Probabilistic Plan Management

Laura M. Hiatt

CMU-CS-09-170
November 17, 2009

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Reid Simmons, chair
Stephen F. Smith
Manuela Veloso
David J. Musliner, SIFT

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

This research was sponsored by the National Aeronautics and Space Administration under grant numbers NNX06AD23G, NNA04CK90A, and NNA05CP97A; Naval Research Laboratory under grant number N00173091G001; Defense Advanced Research Projects Agency under grant number FA8750-05-C-0033; and National Science Foundation under grant numbers DGE-0750271, DGE-0234630, and DGE-0750271.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
17 NOV 2009		**00-00-2009 to 00-00-2009**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Probabilistic Plan Management	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Carnegie Mellon University ,School of Computer Science,Pittsburgh,PA,15213**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**214**	

Keywords: planning under uncertainty, robust planning, meta-level control, cooperative multi-agent systems, plan management, artificial intelligence

Abstract

The general problem of planning for uncertain domains remains a difficult challenge. Research that focuses on constructing plans by reasoning with explicit models of uncertainty has produced some promising mechanisms for coping with specific types of domain uncertainties; however, these approaches generally have difficulty scaling. Research in robust planning has alternatively emphasized the use of deterministic planning techniques, with the goal of constructing a flexible plan (or set of plans) that can absorb deviations during execution. Such approaches are scalable, but either result in overly conservative plans, or ignore the potential leverage that can be provided by explicit uncertainty models.

The main contribution of this work is a composite approach to planning that couples the strengths of both the above approaches while minimizing their weaknesses. Our approach, called *Probabilistic Plan Management* (PPM), takes advantage of the known uncertainty model while avoiding the overhead of non-deterministic planning. PPM takes as its starting point a deterministic plan that is built with deterministic modeling assumptions. PPM begins by layering an uncertainty analysis on top of the plan. The analysis calculates the overall expected outcome of execution and can be used to identify expected weak areas of the schedule.

PPM uses the analysis in two main ways to maximize the utility of and manage execution. First, it makes deterministic plans more robust by minimizing the negative impact that unexpected or undesirable contingencies can have on plan utility. PPM strengthens the current schedule by fortifying the areas of the plan identified as weak by the probabilistic analysis, increasing the likelihood that they will succeed. In experiments, probabilistic schedule strengthening is able to significantly increase the utility of execution while introducing only a modest overhead.

Second, PPM reduces the amount of replanning that occurs during execution via a probabilistic meta-level control algorithm. It uses the probability analysis as a basis for identifying cases where replanning probably is (or is not) necessary, and acts accordingly. This addresses the trade-off of too much replanning, which can lead to the overuse of computational resources and lack of responsiveness, versus too little, which can lead to undesirable errors or missed opportunities during execution. Experiments show that probabilistic meta-level control is able to considerably decrease the amount of time spent managing plan execution, without affecting how much utility is earned. In these ways, our approach effectively manages the execution of deterministic plans for uncertain domains both by producing effective plans in a scalable way, and by intelligently controlling the resources that are used maintain these high-utility plans during execution.

Acknowledgments

I would first like to thank my advisor, Reid Simmons, for being such a great mentor through this process. This would not have been possible without all of his support, questions, and encouragement, and I cannot thank him enough.

Thanks also to Steve Smith, for letting me join his project for part of this thesis. I really enjoyed the opportunity to apply this work to a real domain. Along those lines, thanks to the rest of the ISLL group for the many hours discussing the intricacies of C_TAEMS and the Coordinators project. Terry Zimmerman gets a special thanks for suggesting the "PADS" acronym, saving everyone from days of repeating "probabilistic analysis of deterministic schedules" over and over again; but also, of course, thanks to Zack Rubenstein, Laura Barbulescu, Anthony Gallagher, Matt Danish, Charlie Collins and Dave Crimm.

Thanks to Dave Musliner and Manuela Veloso for being part of my committee, and for all of their good suggestions and guidance.

Thanks to Brennan Sellner for lending me his code for and helping me with ASPEN, and for sharing his ASPEN contacts with me to provide additional support. Thanks also to him and the rest of the Trestle/IDSR group – Fred Heger, Nik Melchior, Seth Koterba, Brad Hamner and Greg Armstrong – for all the fun times in the highbay, even if sometimes the "fun" was in a painful way.

I would like to thank my mother and my sister for not giving me too much trouble about never having a "real job," and for consistently expressing (or, at least, pretending to express) interest in what I've been up to these many years. I would also like to firmly point out to them, this one last time, that my robots are not taking over the world.

Finally, I want to thank my husband, Stephen, for, well, everything.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

We are motivated by the high-level goal of building intelligent robotic systems in which robots, or teams of robots and humans, operate autonomously in real-world situations. The rich sources of uncertainty common in these scenarios pose challenges for the various aspects of execution, from sending commands to the controller to navigate over unpredictable terrain, to deciding which activities to execute given unforeseen circumstances. Many domains of this type are further complicated by the presence of durative actions and concurrent, multi-agent execution. This thesis focuses on the high-level aspect of the problem: generating, executing and managing robust plans for agents operating in such situations.

Not surprisingly, the general problem of planning and scheduling for uncertain domains remains a difficult challenge. Research that focuses on constructing plans (or policies) by reasoning with explicit models of uncertainty has produced some promising results (*e.g.*, [McKay et al., 2000, Beck and Wilson, 2007]). Such approaches typically produce optimal or expected optimal solutions that encode a set of plans that cover many or all of the contingencies of execution. However, in complex domains these approaches can quickly become prohibitively expensive, making them undesirable or impractical for real-world use [Bryce et al., 2008].

Research in robust planning, in contrast, has emphasized the use of expected models and deterministic planning techniques, with the goal of constructing a flexible plan (or set of plans) that can absorb deviations at execution time [Policella et al., 2007]. These approaches are much more scalable than probabilistic planning or scheduling, but either result in overly conservative schedules at the expense of performance (*e.g.*, [Morris et al., 2001]) or ignore the potential leverage that can be provided by an explicit uncertainty model. For example, even though such systems typically replan during execution, the deterministic assumptions unavoidably can create problems to which

replanning cannot adequately respond, such as a missed deadline that cannot be repaired. Replanning very frequently can help to alleviate the number of these problems that occur, but a large amount of online replanning has the potential to lead to a lack of responsiveness and an overuse of computational resources during execution, ultimately resulting in a loss of utility.

The main contribution of this work is adopting a composite approach to planning and scheduling that couples the strengths of both deterministic and non-deterministic planning while minimizing their weaknesses. Our approach, called *Probabilistic Plan Management* (PPM), is a natural synthesis of both of the above research streams that takes advantage of the known uncertainty model while avoiding the large computational overhead of non-deterministic planning. As a starting point, we assume a deterministic plan, or schedule, that is built to maximize utility under deterministic modeling assumptions, such as by assuming deterministic durations for activities or by ignoring unlikely outcomes or effects of activity execution. Our approach first layers an uncertainty analysis, called a *probabilistic analysis of deterministic schedules* (PADS), on top of the deterministic schedule. The analysis calculates the overall expected outcome of execution and can be used to identify expected weak areas of the schedule. We use this analysis in two main ways to manage and bolster the utility of execution. *Probabilistic schedule strengthening* strengthens the current schedule by targeting local improvements at identified likely weak points. Such improvements have inherent trade-offs associated with them; these trade-offs are explicitly addressed by PADS, which ensures that schedule strengthening efforts increase the schedule's expected utility. *Probabilistic meta-level control* uses the expected outcome of the plan to intelligently control the amount of replanning that occurs, explicitly addressing the trade-off of too much replanning, which can lead to the overuse of computational resources and lack of responsiveness, versus too little, which can lead to undesirable errors or missed opportunities during execution.

Probabilistic plan management combines these different components to effectively manage the execution of plans for uncertain domains. During execution, PADS tracks and updates the probabilistic state of the schedule. Based on this, PPM makes the decision of when replanning and probabilistic schedule strengthening is necessary, and replans and strengthens at the identified times, maintaining high utility schedules while also limiting unnecessary replanning.

In the following sections, we introduce the concepts of PADS, probabilistic schedule strengthening, and probabilistic meta-level control, and our basic approaches to them.

1.1 Probabilistic Analysis of Deterministic Schedules

Central to our approach is the idea of leveraging domain uncertainty models to provide information that can help with more effective execution-time management of deterministic schedules. We have developed an algorithm that performs a *P*robabilistic *A*nalysis of *D*eterministic *S*chedules (PADS), given their associated uncertainty model. At the highest level of abstraction, the algorithm explores all known contingencies within the context of the current schedule in order to find the probability that execution "succeeds" by meeting its objective. It calculates, for each activity, the probability of the activity meeting its objective, as well as its expected contribution to the schedule. By explicitly calculating these values, PADS is able to summarize the expected outcome of execution, as well as identify portions of the schedule where something is likely to go wrong.

PADS can be used as the basis for many planning decisions. In this thesis, we discuss how it can be used to strengthen plans to increase the expected utility of execution, and how it can be used as the basis for different meta-level control strategies that control the computational resources used to replan during execution. Further, we show how its low computational overhead makes it a useful analysis for even complex, multi-agent domains.

1.2 Probabilistic Schedule Strengthening

While deterministic planners have the benefit of being more scalable than probabilistic planners, they make strong, and often inaccurate, assumptions on how execution will unfold. In the case of domains with temporal uncertainty (*e.g.*, uncertain activity durations), such as the ones we consider, deterministic modeling assumptions have the potential to lead to frequent constraint violations when activities end at unanticipated times and cause inconsistencies in the schedule. Replanning during execution can effectively respond to such unexpected dynamics, but only as long as dependencies allow sufficient time for one or more agent planners to respond to problems that occur; further, even if the agent is able to sufficiently respond to a problem, utility loss may occur due to the last-minute nature of the schedule repair. One way to prevent this situation is to layer a step on top of the planning process that proactively tries to prevent these problems from occurring, or to minimize their effects, thereby strengthening schedules against the uncertainty of execution [Gallagher, 2009].

Schedules are strengthened via strengthening actions, which make small modifications to the schedule to make them more robust. The use of such improvements, however, typically has an associated trade-off. Example strengthening actions, and their trade-offs, are:

1. Adding redundant back-up activities - This provides protection against activity failure but can clog schedules and reduce slack. In a search and rescue scenario, for example, trying to reach survivors via two rescue points, instead of one, may increase the likelihood of a successful rescue; but, as a result, the agents at the added rescue point may not have enough time to execute the rest of their schedule.
2. Swapping activities for shorter-duration alternatives - This provides protection against an activity missing a deadline, but shorter-duration alternatives tend to be less effective or have a lower utility. For example, restoring power to only critical buildings instead of the entire area may allow power to be restored more quickly, but it is less desirable as other buildings go without power.
3. Reordering scheduled activities - This may or may not help, depending on which activities have tight deadlines and/or constraints with other activities. Deciding to take a patient to the hospital before refueling, for example, may help the patient, but the driver then has a higher risk of running out of fuel and failing to perform other activities.

The set of strengthening actions used can be adapted to suit a variety of domains, depending on what is most appropriate for the domain's structure. Ideally, schedule strengthening should be used in conjunction with replanning during execution, allowing agents to take advantage of opportunities that may arise while also ensuring that they always have schedules that are robust to unexpected or undesirable outcomes.

In deterministic systems, schedule strengthening can be performed by using simple deterministic heuristics, or limited stochastic analyses of the current schedule (*e.g.*, considering the uncertainty of the duration of an activity without any propagation) [Gallagher, 2009]. In this thesis, we argue that explicitly reasoning about the uncertainty associated with scheduled activities, via our PADS algorithm, provides a more effective basis for strengthening schedules. *Probabilistic schedule strengthening* first relies on PADS to identify the most profitable activities to strengthen. Then, improvements are targeted at activities that have a high likelihood of not doing their part to contribute to the schedule's success, and whose failure would have the largest negative impact. As described, all such improvements have an associated trade-off; PADS, however, ensures that all strengthening actions have a positive expected benefit and improve the expected outcome of execution.

