methods for an activity may include enablers, disablers, children (if the activity is a task), or the activity itself (if the activity is a method). A PQ value is represented in two parts: a list of pqe values for the relevant methods, and its aq function, which specifies how the relevant method probabilities should be combined to find the activity’s overall probability of accruing quality.

PADS also makes explicit two types of finish time distributions: the distribution of m_1 ’s finish times given it accrues quality, $FT_q(m_1)$, and the distribution given it does not accrue quality, $FT_{nq}(m_1)$. These distributions can be combined into an overall finish time distribution $FT(m_1)$ if needed. In the paragraphs below, we illustrate the analysis PADS performs on a simple schedule. As we progress, we will see why the above representations are necessary.

Consider the problem of Figure 1 with the simple schedule of Figure 2. We begin with method m_1 , which starts at time 100, has a deadline of 123, a duration distribution $((14\ 0.25)\ (16\ 0.375)\ (17\ 0.375))$, and fails with probability 20%. Because m_1 earns zero quality only when it fails, and its duration is the same whether it succeeds or fails, in this case $FT_q(m_1) = FT_{nq}(m_1)$:

$$\begin{aligned}
 PQ(m_1) &= (pqe(m_1) = 0.8) \text{ in } aq(m_1) \\
 FT_q(m_1) &= ((114\ 0.25)\ (116\ 0.375) \\
 &\quad (117\ 0.375)) \\
 FT_{nq}(m_1) &= ((114\ 0.25)\ (116\ 0.375) \\
 &\quad (117\ 0.375))
 \end{aligned}$$

Here $aq(m_1)$ intuitively reduces to $pqe(m_1)$, so $PQ(m_1) = 0.8$.

Now that we have found the probability profile of T_1 's only scheduled child, we can find T_1 's probability profile. The qaf of T_1 determines how the probability profiles of its children must be combined. Here T_1 has an *or* qaf, so its probability of accumulating quality equals the probability that at least one of its scheduled children do, and $PQ(T_1) = PQ(m_1)$. $FT_q(T_1)$ equals the distribution of the first child to succeed, and $FT_{nq}(T_1)$ equals the distribution of the last child to fail. T_1 's probability profile is:

$$\begin{aligned} PQ(T_1) &= (pqe(m_1) = 0.8) \text{ in } aq(m_1) \\ FT_q(T_1) &= ((114 \ 0.25) (116 \ 0.375) \\ &\quad (117 \ 0.375)) \\ FT_{nq}(T_1) &= ((114 \ 0.25) (116 \ 0.375) \\ &\quad (117 \ 0.375)) \end{aligned}$$

Again, $aq(m_1)$ reduces to $pqe(m_1)$, so $PQ(T_1) = 0.8$.

Probability profiles are found in an analogous manner for tasks with *and* qafs: PQ equals the probability that all children accrue quality, FT_q equals the distribution of the last child to succeed, and FT_{nq} equals the distribution of the first child to fail.

Next is m_3 , which has no failure outcome and cannot miss its deadline. Because m_3 always accrues quality, it suffices to represent $PQ(m_3)$ as a number.

$$\begin{aligned} PQ(m_3) &= 1 \\ FT_q(m_3) &= ((115 \ 0.5) (116 \ 0.5)) \\ FT_{nq}(m_3) &= () \end{aligned}$$

Following m_3 is m_4 , which is the target of an enables NLE. The type of an NLE determines how it affects the probability profile of target methods. In this case, T_1 affects the value of $PQ(m_4)$ because both activities must earn quality in order for m_4 to do so. To accommodate this, we define the operator \otimes such that $aq(x \otimes y)$ equals the probability that both x and y accrue quality. T_1 also affects m_4 's execution interval because m_4 cannot start until T_1 ends.

If T_1 earns quality, m_4 will execute with a start time distribution of $[\max(FT(m_3), FT_q(T_1))]$, indicating potential for m_4 to miss its deadline. If T_1 does not earn quality, m_4 will not know until T_1 ends that it cannot execute and should immediately fail; it will end in this case with distribution $[\max(FT(m_3), FT_{nq}(T_1))]$. Combining this together:

$$\begin{aligned} PQ(m_4) &= (pqe(m_1) = 0.8, pqe(m_4) = 0.625) \\ &\quad \text{in } aq(m_1 \otimes m_4) \\ FT_q(m_4) &= ((125 \ 0.4) (126 \ 0.6)) \\ FT_{nq}(m_4) &= ((115 \ 0.05) (116 \ 0.2) (117 \ 0.15) (127 \ 0.6)) \end{aligned}$$

Here, $aq(m_1 \otimes m_4)$ intuitively equals $pqe(m_1) \cdot pqe(m_4)$, and $PQ(m_4) = 0.5$.