We have evaluated the effect of probabilistic schedule strengthening in a multi-agent simulation environment, where the objective is to maximize utility earned during execution and there is temporal and activity outcome uncertainty. Probabilistic schedule strengthening produced schedules that are significantly more robust and earn 32% more utility than their deterministic counterparts when executed. When it is used in conjunction with replanning during execution, it results in 14%

more utility earned as compared to replanning alone during execution.

1.3 Probabilistic Meta-Level Control

Replanning provides an effective way of responding to the dynamics of execution: it helps agents avoid failure or utility loss resulting from unpredicted and undesirable outcomes, and takes advantage of opportunities that may arise from unpredicted favorable outcomes. Replanning too much during execution, however, has the potential to become counter-productive by leading to a lack of responsiveness and a waste of computational resources. Meta-level control is one technique commonly used to address this trade-off. Meta-level control allows agents to optimize their performance by selectively choosing deliberative actions to perform during execution.

In deterministic systems, meta-level control can be done by learning a model for deliberative action selection based on deterministic, abstract features of the current schedule [Raja and Lesser, 2007]. Deterministic approaches such as this one, however, ignore the benefit that explicitly considering the uncertainty model can provide. For example, whether replanning would result in a better schedule can depend, in part, on the probability of the current schedule meeting its objective; similarly, the benefit of planning actions such as probabilistic schedule strengthening can be greatly affected by the robustness of the current schedule. In this thesis, we show that a *probabilistic meta-level control algorithm*, based on probability profile information obtained by the PADS algorithm, is an effective way of selecting when to replan (or probabilistically strengthen) during execution.

Ideally, agents would replan if it would be useful either in avoiding failure or in taking advantage of new opportunities. Similarly, they ideally would not replan if it would not be useful, either because: (1) there is no potential failure to avoid; (2) there is a possible failure but replanning would not fix it; (3) there is no new opportunity to leverage; (4) or replanning would not be able to exploit the opportunities that were there. Many of these situations are difficult to explicitly predict, in large part because it is difficult to predict whether a replan is beneficial without actually doing it. As an alternative, probabilistic meta-level control uses the probabilistic analysis to identify times where there is potential for replanning to be useful, either because there is likely a new opportunity to leverage or a possible failure to avoid. When such times are identified, no replanning is done; otherwise, replanning is possibly useful and so should be performed.

The control strategy can also be used to select the amount of the current schedule that is replanned. In situations with high uncertainty, always replanning the entire schedule means that activities farther down the schedule are potentially replanned many times before they are actually executed, possibly wasting computational effort. To avoid this, we introduce an *uncertainty hori-*

zon, which distinguishes between portions of the schedule that are likely to need to be managed and replanned at the current time, and portions that are not. This distinction is made based on the probability that activities beyond the uncertainty horizon will need to be replanned before execution reaches them.

We have shown, through experimentation, the effectiveness of probabilistic meta-level control. Its use in a single-agent scenario with temporal uncertainty has shown a 23% reduction in schedule execution time (where execution time is defined as the schedule makespan plus the time spent replanning during execution) versus replanning whenever a schedule conflict arises. Using probabilistic meta-level control in a multi-agent scenario with temporal and activity outcome uncertainty results in a 49% decrease in plan management time as compared to a plan management strategy that replans according to deterministic heuristics, with no significant change in utility earned.

1.4 Thesis Statement

Explicitly analyzing the probabilistic state of deterministic plans built for uncertain domains provides important information that can be leveraged in a scalable way to effectively manage plan execution.

We support this statement by presenting algorithms for *probabilistic plan management* that probabilistically analyze schedules, strengthen schedules based on this analysis, and limit the amount of computational effort spent replanning and strengthening schedules during execution. Our approach was validated through a series of experiments in simulation for two different domains. For one domain, using probabilistic plan management was about to reduce schedule execution time by 23% (where execution time is defined as the schedule makespan plus the time spent replanning during execution). For another, probabilistic plan management resulted in increasing utility earned by 14% with only a small added overhead. Overall, PPM provides some of the benefits of non-deterministic planning while avoiding its computational overhead.

1.5 Thesis Outline

The rest of this document is organized as follows. In Chapter 2, we discuss related work, describing various approaches to the problem of planning and scheduling for uncertain domains, as well as different methodologies for meta-level control. In Chapter 3, we discuss background concepts related to our work, including details on the two domains we consider. Next, Chapter 4 provides an overview of our approach to probabilistic plan management. The following three chapters go into

more detail about the three main components of our approach: probabilistic analyses of deterministic schedules, probabilistic schedule strengthening, and probabilistic meta-level control. Finally, in Chapter 8, we discuss future work and conclusions.

Chapter 2

Related Work

There are many different approaches that handle planning and execution for uncertain domains. Approaches can be distinguished by what restrictions they place on the domains they can handle, such as whether they consider durative actions and temporal uncertainty, and whether they account for concurrency and multiple agents. We present some of these approaches, categorized by whether uncertainty is taken into account in the planning process or planning is done deterministically. We also discuss various approaches to the meta-level control of computation during execution, including work utilizing planning horizons.

2.1 Non-Deterministic Planning

The main distinction between planners that deal with uncertainty is whether they assume agents can observe the world during execution. Of those that make this assumption, there are policy-based approaches based on the basic Markov Decision Process (MDP) model [Howard, 1960] and more search-based approaches such as contingency planners. Some MDP models also assume that the world is partially observable. Planning approaches that assume the world is not observable are commonly referred to as conformant planners.

2.1.1 Policy-Based Planning

One common policy-based planner is based on an MDP model [Howard, 1960]. The classic MDP model has four components: a collection of states, a collection of actions, a transition model (*i.e.*, which states actions transition between, and the probability of the transition happening), and the

rewards associated with each transition (or state). The MDP can be solved to find the optimal policy that maximizes reward based on the problem specification.

This classic MDP model is unable to handle explicit durative actions, which can be considered by our approach. There is a growing body of work, however, that has been done to remove this limitation. Semi-MDPs, for example, handle domains with uncertain action durations and no deadlines [Howard, 1971]. Time-dependent MDPs [Boyan and Littman, 2000] go one step further and can model deadlines by undiscounting time in the MDP model. Hybrid MDPs can also handle deadlines, and can represent both discrete and continuous variables. One of the main difficulties of this model is convolving the probability density functions and value functions while solving the model; this has been approached in a variety of ways [Feng et al., 2004, Li and Littman, 2005, Marecki et al., 2007].

None of the above MDP models, however, can handle concurrency, and so cannot be used for multi-agent scenarios. Generalized Semi-MDPs, an extension of Semi-MDPs [Younes et al., 2003, Younes and Simmons, 2004a], can handle concurrency and actions with uncertain, continuous durations, at the cost of very high complexity. The DUR family of MDP planners can also handle both concurrency and durative actions [Mausam and Weld, 2005], again at a very high computational cost. The work shows, in fact, that the ΔDUR planner that plans with expected durations outperforms the ΔDUR planner that explicitly models temporal uncertainty [Mausam and Weld, 2006].

In addition to the MDP systems above, MDPs have been used to solve a distributed, multi-agent domain with activity duration and outcome uncertainty, the same domain that we use as part of this thesis work [Musliner et al., 2006, 2007]. This domain is especially challenging for MDP approaches, as plans must be constructed in real time. Each agent in this approach has its own MDP, which it partially and approximately solves in order to generate a solution in acceptable time. However, despite this success, scalability ultimately prevented it from performing better than competing deterministic approaches.

MDPs that assume the world is partially observable, or POMDPs, assume that agents can make some local observations during execution but cannot directly observe the true state - instead, agents must probabilistically infer it [Kaelbling et al., 1998]. This assumption makes POMDP's very difficult to solve, and the additional complexity is unnecessary for the domains considered in this document, where the world is fully observable.

A non-MDP policy-based approach is FPG, a Factored Policy Gradient planner [Buffet and Aberdeen, 2009]. This planner uses gradient ascent to optimize a parameterized policy, such as if users prefer a safer, or more optimal but riskier, plan. It still, however, suffers from the high complexity of policy-based probabilistic planners, and the authors suggest that it may be improved

by combining deterministic reasoning with their probabilistic approach in order to allow it to handle larger and more difficult policies.

2.1.2 Search-Based Contingency Planning

Conditional, or contingency, planners create branching plans, and assume observations during execution to choose which branch of the plan to follow. Many contingency planners such as SGP [Weld et al., 1998], MBP [Bertoli et al., 2001], PGraphPlan [Blum and Langford, 1999] and Paragraph [Little and Thiébaux, 2006], however, are unable to support durative actions. Partial-order contingency approaches include C-Buridan [Draper et al., 1994], B-Prodigy [Blythe and Veloso, 1997], DTPOP [Peot, 1998] and Mahinur [Onder and Pollack, 1999]. They do not support concurrency or durative actions; it is possible to add them to these architectures, but at an even higher computational cost.

[Xuan and Lesser, 1999, Wagner et al., 2006] consider a domain similar to ours, and use contingency planning in their scheduling process and to manage the uncertainty between agents, but at the cost of high complexity. Prottle uses forward search planning to solve a concurrent, durative problem over a finite horizon [Little et al., 2005]. Although it uses a probabilistic planning graph-based heuristic to reduce the search space, memory usage prevents this algorithm from being applied to anything but small problems.

Some work tries to lessen the computational burden of contingency planners by planning incrementally. These approaches share some similarities with our approach, as they try to identify weak parts of the plan to improve at plan time. One common incremental approach is Just-In-Case (JIC) scheduling [Drummond et al., 1994], which creates branch points in a schedule by finding the place where it is most likely to fail based on normal task distributions overrunning their scheduled duration. For simple distribution types, this involves performing probability calculations similar to how we calculate our uncertainty horizon for single-agent schedules.

ICP [Dearden et al., 2003] and Tempastic [Younes and Simmons, 2004b] also build contingency plans incrementally, by branching on utility gain and fixing plan failure points, respectively. Both of these approaches use stochastic simulation to find plan outcome probabilities. [Blythe, 1998] calculates plan certainty by translating the plan into a Bayesian network, and adding contingent branches to increase the probability of success. [Meuleau and Smith, 2003]'s OKP generates the optimal plan having a limited number of contingency branches. These incremental approaches make contingency planning more manageable or tractable; however, the approach in general is still unable to model large numbers of outcomes. Due to its difficulty, contingency planning is best for dealing with avoiding unrecoverable or critical errors in domains with small numbers of outcomes

as in [Foss et al., 2007].

2.1.3 Conformant Planning

Conformant planners such as [Goldman and Boddy, 1996], CGP [Smith and Weld, 1998], CMBP [Cimatti and Roveri, 2000], BURIDAN [Kushmerick et al., 1995] and PROBAPOP [Onder et al., 2006] handle uncertainty by developing plans that will lead to the goal without assuming that the results of actions are observable. A common way to think about conformant planners is that they try force the world into a single, known state by generating "fail-safe" plans that are robust to anything that may happen during execution. Conformant planning is very difficult, and also very limiting; typically, effective conformant planning requires domains with limited uncertainty and very strong actions. Additionally, none of the above conformant planners can currently handle durative actions.

Planning to add "robustness" is similar to conformant planning, and has the goal of building plans that are robust to various types of uncertainty. [Fox et al., 2006] "probes" a temporal plan for robustness by testing the effectiveness of different slight variations of the plan, where each variation is generated by stochastically altering the timings of execution points in the plan. [Blackmore et al., 2007] build plans robust to temporal and other types of uncertainty by utilizing sampling techniques such as Monte Carlo simulation to approximately predict the outcome of execution. In [Younes and Musliner, 2002], a probabilistic extension to the CIRCA planner [Musliner et al., 1993] is described that uses Monte Carlo simulation to verify whether a potential deterministic plan meets specified safety constraints. [Beck and Wilson, 2007] also uses Monte Carlo simulation to generate effective deterministic schedules for the job-shop scheduling problem. Using Monte Carlo simulation in this way, however, is not practical for our run-time analysis, as the number of particles needed increases quickly with the amount of domain uncertainty; in contrast, our approach analytically calculates and stores probability information for each activity. [Fritz and McIlraith, 2009] also considers probabilities analytically, in part, while calculating the robustness of plans, but still partially relies on sampling and cannot handle complex plan objective functions.

2.2 Deterministic Planning

The basic case for deterministic planning in uncertain domains is to generate a plan based on a deterministic reduction of the domain, and then replan as necessary during execution. A well-known example of this strategy is FF-Replan [Yoon et al., 2007], which determinizes probabilistic input domains, plans with the deterministic planner FF [Hoffmann and Nebel, 2001], and replans when unexpected states occur. This very simple strategy for dealing with domain uncertainty won the

Probabilistic Track of the 2004 International Planning Competition, and was also the top performer in the 2006 competition. Planning in this way, however, suffers from problems such as unrecoverable errors and spending large amounts of time replanning at execution time (even though the time spent planning may be smaller overall). Below we discuss two main bodies of work that have been developed to improve deterministic planning of uncertain domains: increasing plan flexibility and robustness to lessen the need to replan, and making each replan that does happen as fast as possible.

2.2.1 Plan Flexibility and Robustness

Whereas fixed times schedules specify only one valid start time for each task, flexible times schedules allow execution intervals for tasks to drift within a range of feasible start and end times [Smith et al., 2006]. This allows the system to adjust to unforeseen temporal events as far as constraints will allow, replanning only if one or more constraints in the current deterministic schedule cannot be satisfied. The schedules are modeled as simple temporal networks (STNs) [Dechter et al., 1991], enabling fast constraint propagation through the network as start and end times are updated after execution to generate new execution intervals for later tasks. Flexible times schedules may be total- or partial-order.

IxTeT [Laborie and Ghallab, 1995], VHPOP [Younes and Simmons, 2003] and [Vidal and Geffner, 2006] are temporal planners that generate partial-order plans. Partial-order planners repair plans at run time by searching through the partial plan space. These planners in general, however, have a much larger state space than total-order planners, and so replanning in general is more expensive. This work is orthogonal to our approach, as either total- or partial-order plans could be used as part of our analysis.

In a robustness-focused approach to partial-order planning, [Policella et al., 2007] generates robust partial-order plans by expanding a fixed times schedule to be partial-order, showing improvements with respect to fewer precedence orderings and more temporal slack over a more top-down planning approach. [Lau et al., 2007] also generates partial-order schedules that are guaranteed to have a minimum utility with probability $(1 - \epsilon)$ for scenarios where the only uncertainty is in the processing time of each task.

[Jiménez et al., 2006] look at how to create more robust deterministic plans for probabilistic planning problems by directly considering the underlying probabilistic model. They incorporate probabilistic information as an action "cost model," and by solving the resulting deterministic problems they see a decrease in the frequency of replanning. This approach has been demonstrated to outperform FF-Replan on single agent domains without durative actions, but it is so far unable to handle the types of domains we are considering. [Gallagher, 2009] makes plans more robust

by using simple schedule strengthening that relies on a limited stochastic analysis of the current schedule; in general, however, it is not as effective as our probabilistic schedule strengthening approach which relies on a probability analysis that propagates probability information throughout the entire schedule.

An approach that helps to promote a different kind of flexibility during execution allows agents to switch tasks during execution in order to expedite the execution of bottleneck tasks, minimizing schedule makespan [Sellner and Simmons, 2006]. In order to anticipate problems, the agents predict how much longer executing tasks will take to complete; in theory, this prediction could also be used as part of our probability calculations.

Alternately, a plan can be configured to be dynamically controllable [Morris et al., 2001]. A plan with this property is guaranteed to have a successful solution strategy for an uncertain temporal planning problem. This approach, as well as the above approaches in general, however, are either overly conservative or ignore the extra leverage that explicit consideration of uncertainty can provide.

2.2.2 Plan Management and Repair

A second way to optimize the deterministic planning approach is manage them during execution, identifying problems and, ideally, making replanning as fast as possible. [Pollack and Horty, 1999] describes an approach to plan management which includes commitment management, alternate plan assessment, and selective plan elaboration to manage plans during execution and provide the means to readily change plans if better alternatives are found.

Another common way to manage plans is to perform fast replanning during execution when errors are identified. This typically involves plan repair, where the planner starts with the current, conflicted plan and makes changes to it, such as by adding or removing tasks, until a new plan is found. ASPEN [Chien et al., 2000] is one such planner that makes heuristic, randomized changes to a conflicted plan to find a new, valid plan. Because of the randomness, there are no guarantees on how long it will take before a solution is found. [Gallagher et al., 2006] and [Smith et al., 2007] rely on an incremental scheduler to manage schedules during execution, and use heuristics to both maximize schedule utility and minimize disruption. Other plan repair systems are O-Plan [Tate et al., 1994], GPG [Gerevini and Serina, 2000] and [van der Krogt and de Weerdt, 2005]; however, none of these three approaches has yet been demonstrated to work with durative tasks.

HOTRiDE (Hierarchical Ordered Task Replanning in Dynamic Environments) [Ayan et al., 2007] is a Hierarchical Task Network (HTN) planning system that replans the lowest task in the task network that has a conflict whose parent does not. This helps promote plan stability, but no

data is available about the time spent replanning during execution. RepairSHOP [Warfield et al., 2007] is similar to HOTRiDE, but minimizes replanning based on "explanations" of plan conflicts. RepairSHOP results indicate that it also helps to lessen the time spent replanning during execution. The only uncertainty these two approaches address, however, is whether an executed task succeeds.

[Maheswaran and Szekely, 2008] considers a distributed, multi-agent domain with activity duration and outcome uncertainty that we use as part of this thesis work. Probability-based criticality metrics are used to repair the plan during execution according to pre-defined policies, in the spirit of our schedule strengthening. Their approach, however, is not as methodical in its probabilistic analysis nor its use of it, and is highly tailored to the domain in question.

Overall, deterministic planning systems are at a disadvantage because they ignore the potential leverage that can be provided by an explicit uncertainty model. Frequent replanning can help to overcome this downside, but the deterministic assumptions unavoidably can create problems to which replanning cannot adequately respond. Frequent replanning should also be used with care, as it has the potential to grow out of hand in extremely uncertain environments.

2.3 Meta-Level Control

Meta-level control is "the ability of complex agents operating in open environments to sequence domain and deliberative actions to optimize expected performance" [Raja and Lesser, 2007]. There is a large body of work in the area of meta-level control that addresses a variety of aspects of the problem [Cox, 2005]. Work has been performed to control the computational effort of any-time algorithms [Russell and Wefald, 1991, Hansen and Zilberstein, 2001b, Alexander et al., 2008, Boddy and Dean, 1994], to ensure that any extra time spent finding a better solution is worth the expected increase in utility of that solution. [Alexander et al., 2010] describe an any-time approach that unrolls an MDP incrementally at run time based on the trade-off of improving the policy versus executing it. [Raja and Lesser, 2007] develop a meta-level control framework for a distributed multi-agent domain that learns a meta-level MDP to decide what control actions to take at any give time. The MDP is based on a state space of abstract features of the current schedule; it would be interesting to see how the MDP would perform if the state space also incorporated in the probability profiles generated by our PADS algorithm. [Musliner et al., 2003] perform meta-level control of a real-time agent, and adaptively learn performance profiles of deliberation and the meta-level strategy.

Using a planning horizon is another way of performing meta-level control [Léauté and Williams, 2005, Chien et al., 2000, Estlin et al., 2001]. Typically, planning horizons specify how far into the

future a planner should plan. Although planning with a horizon can threaten the effectiveness of the schedule, as portions of the solution are being ignored, horizons can increase the tractability of planning (or replanning). In the above approaches, however, the horizon is set at a fixed amount of time in the future. Any-time algorithms such as [Hansen and Zilberstein, 2001a, Alexander et al., 2010] also inherently include a planning horizon as they expand an MDP's horizon while time is available; both of these approaches consider the cost of computation in their choice of a horizon.

In general, many of these approaches explicitly reason about the cost of computation, which we do not for several reasons. First, execution and replanning can occur concurrently in our domains, removing the need to explicitly choose between deliberation and action at any given time. Second, deterministic replanning tends to be fast, lessening the benefit of explicitly modeling it to use in a meta-level control algorithm. Finally, performance profiles can be difficult to model for heuristic or randomized planners like the ones we use in this document. Instead, our approach analyzes the schedule to see if replanning is likely necessary based on probabilistic information such as how likely it is that execution will succeed if a replan does not occur.

Chapter 3

Background

In this chapter, we describe core concepts and technologies upon which we build our work. We first discuss our use of plan versus schedule terminology, followed by the representation of domains and schedules that we use. Next, we discuss in detail the two domains we specifically analyze as part of this thesis. Finally, we introduce decision trees with boosting, a technique we use in our probabilistic meta-level control strategy.

3.1 Plan Versus Schedule Terminology

In this document, we use both the terms *planning* and *scheduling* freely. For the purposes of this document, we define planning as the problem of deciding what to execute, and scheduling as the problem of deciding when to execute it. By this definition, all temporal planning domains include a scheduling component; similarly, many domains typically considered scheduling domains include at least a limited planning component.

This leads us to the choice of planning/scheduling in our title and in the name of our algorithms. We name our algorithms for the general case, where there is an overall plan with a schedule that is executed to realize it. Our title, "Probabilistic Plan Management," reflects the idea that we are managing plans to satisfy goals at the highest level. Two of our algorithms, "Probabilistic Analysis of Deterministic Schedules" and "Probabilistic Schedule Strengthening" work to analyze plans at the schedule level, so we use the word "schedule" to describe them.

Our terminology, however, should not be thought to limit the scope of our approach. The ideas in this document could very well be tailored to probabilistic schedule management, a probabilistic

analysis of deterministic plans, or plan strengthening, depending on the needs of the domain.

3.2 Domain and Schedule Representation

3.2.1 Activity Networks

We present much of the description of our work in terms of activity networks, of which Hierarchical Task Networks (HTNs) are a well-known example [Erol et al., 1994]. Activity networks represent and organize domain knowledge by expressing dependencies among executable actions as a hierarchical network (or tree) of activities. At the lowest level of the tree are the primitive, executable actions, which are called *methods*. The methods are collected under aggregate nodes, or *tasks*. Tasks can be children of other tasks, and ultimately all *activities* (a term that refers to either methods or tasks) are collected under the root node of the network, called the *taskgroup*. The successful satisfaction of the objective of the root node corresponds to the goal of the domain.

Constraints are used to describe dependency relationships between any two activities in the network (although typically there are no explicit dependency constraints between activities with direct ancestry relationships). The most common constraint type is a temporal constraint between activity execution time points, which we refer to as a *prerequisite relationship*; for example, a prerequisite relationship from activity A to activity B means that the *source* A must come before the *target* B. As part of our analysis of one of our domains we also consider other types of constraints such as *facilitates*, an optional prerequisite that, if observed, magnifies the utility of an action.

Activities can also have constraints such as release (earliest possible start) times and deadlines. These types of constraints are passed down such that child activities inherit constraints from their parents, and an activity's effective execution window is defined by the tightest of these boundaries. Sibling activities can be executed concurrently in the absence of any constraints that require otherwise.

In our activity networks, each task has a type: AND, OR, or ONE. An AND task must have all children underneath it succeed by meeting their objective in order for its own objective to be satisfied. An OR task must have at least one child underneath it execute successfully, and a ONE task can have only one child underneath it successfully execute. Typically, a good schedule would include all children under an AND task, one or more under an OR task, and only one child under a ONE task; otherwise, the schedule is at risk of losing utility by not observing the task types.

In this work, we assume all domains can be represented by activity networks. The assumption that such domain knowledge is available is common for many real-world domains [Wilkins and

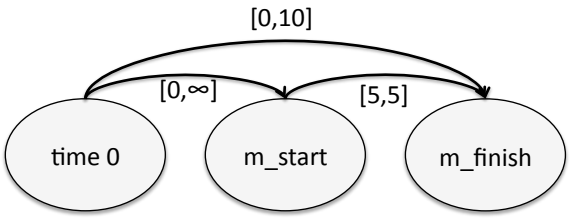


Figure 3.1: A sample STN for a single scheduled method m with duration 5 and a deadline of 10. Notice that the method is constrained to start no earlier than time 0 (via the constraint between $time\ 0$ and m_start) and finish by time 10 (via the constraint between $time\ 0$ and m_finish).

desJardins, 2000]. A simple example of an activity network representation for a domain where a set of activities must be executed is a network where all methods are children of an AND taskgroup.