The impact of disablers and soft NLEs on their target, in contrast to enablers, is limited to the probability that the

NLE source accrues quality before the target begins execution. These NLEs do not impact the target's start time distribution.

T_2 , an *and* task, has the same probability profile as m_4 , as m_4 is its last child and all earlier children will always accrue quality.

For *or* tasks such as TG , we define \ominus such that $aq(x \ominus y)$ equals the probability that at least one of x and y accrues quality:

$$\begin{aligned} PQ(TG) &= (pqe(m_1) = 0.8, pqe(m_4) = 0.625) \\ &\quad \text{in } aq(m_1 \ominus (m_1 \otimes m_4)) \end{aligned}$$

Because it now contains duplicates (i.e. m_1 appears more than once in the formula), aq is evaluated here by considering all combinations of the methods in the formula accruing and not accruing quality, weighted by the bound pqe values. It returns the sum of the probabilities of the combinations that lead to overall quality accrual as defined by the \otimes and \ominus operators. The above aq function returns 80%, as only combinations where m_1 accrues quality lead to TG accruing quality. This example highlights why we chose this type of representation over representing PQ as a single number: the latter would have led to incorrectly double-counting m_1 's influence in $PQ(TG)$.

The rest of TG 's probability profile is:

$$\begin{aligned} FT_q(TG) &= ((114 \ 0.25) (116 \ 0.375) \\ &\quad (117 \ 0.375)) \\ FT_{nq}(TG) &= ((114 \ 0.25) (116 \ 0.375) \\ &\quad (117 \ 0.375)) \end{aligned}$$

There is one additional point to make. We have seen above how to calculate PQ for both methods and tasks, which represents an activity's overall probability of earning quality, reflecting whether it is enabled or disabled. PQ is useful for determining the importance of an activity - i.e., how the total quality would change if that activity did not accrue quality, and did not enable its NLE targets, etc. A second probability of accruing quality is PLQ , which we define as an activity's local probability of earning quality given that it executes, i.e. ignoring enablers and disablers. To calculate PLQ at the method level, methods consider only their own pqe values and ignore the pqe values of enablers and disablers; at the task level PLQ is calculated by combining children's PLQ values in exactly the same way as described for PQ . PLQ is useful for targeting robustness improvements to high-risk activities.

With these probabilities in hand, we can calculate the expected quality of the activities in our schedule. A method's expected quality is simply $EQ(m_i) = PQ(m_i) \cdot qual(m_i)$; therefore,

$$\begin{aligned} EQ(m_1) &= 0.8 \cdot 12 = 9.6 \\ EQ(m_3) &= 1 \cdot 6 = 6 \\ EQ(m_4) &= 0.5 \cdot 8 = 4 \end{aligned}$$

For a *max* task, it is the expected max of its children:

$$EQ(T_1) = EQ(m_1) = 9.6$$

For *sum_and* tasks, it is the sum of the expected quality of its children *given they each earn quality* multiplied by the probability that the task earns quality. For shorthand we represent the expected quality of an activity given it earns quality, $EQ(a_1 | PQ(a_1) = 1)$, as $EQ(a_1 | a_1)$. Thus T_2 's expected quality is:

$$\begin{aligned} EQ(T_2) &= PQ(T_2) \cdot (EQ(m_3 | m_3) + EQ(m_4 | m_4)) \\ &= 0.5 \cdot (6 + 8) = 7 \end{aligned}$$

For *sum* tasks, it is the sum of the expected quality of its children:

$$EQ(TG) = EQ(T_1) + EQ(T_2) = 16.6$$

We can also calculate the expected quality loss of an activity, $EQ_L(a_i) = EQ(TG|a_i) - EQ(TG|\neg a_i)$, which represents how much an activity earning zero quality would impact the quality of the overall schedule. For example, T_1 's expected quality loss can be calculated as follows (note that assuming T_1 succeeds is the same as assuming m_1 succeeds):

$$\begin{aligned} EQ_L(T_1) &= EQ(TG | m_1) - EQ(TG | \neg m_1) \\ &= (EQ(T_1 | m_1) + EQ(T_2 | m_1)) \\ &\quad - (EQ(T_1 | \neg m_1) + EQ(T_2 | \neg m_1)) \\ &= (EQ(m_1 | m_1) \\ &\quad + PQ(T_2 | m_1) \cdot (EQ(m_3 | m_1, m_3) \\ &\quad \quad + EQ(m_4 | m_1, m_4)) \\ &\quad - (0 + 0)) \\ &= 12 + 0.625 \cdot (6 + 8) - 0 = 23 \end{aligned}$$

Targeted Backfilling

Targeted backfilling uses the probability profiles and expected quality calculations described above to focus backfilling effort on high-risk max-leaves under the constraint that expected schedule quality not be reduced due to schedule oversaturation. Beginning with an initial schedule, the list of its scheduled max-leaves is used as the basis for backfilling. The list is sorted in decreasing EQ_L order and each task is sequentially considered for backfilling. This results in backfilling higher quality loss tasks first, before the schedules are at risk of becoming saturated, as Figure 3 confirms.