3.2.2 Flexible Times Schedules

We assume that schedules are represented as deterministic, flexible times schedules as in [Smith et al., 2007]. This type of schedule representation allows the execution intervals of activities to float within a range of feasible start and finish times, so that slack can be explicitly used as a hedge against temporal uncertainty. As the underlying representation we use Simple Temporal Networks (STNs) [Dechter et al., 1991]. An STN is a graph where nodes represent event time points, such as method start and finish times, and edges specify lower and upper bounds (*i.e.*, $[lb, ub]$) on temporal distances between pairs of time points. A special time point represents time 0, grounding the network. All constraints and relationships between activities, such as deadlines, durations, ancestry relationships and prerequisites, are represented as edges in the graph. For example, a deadline dl is represented as an edge with bounds $[0, dl]$ between time 0 and the finish time of the constrained method, and prerequisites may be represented as edges of $[0, \infty]$ between the finish time of the first method and the start of the second. Method durations are represented as an edge of $[d, d']$ between a method's start and finish time. In the deterministic systems we consider, method durations are $[d, d]$, where d is the method's deterministic, scheduled duration. An agent's schedule is represented by posting precedence constraints between the start and end points of each neighboring pair of methods, termed *predecessors* and *successors*. The STN for a single scheduled method with duration 5 whose execution must finish before its deadline at time 10 is shown in Figure 3.1.

Constraints are propagated in order to maintain lower and upper bounds on all time points in the STN. Specifically, the constraint network provides an earliest start time `est`, latest start time

`lst`, earliest finish time `eft` and latest finish time `lft` for each activity. Conflicts occur in the STN whenever negative cycles are present, which indicates that some time bound is violated.

Even if a schedule is not generated by a flexible times planner, and is fixed times, it can easily be translated into a flexible times schedule, similar to [Policella et al., 2009]. The process begins by representing the schedule as described above, preserving the sequencing of the original schedule. To generate initial time bounds, each node can be initialized with all possible times (*e.g.*, times in the range $[0, h]$ where h is a time point known to be after all deadlines and when execution will have halted), and constraint propagation used to generate the valid execution intervals of each activity.

3.3 Domains

We have investigated our approach in two domains: a single-agent domain and a multi-agent one. The objective in the single-agent domain is to execute a set of tasks by a final deadline, where there is uncertainty in method durations. The domain is based on a well-studied problem with temporal constraints and little schedule slack, and its success is greatly affected by the temporal uncertainty. These characteristics make it an ideal domain for which to demonstrate the fundamental principles of our probabilistic analysis.

The second domain is a multi-agent domain that has a plan objective of maximizing utility. Utility is explicitly modeled as part of the domain description. There is uncertainty in method duration and outcome, as well as a rich set of constraints between activities. Using this large, complex domain demonstrates the generality and scalability of our algorithms. The next subsections describe these domains in detail.

3.3.1 Single-Agent Domain

We first demonstrate our work on a single-agent planning problem, which is concerned with executing a set of tasks by a final deadline. Minimal and maximal time lags specify temporal constraints between activity start and finish time points. All tasks must be executed, observing all temporal domain constraints, in order for the plan to succeed, and tasks do not have explicitly utility. Each task has a variable number of children methods, only one of which can be executed to fulfill that task's role in the plan. The agent can execute at most one method at a time.

Our domain is based on the Multi-Mode Resource Constrained Project Scheduling Problem with Minimal and Maximal Time Lags (MRCPSP/max) [Schwindt, 1998]. This NP-complete makespan minimization problem is well-studied in the scheduling community. Although the do-

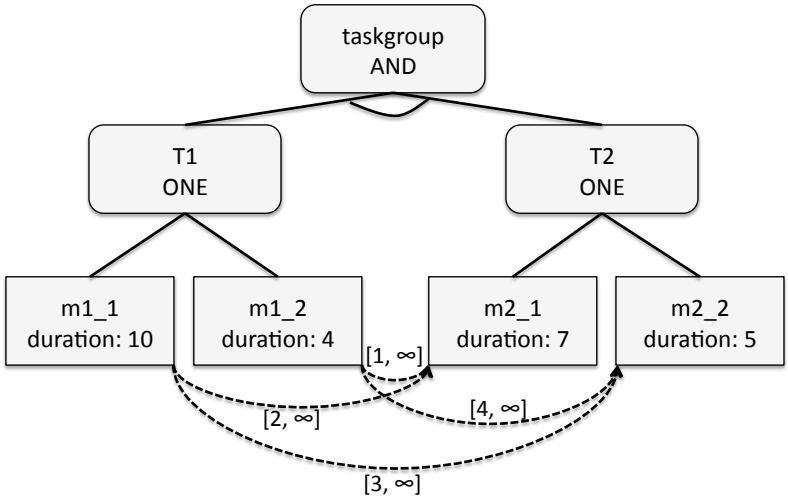


Figure 3.2: A sample MRCPSP/max problem.

main description allows a variable number of tasks and methods, each problem we use in this thesis has 20 tasks, and tasks have 2, 3 or 5 methods underneath them, each with potentially different durations. Activity dependency constraints are in the form of time lags between the finish times of the children of a source task and the start times of the children of a target task. Although all children of a source task are constrained with all children of a target task, the specifics of the constraints can vary. These constraints act as prerequisite constraints that have an associated minimal and/or maximal time delay between the execution time points of the two methods involved. The time lags specified in the problem description can also be negative; intuitively, this reverses the prerequisite constraint. The domain can be represented as an activity network with the AND taskgroup decomposing into each of the 20 tasks of the problem; those tasks, represented as ONE nodes, further decompose into their child methods.

A sample problem (with only 2 tasks) is shown in Figure 3.2. Underneath the parent taskgroup are the two tasks, each with two children. Constraints are in the form of prerequisite constraints between the end time of the source task and the start time of the target task. Note how all of T1's children are constrained with each of T2's children, but with different time lags.

The problems specify a duration $\texttt{dur}_m$ for each method m, which is the method's deterministic, scheduled duration. As the domain itself does not include uncertainty, we modified the problems by assigning each method a normal duration distribution $\texttt{DUR}\,(m)$ with a standard deviation $\sigma\,(\texttt{DUR}\,(m))$ that is uniformly sampled from the range $[1, 2 \cdot \texttt{dur}_m/5]$, and a mean $\mu\,(\texttt{DUR}\,(m))$ of $(\texttt{dur}_m - \sigma\,(\texttt{DUR}\,(m))\,/2)$. These values were chosen because, after experimenting with sev-

eral values, they seemed to present the best compromise between schedule robustness (*i.e.*, more slack built into the schedule) and a minimal schedule makespan (*i.e.*, no slack build into the schedule). When sampling from duration distributions to find methods' actual durations, durations are rounded to the nearest integer in order to simplify the simulation of execution. Finally, we assign each problem a final deadline, since the original problem formulations do not have explicit deadlines represented in it. The addition of deadlines adds to the temporal complexity of the domain.

Core Planner

We use ASPEN (Automated Scheduling/Planning ENvironment) [Chien et al., 2000] as the planner for this domain. ASPEN is an iterative-repair planner that produces fixed times plans and is designed to support a wide range of planning and scheduling domains. To generate a valid plan, ASPEN first generates an initial, flawed plan, then iteratively and randomly repairs plan conflicts until the plan is valid. It will continue to attempt to generate a plan until a solution is found or the number of allowed repair iterations is reached, in which case it returns with no solution. When replanning during execution, the planner begins with the current, potentially conflicted plan and performs the same repair process until the plan is valid or until it times out. The schedule is generated using $\texttt{dur}_m$ for each method m's deterministic duration. We used ASPEN instead of a flexible times planner because it was what we had available to use, at the time. A sample problem for this domain in ASPEN syntax is shown in Appendix A.1. In order to fit the problem on a reasonable number of pages, the problem was simplified and some constraints between methods were removed.

After planning (or replanning), we expand the plan output by ASPEN into a total-order, flexible times representation, as described in Section 3.2.2. As our constraint propagation algorithm for the STN, we implemented a version of a standard constraint propagation algorithm AC3 [Machworth, 1977] that propagates constraints both forward and backward. The algorithm begins with all edges marked as pending, meaning they are yet to be checked. It examines each edge and removes possible values from the two connected nodes if they are not consistent with the constraint asserted by the edge. If any node has a value removed, all edges except for the current one are added back onto the pending list. The algorithm terminates when the pending list is empty.

During execution, the executive monitors the prerequisite conditions of any pending activities, and executes the next activity as soon as its prerequisites are satisfied. The agent replans whenever the STN becomes inconsistent. We will present the run-time plan management approach for this domain later, in Section 7.2.1.

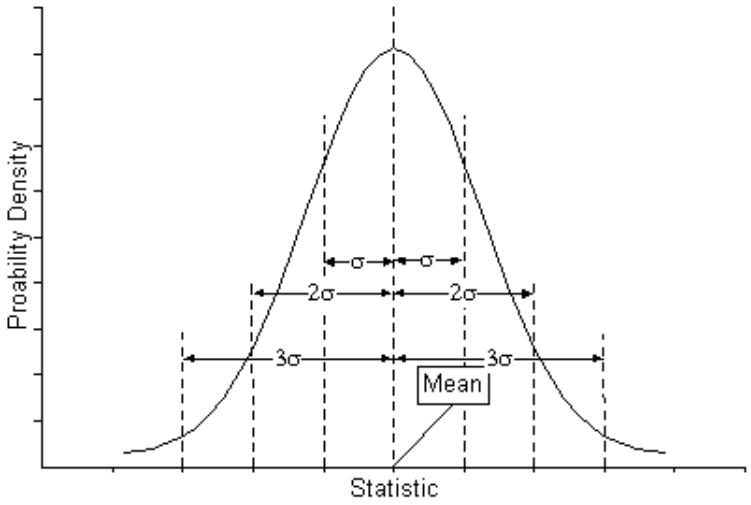


Figure 3.3: A graph of a normal distribution.

Normal Distributions

Task duration distributions are modeled in this domain as normal distributions. Normal, or Gaussian, distributions are continuous probability distributions typically used to describe data that is clustered around a mean, or average, μ (Figure 3.3). A distribution's standard deviation (σ) describes how tightly the distribution is clustered around the mean; its variance σ^2 is the square of the standard deviation. To indicate that a real-valued random variable X is normally distributed with mean μ and variance σ^2, we write $X \sim N\left(\mu, \sigma^2\right)$. Normal distributions have the nice property that the mean of the sum of two normal distributions equals the sum of the means of each of the distributions; the same is true for variance. Thus, the normal distribution exactly representing the distribution of the sum of two random variables drawn from $\sim N\left(\mu_1, \sigma_1^2\right)$ and $\sim N\left(\mu_2, \sigma_2^2\right)$ is $\sim N\left(\mu_1 + \mu_2, \sigma_1^2 + \sigma_2^2\right)$. This property makes normal distributions very easy to work with in our analysis of this domain, which requires summing distributions over groups of tasks, as the sum of n distributions can be done in $O(n)$ time. For other functions that some domains may require (*e.g.*, maximum, minimum, etc.), however, normals do not combine as nicely and an approximation or discretization may be appropriate. Of course, if a domain's distribution type is not represented as a normal, these calculations can be changed as appropriate.

We use two functions as part of our use of normal distributions. The first is `normcdf` (c, μ, σ), the normal cumulative distribution function. This function calculates the probability that a variable drawn from the distribution will have a value less than or equal to c. As an example, the value of

$\texttt{normcdf}\,(\mu, \mu, \sigma)$ is 0.5, as 50% of the distribution's probability is less than the mean.

As part of its analysis, PADS needs to calculate the distribution of a method's finish time, given that it meets a deadline. This requires truncating distributions and combining them, such as by addition, with other normal distributions. There is no known way to represent exactly the combination of a truncated normal distribution and a normal distribution; the best approximation we are aware of finds the mean and standard deviation of the truncated distribution and treats it as the mean and standard deviation of a normal distribution. We define this function as $\mu_{trunc}, \sigma_{trunc} = \texttt{truncate}\,(lb, ub, \mu, \sigma)$, meaning that the normal distribution $\sim N(\mu, \sigma^2)$ is being truncated below and above at $[lb, ub]$. To find the mean and standard deviation of a truncated distribution, we base our calculations on [Barr and Sherrill, 1999], which specifies that the mean of a standard normal random variable $\sim N(0, 1)$ truncated below at a fixed point c is

$$\mu_{trunc} = \frac{e^{-c^2/2}}{\sqrt{2\pi}\,(1 - \texttt{normcdf}\,(c, 0, 1))}$$

and the variance[1] is

$$\sigma^2_{trunc} = \frac{\left(1 - \texttt{chi-square-cdf}\left(c^2, 3\right)\right)}{2 \cdot (1 - \texttt{normcdf}\,(c, 0, 1))} - \mu^2_{trunc}$$

From this we derive equations that truncate both above and below at $[lb, ub]$:

$$\mu_{trunc} = \frac{-e^{-ub^2/2} + e^{-lb^2/2}}{\sqrt{2\pi}\,(\texttt{normcdf}\,(ub, 0, 1) - \texttt{normcdf}\,(lb, 0, 1))}$$

$$\sigma^2_{trunc} = \frac{\left(\texttt{chi-square-cdf}\left(ub^2, 3\right) - \texttt{chi-square-cdf}\left(lb^2, 3\right)\right)}{2 \cdot (\texttt{normcdf}\,(ub, 0, 1) - \texttt{normcdf}\,(lb, 0, 1))} - \mu^2_{trunc}$$

These equations are the basis of the function `truncate`, which we use in our probabilistic analysis of this domain.

3.3.2 Multi-Agent Domain

Our second domain is an oversubscribed, multi-agent planning problem that is concerned with the collaborative execution of a joint mission by a team of agents in a highly dynamic environment. Missions are formulated as a hierarchical network of activities in a version of the *TAEMS* language

[1] The formula used in this calculation, $\texttt{chi-square-cdf}\,(x, N)$ is the chi-squared cumulative distribution function and returns the probability that a single observation from the chi-squared distribution with N degrees of freedom will fall in the interval $[0, x]$.

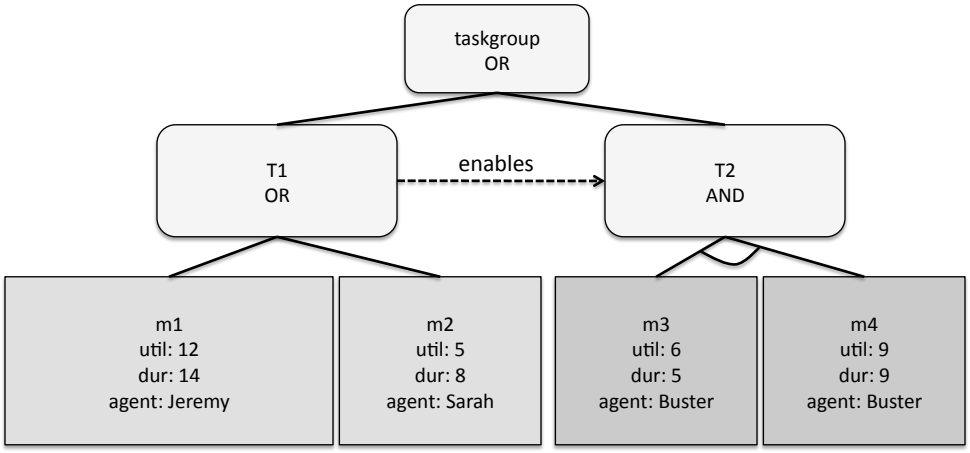


Figure 3.4: A simple, deterministic 3-agent problem.

(*T*ask *A*nalysis, *E*nvironment *M*odeling and *S*imulation) [Decker, 1996] called *C_TAEMS* [Boddy et al., 2005]. The plan objective is to maximize earned utility, which is explicitly modeled in this domain[2]. The taskgroup node of the network, which is an OR node, is the overall mission task, and agents cooperate to accrue as much utility at this level as possible. On successive levels, interior nodes constitute aggregate activities that can be decomposed into sets of tasks and/or methods (leaf nodes), which are directly executable in the world. Other than this, there is no pre-determined structure to the problems of this domain. The domain's activity network also includes special types of constraints and relationships between activities that warrant special consideration while probabilistically analyzing plans of this domain, as described further below.

To illustrate the basic principles of this domain, consider the simple activity network shown in Figure 3.4. At the root is the OR taskgroup, with two child tasks, one OR and one AND; each child task has two methods underneath it. Each method can be executed only by a specified agent (*agent: Name*) and each agent can execute at most one method at a time. The *enables* relationship shown is a prerequisite relationship.

Given this activity network, one could imagine a number of schedules being generated. We list below all possible valid schedules, as well as situations in which they would be appropriate.

- **m1** - Jeremy executes method m1, but there is not enough room in Buster's schedule for m3

[2]In the terminology of the domain and our previous publications of PPM for this domain [Hiatt et al., 2008, 2009], utility is called "quality"; for consistency in this document, we will continue to refer to it as "utility."

and m4.

- **m2** - Jeremy does not have enough room in its schedule to execute method m1, so Sarah executes the lower-utility option of m2.
- **m1**$\rightarrow \{$**m3, m4**$\}$ - Jeremy executes method m1, which fulfills the prerequisite constraint and allows Buster to execute m3 and m4, which can be executed in any order.
- **m2** $\rightarrow \{$**m3, m4**$\}$ - Jeremy does not have enough room in its schedule to execute method m1, so Sarah executes the lower-utility option of m2; this fulfills the prerequisite constraint and allows Buster to execute m3 and m4, which can be executed in any order.
- $\{$**m1 or m2**$\} \rightarrow \{$**m3, m4**$\}$ - Jeremy executes method m1, and Sarah concurrently executes method m2. Whichever finishes first fulfills the prerequisite constraint and allows Buster to execute m3 and m4, which can be executed in any order. An example of when this schedule is appropriate is if Buster's schedule is tight and it is useful for m3 and m4 to begin execution before m1 is able to finish, calling for the scheduling of m2; since m1 earns higher utility, however, it is also scheduled.

Note that the reason why methods m3 and m4 are not executed concurrently in this example is that they are owned by the same agent.

Now that we have illustrated the basic principles of the domain, we show the full model of this example problem in Figure 3.5, and refer to it as we describe the further characteristics of the domain.

Method durations are specified as discrete probability distributions over times (*DUR: X*), with the exact duration of a method known only after it has been executed. Method utilities, (*UTIL: X*), are also specified as discrete probability distributions over utilities. Methods do not earn utility until their execution has completed. Methods may also have an *a priori* likelihood of ending with a failure outcome (*fail: X%*), which results in zero utility. In subsequent text, we refer to `DUR`, `UTIL` and `fail` as functions, which when given a method as input return the duration distribution, utility distribution and failure likelihood, respectively, of that method. We also define the functions `averageDur` and `averageUtil`, which return the deterministic expectation of an activity's duration and utility, respectively. We use "average" here to distinguish these values from the expected utility which is calculated as part of the probabilistic analysis. Methods can fail for reasons other than a failure outcome, such as by missing a deadline; however, any type of failure results in the method earning zero utility. Even if a method fails, execution of the schedule still continues, since the complicated activity structure and plan objective means that there is very likely still potential for earning further utility at the taskgroup level.

Each task in the activity network has a specified *utility accumulation function* (*uaf*) defining

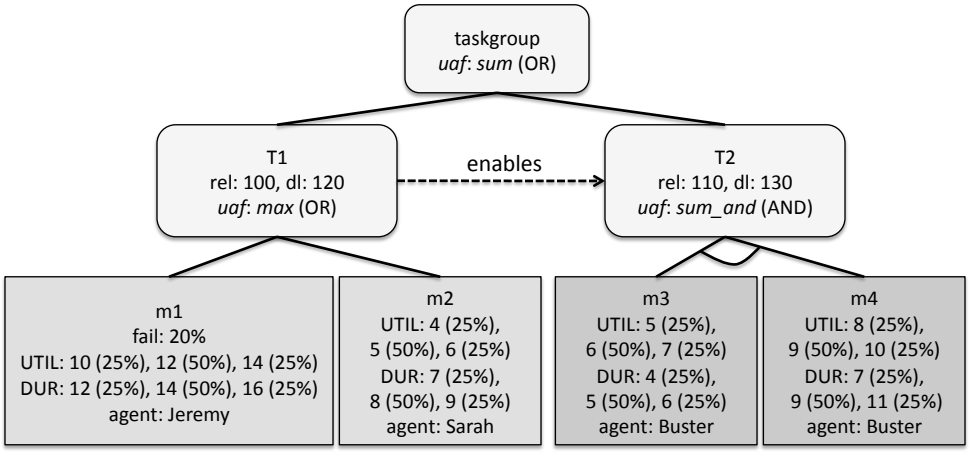


Figure 3.5: A small 3-agent problem.

when, and how, a task accumulates utility. The *uafs* are grouped by node type: OR, AND and ONE. Tasks with OR *uafs* earn positive utility as soon as their first child executes with positive utility. Tasks with AND *uafs* earn positive utility once *all* children execute with positive utility. Trivially, ONE tasks earn utility when their sole child executes with positive utility. *Uafs* also specify how much final utility the tasks earn. For example, a *sum* task's utility at any point equals the total earned utility of all children. Other *uafs* are *max*, whose utility equals the maximum utility of its executed children, and *sum_and*, which earns the sum of the utilities of its children once they all execute with positive utility. A less common *uaf* is *min*, which specifies that tasks accumulate the minimum of the utilities of their children once they all execute with positive utility. A summary of the *uaf* types is shown in Table 3.1.

Additional constraints in the network include a release (earliest possible start) time and a deadline (denoted by *rel: X* and *dl: X*, respectively, in Figure 3.5) that can be specified for any activity. 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 fails if any of these constraints are violated. In the rest of this document, we refer to `rel` and `dl` as functions, which when given an activity as input return its release and deadline, respectively.

Inter-dependencies between activities are modeled via constraints called non-local effects (NLEs). NLEs express causal effects between a *source* activity and a *target* activity. NLEs can be between any two activities in the network that do not have a direct ancestry relationship. If the target activity is a task, the NLE is interpreted as applying individually to each of the methods in the activity net-

Table 3.1: A summary of utility accumulation function (*uaf*) types.

uaf	**type**	**description**
sum	OR	utility is sum of children's utility (task earns utility as soon as the first child earns positive utility)
max	OR	utility is max of children's utility (task earns utility as soon as the first child earns positive utility)
sum_and	AND	utility is sum of children's utility (task earns utility once all children earn positive utility)
min	AND	utility is min of children's utility (task earns utility once all children earn positive utility)
exactly_one	ONE	utility is only executed child's utility (task earns utility as soon as the first child earns positive utility, task has zero utility if any other children earn positive utility)

work under the task. Therefore, in much of our future discussion, we treat NLEs where the target is a task as the analogous situation where there are enabling NLEs from the source to each of the task's method descendants.

NLEs are activated by the source activity accruing positive utility, and must be activated before a target method begins execution in order to have an effect on it. All NLEs have an associated delay parameter specifying how much time passes from when the source activity accrues positive utility until the NLE is activated; although our approach can handle any arbitrary delay, for the sake of simplicity we present our discussion as if delays do not exist.

There are four types of NLEs. The most common is the *enables* relationship. The enables NLE stipulates that unless the target method begins execution after the source activity earns positive utility, it will fail. For example, in Figure 3.5, the execution of targets m_3 and m_4 will not result in positive utility unless the methods are executed after the source T_1 earns positive utility. The enables relationship is unique among the NLEs because an effective agent will wait for all of a method's enablers to earn positive utility before it attempts to begin executing it; otherwise, its execution would be a waste as the method would not earn utility.