Targeted backfilling attempts to schedule an additional child under a task if the task has a PLQ less than one. This condition indicates a potential problem with the task, such as a missed deadline or method failure outcome, and suggests a benefit from backfilling. If scheduling the extra child succeeds, the new schedule's expected quality is compared with the expected quality before the child was scheduled. If the expected quality has not decreased, the backfilling is accepted, all probabilities and expected quality losses are

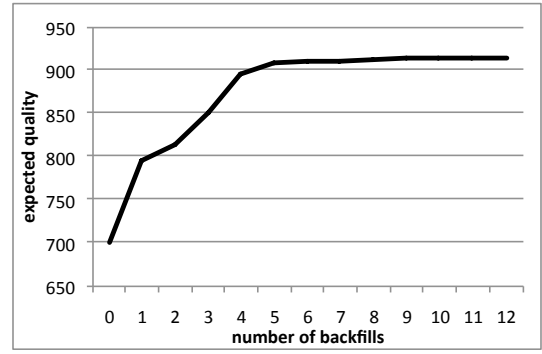


Figure 3: Schedule expected quality for a single problem, shown as a function of how many targeted backfills have been performed.

recomputed, the max-leaf list is resorted, and the backfilling process begins anew. Otherwise, the added child is unscheduled. Targeted backfilling terminates when no additional methods can be added under these criteria.

Experiments and Results

Using a multi-agent simulation environment, we ran an experiment to quantify the effect of targeted backfilling on quality earned when executing schedules. Within this environment, each agent manages the execution of its own schedule. Agents communicate method start commands as appropriate to a simulator, which operates on a separate thread. They receive back method execution results, based on the stochastics of the problem description, from the simulator as they become known, and pass status information to other agents asynchronously. All agents and the simulator run on separate threads on a 2.4 GHz Intel Core 2 Duo computer with 2GB RAM. If a conflict arises in an agent's schedule during execution, the only action the agent can take is to remove the conflicting method from the timeline (i.e. there is no rescheduling). This allows us to compare directly the relative robustness of different schedules by comparing the quality earned at the end of the simulation.

We ran simulations under three different backfill modes. (1) is initializing the agents with the initial schedule generated by the deterministic, centralized planner (no backfilling), and (2) is initializing the agents with the reinforced schedule produced by targeted backfilling. We compare these against a third mode that provides the schedule strengthening benefits of backfilling without either the algorithmic overhead or the probabilistic focus of targeted backfilling, *unconstrained backfilling*. The unconstrained backfilling algorithm loops repeatedly through the set of all scheduled max-leaf tasks and attempts to add one additional method under each task in each pass. The algorithm will repeatedly loop over the set until it can no longer backfill under any of the max-leaves due to lack of space on agent timelines.

Table 1: Characteristics of test suite problems

	min	max	average
methods	30	139	70.7
max-leaves	15	113	58.7
tasks	40	167	91.9
enabling NLEs	0	40	12.2
disabling NLEs	0	8	3.9
facilitating NLEs	0	20	7.6
hindering NLEs	0	16	5.9

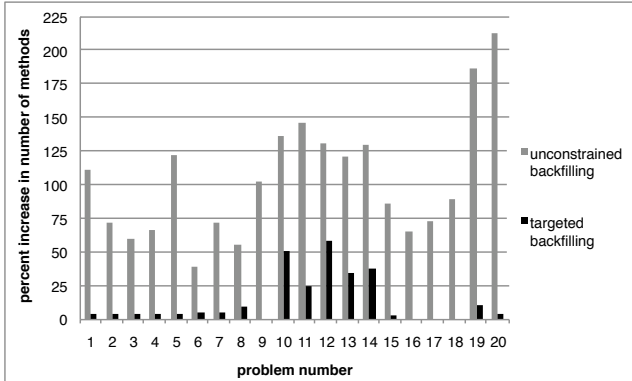


Figure 4: The percent increase of the number of methods in the initial schedule after backfilling for each mode.