The second NLE is the *disables* NLE, which has the opposite effect of enabling. The disables relationship stipulates that the target method fails (earns zero utility) if it begins executing *after* its source has accrued positive utility. Typically, an effective planner will schedule sources of disables to finish after the targets start. If a source activity does finish, however, before the target has started, executing the target serves no purpose and so the planner should remove it from the schedule. The

disables NLE and enables NLE are referred to as "hard" NLEs, as they must be observed to earn utility.

The third NLE is the *facilitates* NLE. This NLE, when activated, increases the target method's utility and decreases its executed duration. The amount of the effect depends on the *proportional utility* ($prop\text{-}util$) of the source s at the time e when the target starts execution:

$$prop\text{-}util\,(s,e) = \texttt{min}\,(1,\ util\text{-}earned\,(s,e))\,/MaxU\,(s)$$

In this formula, $util\text{-}earned$ refers to the utility so far accumulated by s, and $MaxU\,(s)$ refers to an approximation of the maximum possible utility of the activity s. As it is not important to our approach, we omit describing how $MaxU$ is calculated.

This formula is used to specify the effect of the NLE in terms of a utility coefficient uc such that $0 \leq uc \leq 1$, and a duration coefficient dc such that $0 \leq dc < 1$. Putting it all together, the utility distribution of the target t if it starts at time e becomes:

$$\texttt{UTIL}\,(t) = \texttt{UTIL}\,(t) \cdot (1 + uc \cdot prop\text{-}util\,(s,e))$$

The duration distribution, similarly, becomes:

$$\texttt{DUR}\,(t) = \lceil \texttt{DUR}\,(t) \cdot (1 - dc \cdot prop\text{-}util\,(s,e)) \rceil$$

Note that the presence of $\lceil\ \rceil$ in the duration ensures that durations are integral; utilities may not be after a facilitating effect. Also note that according to this feature of the domain description, a target's utility can be, at most, doubled by the effect of a single facilitator.

The *hinders* NLE is exactly the same as the facilitates, except that the signs of the effect are changed:

$$\texttt{UTIL}\,(t) = \texttt{UTIL}\,(t) \cdot (1 - uc \cdot prop\text{-}util\,(s))$$

$$\texttt{DUR}\,(t) = \lceil \texttt{DUR}\,(t) \cdot (1 + dc \cdot prop\text{-}util\,(s)) \rceil$$

Again, note that durations are always integral and, according to this feature of the domain description, a target's duration can, at most, be doubled by the effect of a single hinderer.

If a method is the target of multiple facilitates or hinders, the effects can be applied in any order, as multiplication is communicative; for the duration calculation, the ceiling function is applied after the individual NLE effects have been combined. The facilitates and hinders NLEs are referred to as "soft" NLEs as, unlike the hard NLEs, utility can be earned without observing them. A summary of the NLE types is shown in Table 3.2.

In the rest of this document, we refer to `en`, `dis`, `fac` and `hind` as functions that, when given an activity as input, return the scheduled enablers, disablers, facilitators and hinderers of

Table 3.2: A summary of non-local effect (NLE) types.

NLE	hard/soft	description
enables	hard	target will fail if it begins executing before the source earns positive utility
disables	hard	target will fail if the source earns positive utility before the target begins executing
facilitates	soft	target's duration/utility are decreased/increased if the source earns positive utility before the target begins executing
hinders	soft	target's duration/utility are increased/decreased if the source earns positive utility before the target begins executing

the activity, as appropriate. The function `enabled` and `disabled` return the activities directly enabled (or disabled) by the input activity. `NLEs` returns the scheduled source NLEs of all types of an activity. We will also refer later to the function `directlyDisabledAncestor` which, when given a method and (one of) its disabler, returns the ancestor of the method that is directly targeted by the disables NLE. A sample problem for this domain in *C_TAEMS* syntax is shown in Appendix A.2. In order to fit the problem on a reasonable number of pages, the problem was simplified and some activities were deleted.

We assume that a planner generates a complete flexible times schedule for all the agents. When execution begins, agents receive a copy of their portion of the initial schedule and limited local views of the activity network. Agents' local views consist of activities that are designated as *local* (the agent is responsible for executing and/or managing them) or *remote* (the agent tracks current information for these activities via status updates from the agent responsible for maintaining them). Agents receive copies of all of their local activities, as well as remote copies of all activities that are constrained via NLEs with their local activities. Additionally, all activities are ultimately connected to the root node (*i.e.*, the local view consists of a single tree rooted at the taskgroup node). The limited local view for agent Buster of Figure 3.5 is shown in Figure 3.6. After execution begins, agents have a short, fixed period of time to perform initial computations (such as calculating initial probability information) before they actually begin to execute methods.

We have developed two versions of our probabilistic plan management approach for this domain. One analysis applies to the initial, global schedule and is used to strengthen it before it is distributed to agents. The second analysis is used by each agent during execution to distributively manage their portion of the schedule. During our discussion of PADS, we distinguish between the functions `jointSchedule`, which returns the scheduled methods across all agents, and `localSchedule`, which returns the methods scheduled locally on a single agent. Similarly,

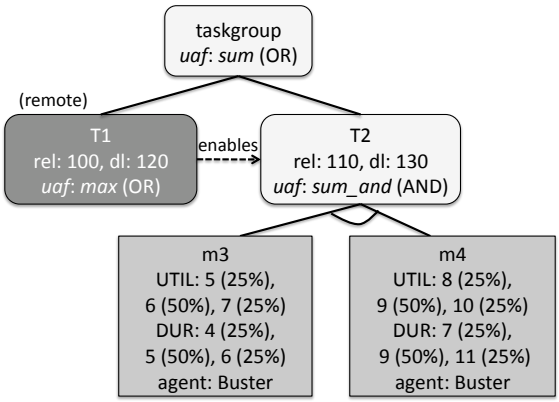


Figure 3.6: The limited local view for agent Buster of Figure 3.5.

the `local` function, when given an activity, returns whether that activity is local to the agent.

Core Planner

The planner for this domain is the incremental flexible times scheduling framework described in [Smith et al., 2007]. It adopts a partial-order schedule (*POS*) representation, with an underlying implementation of an STN, as described in Section 3.2.2. The constraint propagation algorithm used an incremental, arc-consistency based shortest path algorithm, based on [Cesta and Oddi, 1996]. The schedule for any individual agent is total-order; however, the joint schedule as a whole may be partial-order as inter-agent activities are constrained with each other only by NLEs, and not by explicit sequencing constraints.

The planner uses expected duration and utility values for methods, ignores *a priori* failure likelihoods, and focuses on producing a *POS* for the set of agents that maximizes overall problem utility. Given the complexity of solving this problem optimally and an interest in scalability, the planner uses a heuristic approach. Specifically, the initial, global 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 utility if there were no capacity constraints. In the second phase, an attempt is made to sequentially allocate these methods to agent schedules. 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 phase is re-invoked with the problematic method excluded. A similar process takes place within each agent when it replans locally. More

detail on the planner can be found in [Smith et al., 2007].

During distributed execution, the executor monitors the NLEs of any pending activities and executes the next pending activity as soon as all the necessary constraints are satisfied. Agents replan selectively in response to execution, such as if the STN becomes inconsistent, or if a method ends in a failure outcome. We describe the run-time plan management system further in Section 7.3.1.

Discrete Probability Distributions

In this domain, uncertainty is represented as discrete distributions. We use summation, multiplication, maximization and minimization in our algorithms in later chapters; if a domain's distribution type is different than discrete, the below equations can be changed as appropriate.

The sum of two discrete probability distributions, $Z = X + Y$ is [Grinstead and Snell, 1997]:

$$P(Z = z) = \sum_{k=-\infty}^{\infty} P(X = k) \cdot P(Y = z - k)$$

The max of two discrete probability distributions, $Z = \texttt{max}(X, Y)$ is:

$$P(Z = z) = P(X = z) \cdot P(Y < z) + P(X < z) \cdot P(Y = z)$$

For the sake of redundancy, we omit the versions of these formulas for the multiplication of and the minimum of two distributions. These equations can be implemented by iterating through the distributions in question. Unlike with normal distributions, therefore, these calculations are not linear and are done in $O(|X| \cdot |Y|)$ time; the size of the resulting distribution is bounded by $|X| \cdot |Y|$ as well (although typically such a distribution has duplicate values and so its representation is not that large). Discrete distributions do, however, have the benefit of being able to be combined or modified exactly by any arbitrary operator.

3.4 Decision Trees with Boosting

We use decision trees with boosting [Quinlan, 1986] as part of our work on probabilistic meta-level control. Decision trees, and classifiers in general, are a way of assigning one of a number of classes to a given state. For our purposes, this means determining what high-level planning action to take based on the current probabilistic analysis of the schedule. Decision trees, however, are easily susceptible to *overfitting*: they can become unnecessarily complex and describe random

error or noise in the dataset instead of the true underlying model. Boosting is one technique that is commonly used to overcome this problem. Boosting is a meta-algorithm that creates many small, weak classifiers (such as decision trees restricted in size), weighs them according to their goodness, and lets them "vote" to determine the overall classification. We choose this type of classification because it is easy to implement and very effective; [Breiman, 1996] even called AdaBoost with decision trees the "best off-the-shelf classifier in the world."

To construct a decision tree, one needs a *training set* of example states, $x_1...x_j$, each with a known class $Y \in y_1...y_k$. Each example consists of a number of *attributes*; in our case, these attributes are variables that have continuous values. Because we are using a boosting algorithm, the examples also have associated weights, $w_1...w_j$. A decision tree is built recursively. It starts at the root node and inspects the data. If all examples have the same class, the node becomes a leaf node specifying that class. If all examples have identical states, it becomes a leaf node specifying the class that the weighted majority of the states have. Otherwise, the node branches based on an attribute and a threshold for that attribute: all examples with a value for that attribute less than the threshold are passed to the right child, all others are passed to the left. This process recurses for each child, continuing until the algorithm terminates. An example decision tree where all examples are weighted equally is shown in Figure 3.7.

To choose which attribute to branch on, we use the algorithm C4.5 [Quinlan, 1993]. At each node of the tree C4.5 chooses an attribute A and, if A is a continuous attribute, a threshold t, which maximize *Information Gain* (IG), or the reduction in entropy that occurs by splitting the data by A and t. Formally, IG is defined as

$$IG\left(A,t\right) = H\left(Y\right) - H\left(Y \mid A,t\right)$$

where $H\left(Y\right)$ represents the entropy of the class variable Y before the split:

$$H\left(Y\right) = -\sum_{i=1}^{k} P\left(Y = y_i\right) \log_2 P\left(Y = y_i\right)$$

and $H\left(Y \mid A,t\right)$ represents the entropy of the class variable Y after the split, which divides the data according to whether the value for attribute A is less than t:

$$\begin{aligned} H\left(Y \mid A,t\right) = &- P\left(A < t\right) \sum_{i=1}^{k} P\left(Y = y_i \mid A < t\right) \log_2 P\left(Y = y_i \mid A < t\right) \\ &- P\left(A \geq t\right) \sum_{i=1}^{k} P\left(Y = y_i \mid A \geq t\right) \log_2 P\left(Y = y_i \mid A \geq t\right) \end{aligned}$$

Example	Time	Rain	Traffic
1	6:00am	0.04in	Yes
2	6:15am	0.01in	No
3	8:00am	0.0in	Yes
4	8:30am	0.02in	Yes
5	9:30am	0.0in	No
6	1:30pm	0.01in	No
7	4:30pm	0.03in	Yes
8	4:45pm	0.01in	Yes
9	5:45pm	0.0in	Yes
10	7:00pm	0.0in	No
11	8:00pm	0.05in	Yes