We created schedules according to the above 3 modes for 20 separate test problems, taken from the Phase II evaluation test problem suite of the DARPA Coordinators program¹. Fifteen of these problems involved 20 agents, 3 involved only 10 agents, and 2 involved 25 agents. Table 1 shows the minimum, maximum and average numbers of other features in the initially scheduled (not backfilled) problems. All initial schedules (with and without backfilling) were generated on a 3.6 GHz Intel computer with 2GB RAM.

To ensure a fair comparison between modes while randomly sampling duration and outcome distributions, we numbered the 10 runs per problem per mode $i = 1 : 10$. Then, for any run i for some problem p , we randomly chose in advance the durations and outcomes for all methods according to the random seed i , and provided them to the simulator to use at run time. The durations and outcomes were then consistent across modes: for any scheduled method m and two modes c_1 and c_2 of some run i of problem p , $[dur(m_{ic_1}) = dur(m_{ic_2})]$ and $[outcome(m_{ic_1}) = outcome(m_{ic_2})]$.

Figure 4 shows the percent increase in the number of methods in the initial schedule after backfilling for the two modes that performed backfilling. Unconstrained backfilling far exceeds targeted backfilling in the number of backfills. In all cases but one, targeted backfilling added at least 1 method. In this and all future graphs, problems are sorted in increasing order of the number of methods initially scheduled.

¹<http://www.darpa.mil/ipto/programs/coor/coor.asp>

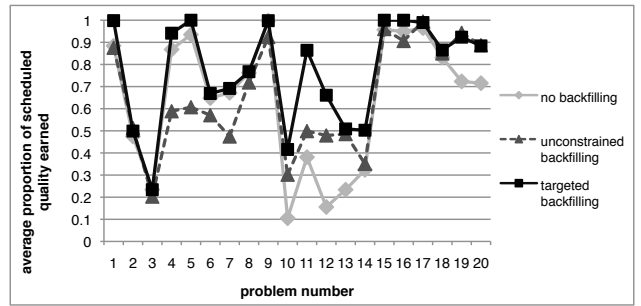


Figure 5: The proportion of quality of the original schedule earned by each mode, averaged over 10 runs.

The average quality earned by the three modes for each of the 20 problems is shown in Figure 5. The quality is expressed as the proportion of the scheduled quality of the original initial schedule earned during execution. On all but 4 of the problems, targeted backfilling earned the highest percentage of the scheduled quality. In the four problems that unconstrained backfilling wins, it does so marginally. On average across all problems, no backfilling earned a proportional quality of 0.64, unconstrained backfilling earned a proportional quality of 0.66, and targeted backfilling earned a proportional quality of 0.77. Using a repeated measures ANOVA, targeted backfilling is significantly better than either other mode with $p < 0.02$. Unconstrained backfilling and no backfilling are not statistically different; although unconstrained backfilling sometimes outperformed no backfilling, it also sometimes performed worse than no backfilling.

Overall, these results clearly illustrate the trade-off associated with reinforcing schedules in resource constrained circumstances. There is certainly advantage to be gained by adding redundant methods, as seen by comparing the targeted backfilling results to the no backfilling results; however, this advantage can be largely negated if attention is not paid to timeline congestion, as seen by comparing the no backfilling results to the relatively similar results of unconstrained backfilling. Our explicit use of probability profiles provides a means of effectively balancing this trade-off.

Looking more closely at the 4 problems where unconstrained backfilling outperformed targeted backfilling, we observed that these results in some cases are due to occasional sub-par scheduling by the deterministic scheduler. For example, under some circumstances, the method allocated by the scheduler under a max-leaf may not be the highest quality child, even though there is enough timeline space for the latter. Unconstrained backfilling may correct this mistake coincidentally due to its undirected nature, while targeted backfilling will not even consider this max-leaf if the originally scheduled node is not in danger of earning no quality. We expect that incorporation of a better deterministic scheduler would hence likely increase the advantage of targeted backfilling over this simplistic alternative.

Future Work and Conclusions

One direction of our current work focuses on developing additional, complementary mechanisms for schedule improvement. We would like, for example, to extend the backfilling mechanism to also include the option of substituting lower quality, but shorter, methods with lower failure likelihoods for the originally scheduled child. Although it may result in a lower quality for that task, it also does not crowd agent timelines and so may prove useful for very inflexible schedules.

Our longer term goal is to strengthen schedules in a distributed fashion, so it can be done throughout execution in a multi-agent environment. In the larger class of problems we are interested in, new activities and constraints can be introduced during execution and agents must continually adjust their schedules to reflect these evolving circumstances. In this context, a run-time version of PADS would serve as the basis of a plan-monitoring mechanism, allowing agents to recognize and fortify weaknesses in the schedule dynamically as they arise.