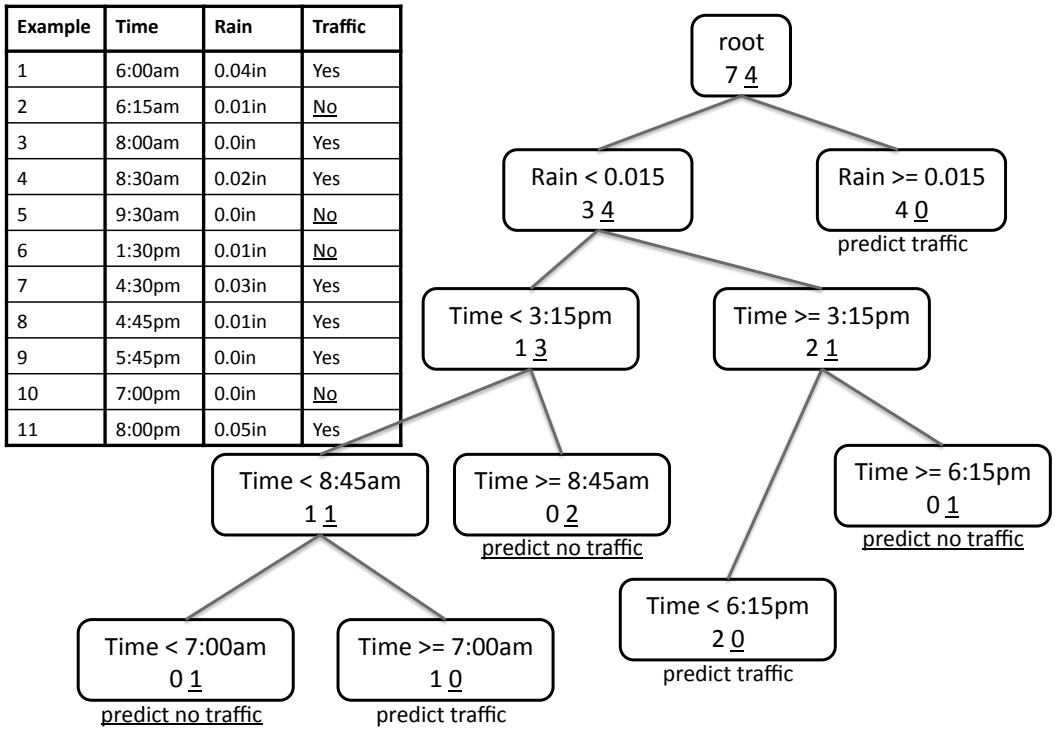


Figure 3.7: An example decision tree built from a data set. The tree classifies 11 examples giving state information about the time of day and how much rain there has been in the past 10 minutes as having or not having traffic at that time. Each node of the tree shows the attribute and threshold for that branch as well as how many instances of each class are present at that node. Instances that are not underlined refer to traffic being present; underlined instances refer to traffic not being present.

As the data is weighted, $P(Y = y_i)$ is:

$$P(Y = y_i) = \frac{\sum_{i=1}^{j} (w_i \cdot \mathbf{1}(Y(x_i) = y_i))}{\sum_{i=1}^{j} w_i}$$

where $\mathbf{1}$ is the indicator function, which returns 1 if the passed in equation is true and 0 otherwise.

Because we are solving a multi-class classification problem, we use the boosting algorithm SAMME [Zhu et al., 2005], a multi-class variant of AdaBoost [Freund and Schapire, 1997]. The algorithm begins by weighing all training examples equally. It builds a simple decision tree (*e.g.*, a decision tree with two levels), and calculates a "goodness" term α for the classifier based on the (weighted) percent of examples that were misclassified. Next, it increases the weight of the

misclassified examples and decreases the weight of the correctly classified examples, and builds another decision tree based on the newly weighted examples. It repeats the process of building decision trees and reweighing the examples until the desired number M of classifiers are built. The overall classification $C(x)$ of an example x is then:

$$C(x) = \arg\max_{y} \sum_{m=1}^{M} \alpha(m) \cdot \mathbf{1}(C^m(x) = y)$$

This equation, intuitively, means that the classifier returns the classification that receives the highest total of votes from each of the weighted decision trees.

Chapter 4

Approach

Probabilistic plan management (PPM) is a composite approach to planning and scheduling under uncertainty that couples the strengths of deterministic and non-deterministic planning, while avoiding their weaknesses. The goal is to manage and execute effective deterministic plans in a scalable way. As its starting point, PPM assumes a deterministic planner that generates an initial schedule to execute and replans as necessary during execution. It then layers a probabilistic analysis on top of the planner to explicitly analyze the uncertainty of the schedules the planner produces. PPM maintains this probabilistic analysis of the current deterministic schedule as it changes due to both the dynamics of execution and replanning. Both before and during execution, it uses the analysis to make the schedule more robust to execution uncertainty, via *probabilistic schedule strengthening*. It also uses the analysis to manage the execution of the schedule via *probabilistic meta-level control*, limiting the use of resources to replan and strengthen the schedule during execution while maintaining the effectiveness of the schedule.

In this chapter, we first discuss the types of domains that our approach can analyze. We then detail each of the three main components of probabilistic plan management, as well as discussing how they fit together to probabilistically manage plans.

4.1 Domain Suitability

In the problems we consider, plan execution has an overall, explicit objective. Objectives typically fall under one of two general categories: all activities must be executed within their constraints; or reward (or another metric such as cost) is maximized or minimized. The objective can also be a combination of these two categories. In all cases, the objective must be specified in terms of a

numeric *utility*, which quantifies how well the objective was achieved. For example, if all activities must be executed within their constraints, then one possible utility function is assigning a schedule that does so a utility of 1, and one that does not a utility of 0. Similarly, the objective of maximizing reward leads to a utility that simply equals the reward earned during execution. For multi-agent scenarios, we assume agents cooperate together to jointly achieve the plan objective. Additionally, the objective must be affected in some way by the presence of uncertainty in the problem. If it is not, then our analysis will not provide any information helpful for probabilistically managing plan execution.

Any domain that is to be considered by our approach must have an explicit model of its uncertainty. No specific representation of uncertainty is assumed, as long as the distributions representing the uncertainty can be combined, or approximately combined, according to various functions (*e.g.*, sum, max). Two common distribution types (and the two that we consider explicitly in this document) are normal and discrete distributions. Normal distributions can be very readily summed to calculate activities' finish time distributions; however, they are unable to exactly capture more complicated functions, such as the finish time of an activity given that it meets a deadline [Barr and Sherrill, 1999]. Discrete distributions, on the other hand, allow for the exact calculation of combinations of distributions; however, the size of the combination of two distributions is bounded by the product of the size of the two individual distributions.

In this work, we make the assumption that domain representations can be translated into hierarchical activity networks, which are then considered by our analysis (Section 3.2.1). Discussion of our analysis does presuppose that we are considering planning problems with durative tasks and temporal uncertainty; such problems are in general difficult to solve, and we show our approach is effective in such domains. We assume that schedules are represented as STNs. However, the temporal aspect of the analysis can easily be left out to perform PPM for domains without durative actions. Our analysis also assumes that plans do not branch; again, however, this assumption could be broken and the approach extended to include branching or conditional plans. Our approach does not place any restrictions on what planner is used or how it determinizes the domain, but ideally it generates good deterministic plans for the problem of interest. This is because we do not integrate our approach with the planner, but instead layer an uncertainty analysis on top of the schedules it produces. In particular, we do not assume the planner plans with activity networks; the planner can work with whatever domain representation is most convenient.

Although a variety of executive systems can be used in our approach, the most suitable would provide execution results as they become known throughout execution, allowing for more effective runtime probabilistic plan management. Also, for the purposes of this analysis, we assume that the executive is perfect (*e.g.*, tasks always start when requested, no latency exists, etc.). If the executive

system broke this assumption often enough, it should be explicitly modeled and incorporated into the uncertainty analysis to increase PPM's effectiveness.

For multi-agent domains, we also assume that communication is reliable and has sufficient bandwidth for agent communications; this assumption is not broken in the multi-agent domain we specifically consider, as all communication is done via message passing over a wired intranet. Ways to limit communication in domains with limited bandwidth are discussed in Sections 5.3.5 and 8.1.1. If communication is unreliable and readily goes down, it would adversely affect our approach as agents only intermittently be able to share probability information. However, the effect of communication accidently going down can be alleviated somewhat by probabilistic schedule strengthening, because agents' last schedules would have the benefit of having been strengthened. We discuss this further in Section 6.4.4.

In this thesis, we use PPM in two different domains. One is a single-agent domain where the objective is to execute all tasks within their temporal constraints. There is uncertainty in method duration, which is represented as normal distributions. The second is a distributed, oversubscribed multi-agent domain where the objective is to maximize reward. There are a variety of sources of uncertainty, such as in method durations and outcomes, which are represented as discrete distributions.

4.2 Probabilistic Analysis of Deterministic Schedules

At the core of our approach is a *P*robabilistic *A*nalysis of *D*eterministic *S*chedules (PADS). At the highest level of abstraction, the algorithm explores all known contingencies within the context of the current schedule in order to find its expected utility. It calculates, for each activity, the probability of the activity meeting its objective, as well as its expected utility; these values, in part, make up a *probability profile* for each activity. PADS culminates in finding the probability profile of the taskgroup, which summarizes the expected outcome of execution and provides the overall expected utility. For domains where the plan objective is to execute all activities, the overall expected utility is between 0 and 1; trivially, it equals the probability that the plan objective will be met by executing the schedule. For domains with an explicit notion of reward, the taskgroup expected utility equals the expected earned reward.

By explicitly calculating a schedule's expected utility, PADS allows us to distinguish between the *deterministic* and *probabilistic states* of a schedule. We consider the deterministic state to be the status of the deterministic schedule, such as the deterministically scheduled start and finish times of activities. The probabilistic state, on the other hand, tracks activities' finish time distributions,

probabilities of meeting deadlines, and expected utility. The deterministic state knows about one (or a small set of) possible execution paths, whereas the probabilistic state expands that to summarize all execution paths within the scope of the current schedule. Thus, the deterministic state and the probabilistic state have very different notions of what it means for something to be "wrong" with the schedule. In the deterministic state, something is wrong if execution does not follow that one path (or small set of paths) and an event happens at a non-scheduled time or in a non-planned way; in the probabilistic state, something is wrong if the average utility is below the targeted amount or the likelihood of not meeting the plan objective is unacceptably low.

In general, domains can have a wide variety of sources of uncertainty, which can affect the accomplishment of their objectives. Common examples are uncertainty associated with an activity's duration, utility, and outcome (*e.g.*, was the activity successful in achieving its goal or not). These sources of uncertainty propagate through the schedule, affecting the execution of other scheduled activities and, ultimately, determining how much utility is earned during execution. The overall structure of PADS analyzes this, starting with the earliest and least constrained activities and propagating probability values throughout the schedule to find the overall expected utility.

PADS can also provide useful information about the relative importance of activities via the *expected utility loss* metric. This metric measures the relative importance of an activity by quantifying how much the expected utility of the schedule would suffer if the activity failed, scaled by the activity's likelihood of failing. This metric is most useful for plan objectives that involve maximizing or minimizing some characteristic of execution; for plan objectives that involve executing all activities, the schedule's expected utility suffers equally from any activity failing, and so expected utility loss could simply equal the probability of the activity failing.

In distributed, multi-agent scenarios, PADS cannot be calculated all at once since no agent has complete knowledge or control over the entire activity hierarchy. In such situations, therefore, the PADS algorithm is also distributed, with each agent calculating as much as it can for its portion of the activity network, and communicating the information with others. Calculations such as expected utility loss, however, are approximated by agents as needed during execution, because agents cannot calculate changes in expected utility for the entire schedule with only partial views of the activity network.

PADS can be used as the basis of many potential planning actions or decisions. In this document, we discuss how it can be used to strengthen plans to increase expected utility, and how it can be used as the basis of different meta-level control algorithms that intelligently control the computational resources used to replan during execution. Other potential uses are described in Chapter 8, which discusses future work.

A novelty of our algorithm is how quickly PADS can be calculated and its results utilized to

make plan improvements, due to analytically calculating the uncertainty of a problem in the context of a given schedule. On large reference multi-agent problems with 10-25 agents, the expected utility is globally calculated in an average of 0.58 seconds, which is a very reasonable time for large, complicated multi-agent problems such as these.