Operating in an online execution mode places a sharp focus on the running time of PADS. For our largest problem, which had 25 agents, 139 initially scheduled methods and 113 scheduled max-leaves, targeted backfilling took 24.3 seconds, in contrast to 1.4 seconds for unconstrained backfilling. This might cause concern as we move to an execution-time analysis in a distributed setting; however, to address the issue we point out that the PADS algorithm is inherently ‘anytime’ in nature, as the backfilling loop can be interrupted at any time. Figure 3 reinforces the appeal of the algorithm in this regard, as the most profitable backfills are done first due to the sort order of the max-leaves. Consequently, we believe that the algorithm will be scalable and effective as an execution-time algorithm.

Overall, we have shown that by analyzing deterministic schedules of uncertain domains using the problems’ uncertainty model, we can make more effective decisions about how to make a schedule more robust in the face of the uncertainties of execution, at a low computational cost.

References

- Beck, J. C., and Wilson, N. 2007. Proactive algorithms for job shop scheduling with probabilistic durations. *Journal of Artificial Intelligence Research* 28:183–232.
- Bertoli, P.; Cimatti, A.; Roverie, M.; and Traverso, P. 2001. Planning in nondeterministic domains under partial observability via symbolic model checking. In *Proceedings of IJCAI ’01*.
- Blum, A., and Langford, J. 1999. Probabilistic planning in the graphplan framework. In *Proceedings of the Fifth European Conference on Planning (ECP ’99)*.
- Boddy, M.; Horling, B.; Phelps, J.; Goldman, R.; Vincent, R.; Long, C.; and Kohout, B. 2005. C-TAEMS language specification v. 1.06.
- Bryce, D.; Mausam; and Yoon, S. 2008. Workshop on a reality check for planning and scheduling under uncertainty. <http://www.ai.sri.com/bryce/ICAPS08-workshop.html>.
- Decker, K. 1996. TAEMS: A framework for environment centered analysis & design of coordination mechanisms. In O’Hare, G., and Jennings, N., eds., *Foundations of Distributed Artificial Intelligence*. Wiley Inter-Science. chapter 16, 429–448.
- Drummond, M.; Bresina, J.; and Swanson, K. 1994. Just-in-case scheduling. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*.
- Little, I., and Thiébaux, S. 2006. Concurrent probabilistic planning in the graphplan framework. In *Proceedings of ICAPS ’06*.
- Mausam, and Weld, D. S. 2006. Probabilistic temporal planning with uncertain durations. In *Proceedings of AAAI-06*.
- McKay, K.; Morton, T.; Ramnath, P.; and Wang, J. 2000. Aversion dynamics’ scheduling when the system changes. *Journal of Scheduling* 3(2).
- Morris, P.; Muscettola, N.; and Vidal, T. 2001. Dynamic control of plans with temporal uncertainty. In *IJCAI*, 494–502.
- Musliner, D. J.; Carciofini, J.; Durfee, E. H.; Wu, J.; Goldman, R. P.; and Boddy, M. S. 2007. Flexibly integrating deliberation and execution in decision-theoretic agents. In *Proceedings of the 3rd Workshop on Planning and Plan Execution for Real-World Systems (held in conjunction with ICAPS 2007)*.
- Onder, N.; Whelan, G. C.; and Li, L. 2006. Engineering a conformant probabilistic planner. *Journal of Artificial Intelligence Research* 25:1–15.
- Policella, N.; Cesta, A.; Oddi, A.; and Smith, S. F. 2006. From precedence constraint posting to partial order schedules. a CSP approach to robust scheduling. *AI Communications* 20(3):163–180.
- Rossi, F.; Venable, K. B.; and Yorke-Smith, N. 2006. Uncertainty in soft temporal constraint problems: A general framework and controllability algorithms for the fuzzy case. *Journal of Artificial Intelligence Research* 27:617–674.
- Smith, D. E., and Weld, D. S. 1998. Conformant graphplan. In *Proceedings of AAAI-98*.
- Smith, S. F.; Gallagher, A.; Zimmerman, T.; Barbulescu, L.; and Rubinstein, Z. 2007. Distributed management of flexible times schedules. In *Proceedings of the 2007 International Conference on Autonomous Agents and Multi-agent Systems (AAMAS)*.
- Younes, H. L. S., and Simmons, R. G. 2004a. Policy generation for continuous-time stochastic domains with concurrency. In *Proceedings of ICAPS ’04*.
- Younes, H. L. S., and Simmons, R. G. 2004b. Solving generalized semi-markov decision processes using continuous phase-type distributions. In *Proceedings of AAAI-04*.