4.3 Probabilistic Schedule Strengthening

While deterministic planners have the benefit of being more scalable than probabilistic planners, they make strong, and often inaccurate, assumptions on how execution will unfold. Replanning during execution can effectively counter such unexpected dynamics, but only as long as planners have sufficient time to respond to problems that occur; moreover, there are domains when replanning is always too late to respond to activity failure, as backup tasks would have needed to be set up ahead of time. Additionally, even if the agent is able to sufficiently respond to a problem, utility loss may occur due to the last minute nature of the schedule repair. Schedule strengthening is one way to proactively protect schedules against such contingencies [Gallagher, 2009]. In this thesis, we utilize the information provided by PADS to layer an additional *probabilistic schedule strengthening* step on top of the planning process, making local improvements to the schedule by anticipating problems that may arise and probabilistically strengthening them to protect schedules against them.

The philosophy behind schedule strengthening is that it preserves the existing plan structure while making it more robust. Probabilistic schedule strengthening improvements are targeted at activities that have a (high) likelihood of failing to contribute to the plan's objective and whose failure negatively impacts schedule utility the most. By minimizing the likelihood of these failures occurring, probabilistic schedule strengthening increases schedule robustness, thereby raising the expected utility of the schedule.

Because schedule strengthening works within the existing plan structure, its effectiveness can be limited by the original goodness of the schedule. If a schedule has low utility to begin with, for example, probabilistic schedule strengthening will not be that useful; if a schedule has high utility but with a high chance of not meeting its objective or earning that utility, probabilistic schedule strengthening can help to decrease the risk associated with the schedule and raise the schedule's expected utility. Probabilistic schedule strengthening improves the expected outcome of the current schedule, not the schedule itself.

The effectiveness of schedule strengthening also depends on the structure of the domain, and on knowledge of that structure, as the specific actions that can be used to strengthen these activities

depend on the domain's activity network. Domains well-suited to schedule strengthening ideally have a network structure that allows for flexibility in terms of activity redundancy, substitution, or constraints; without this flexibility, it would be difficult to find ways to make schedules more robust. We list here example strengthening actions that can be applied to a variety of domains:

1. *Adding a redundant activity* - Scheduling a redundant, back-up activity under a parent OR task. This protects against a number of contingencies, such as an activity having a failure outcome or not meeting a deadline. However, it can also decrease schedule slack, potentially increasing the risk that other activities will violate their temporal constraints.

2. *Switching scheduled activities* - Replacing an activity with a sibling under a parent OR or ONE task, with the goal of shortening activity duration or lowering *a priori* failure likelihood. This may result, however, in a lower utility activity being scheduled.

3. *Decreasing effective activity duration* - Adding an extra agent to assist with the execution of an activity to shorten its effective duration [Sellner and Simmons, 2006] or to facilitate its execution in some other way. This decreases the chance that the activity will violate a temporal constraint such as a deadline; however, adding agents to activities or facilitating them in another way may decrease schedule slack and increase the likelihood other activities miss their deadline.

4. *Removing an activity* - Unscheduling an activity. Although an unexecuted activity does not contribute to a schedule's utility, removing activities increases schedule slack, potentially increasing the likelihood that remaining activities meet their deadlines.

5. *Reordering activities* - Changing the execution order of activities. This has the potential to order activities so that constraints and deadlines are more likely to be met; however, it might also increase the likelihood that activities miss their deadlines or violate their constraints.

PADS allows us to explicitly address the trade-offs inherent in each of these actions to ensure that all modifications make improvements with respect to the domain's objective. After any given strengthening action, the expected utility of the plan is recalculated to reflect the modifications made. If expected utility increases, the strengthening action is helpful; otherwise it is not helpful and the action can be rejected and undone.

During overall probabilistic schedule strengthening, activities are prioritized for strengthening by expected utility loss so that the most critical activities are strengthened first. Strengthening actions can be prioritized by their effectiveness: in general, the most effective actions preserve the existing schedule structure (such as (1) and (3)); the least effective do not (such as (4)).

During execution, probabilistic schedule strengthening can be performed at any time to robustify the schedule as execution proceeds. Ultimately, however, it is best when coupled with replanning to build effective, robust schedules. Replanning during execution allows agents to generate schedules that take advantage of opportunities that may arise; schedule strengthening in turn makes those schedules more robust to unexpected or undesirable outcomes, increasing schedule expected utility.

On reference problems in the multi-agent domain, we have shown that probabilistically strengthened schedules are significantly more robust, and have an average of 32% more expected utility than their initial, deterministic counterparts. Using probabilistic schedule strengthening in conjunction with replanning during execution results, on average, in 14% more utility earned than replanning alone during execution.

4.4 Probabilistic Meta-Level Control

Replanning during execution provides an effective way of responding to the unexpected dynamics of execution. There are clear trade-offs, however, in replanning too much or too little during execution. On one hand, an agent could replan in response to every update; however, this could lead to lack of responsiveness and a waste of computational resources. On the other hand, an agent could limit replanning, but run the risk of not meeting its goals because errors were not corrected or opportunities were not taken advantage of.

Ideally, an agent would always and only replan (or take some other planning action, like strengthening) when it was useful; either to take advantage of opportunities that arise or to minimize problems and/or conflicts during execution. At other times, the agent would know that the schedule had enough flexibility – temporal or otherwise – to absorb unexpected execution outcomes as they arise without replanning. The question of whether replanning would be beneficial, however, is difficult to answer, in large part because it is difficult to predict whether replanning is beneficial without actually doing it.

As an alternative to explicitly answering the question, we use our probabilistic analysis to identify times where there is potential for replanning to be useful, either because there is a potential new opportunity to leverage or a potential failure to avoid. The probabilistic control algorithm thus distinguishes between times when it is *likely* necessary (or beneficial) to replan, and times when it is not. This eliminates potentially unnecessary and expensive replanning, while also ensuring that replans are done when they are probably necessary or may be useful.

There are several different ways to use PADS to drive the meta-level decision of when to replan.

For example, strategies can differ in which activity probability profiles they consider when deciding whether to replan. Which component of a probability profile is utilized (*e.g.*, expected utility, probability of meeting objective, etc.) can also vary between strategies. Additionally, different strategies can either consider activities' profiles on their own, or look at how the profiles change over time.

When designing meta-level control strategies, the objective of the plan can help lead to finding the best strategy. For planning problems with a clear upper bound on utility, such as ones in which the objective is to execute all or a set of activities within their temporal constraints, a good strategy is to replan whenever the expected utility of the entire schedule falls below a threshold. For oversubscribed problems with the goal of maximizing reward, the maximum feasible utility is typically not known, and a better strategy might be a learning-based approach that learns when to replan based on how activities' expected utilities change over time.

Whether the approach is centralized or distributed can also affect the choice of a control strategy. For centralized applications, looking at the plan as a whole may suffice for deciding when to replan. For distributed applications, however, when agents decide to replan independently of one another, the plan as a whole may be a poor basis for a control algorithm because the state of the joint plan does not necessarily directly correspond to the state of an agent's local schedule and activities. Instead, in distributed cases, it may be more appropriate to consider the local probabilistic state, such as the probability profiles of methods in an agent's local schedule, when deciding whether to replan.

The control strategy can also be used to limit not only the number of replans, but also the amount of the current schedule that is replanned. In situations with high uncertainty, always replanning the entire schedule means that tasks far removed from what is currently being executed are replanned again and again as the near term causes conflicts, potentially causing a waste of computational effort. To this end, we introduce an *uncertainty horizon*, which uses PADS to distinguish between portions of the schedule that are likely to need to be replanned at the current time, and portions that are not. This distinction is made based on the probability that activities beyond the uncertainty horizon will need to be replanned before executing them.

Experiments in simulation for the single-agent domain have shown a 23% reduction in schedule execution time (where execution time is defined as the schedule makespan plus the time spent replanning during execution) when probabilistic meta-level control is used to limit the amount of replanning and to impose an uncertainty horizon, versus when the entire plan is replanned whenever a schedule conflict arises. Experiments in the multi-agent domain show a 49% decrease in plan management time when our probabilistic meta-level control strategy is used, as compared to a plan management strategy that replans according to deterministic heuristics, with no significant change

in utility earned.

In PPM, all these components come together to probabilistically and effectively manage plan execution. A flow diagram of PPM is shown in Figure 4.1. At all times, PADS calculates and updates probabilistic information about the current schedule as execution progresses. The default state of an agent is a "wait" state, where it waits to hear of changed information, either from the simulator specifying the results of execution or, in multi-agent scenarios, messages from other agents specifying changes to their probabilistic information. Any time such an update is received, the agent updates its probability profiles via PADS (described in Chapter 5). Then, in order to control the amount of replanning, probabilistic meta-level control (PMLC in Figure 4.1, described in Chapter 7) decides whether to replan (or probabilistically strengthen the schedule) in response to the changes in the agents probability profiles. Probabilistic meta-level control makes this decision with the goal of maintaining high utility schedules while also limiting unnecessary replanning. Finally, each time that a replan is performed, PADS first updates the probability profiles of the new schedule, and then a probabilistic schedule strengthening pass is performed (PSS in Figure 4.1, described in Chapter 6) in order to improve the expected utility of the new schedule. Finally, in multi-agent scenarios, agents would communicate these changes to other agents. Note that this diagram illustrates only the flow of PPM in order to demonstrate its principles. It does not, for example, show the flow of information to or from agents' executive systems.

Overall, the benefit of PPM is clear. When all components of PPM are used, the average utility earned in the multi-agent domain increases by 14% with an average overhead of only 18 seconds per problem, which corresponds to only 2% of the total time of execution.

In the next chapter, we discuss the PADS algorithm for both the single-agent and multi-agent domains. Chapter 6 details our probabilistic schedule strengthening approach, while Chapter 7 explains in depth the probabilistic meta-level control strategy. Finally, Chapter 8 presents our conclusions and future work.

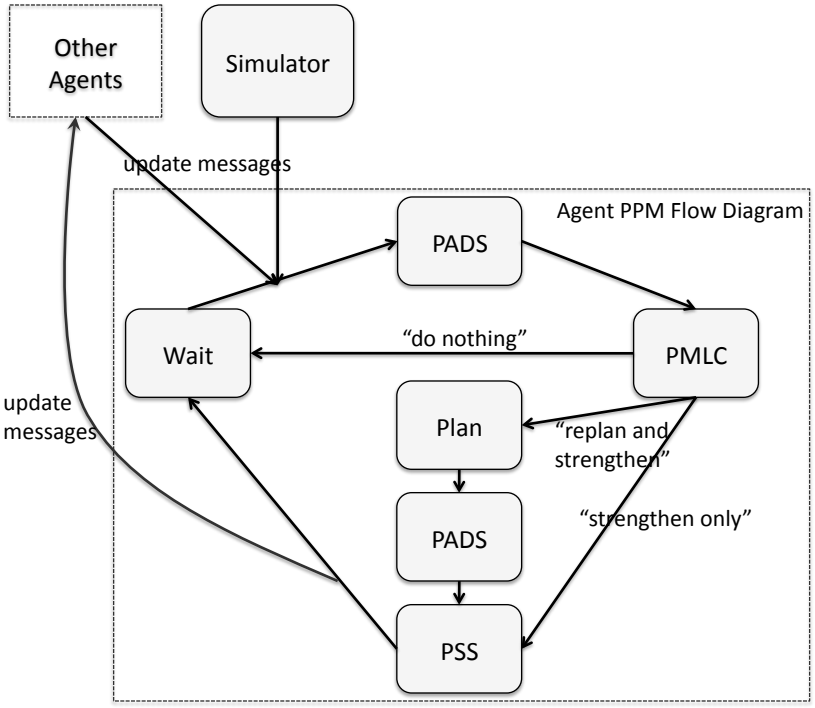


Figure 4.1: Flow diagram for PPM.

Chapter 5

Probabilistic Analysis of Deterministic Schedules

This section discusses in detail how to perform a *P*robabilistic *A*nalysis of *D*eterministic *S*chedules (PADS), which is the basis of our probabilistic plan management approach. We developed versions of this algorithm for both of our domains of interest. Section 5.1 provides an overview of how the probability analysis is performed in the two domains we consider, while Sections 5.2 and 5.3 provide details about the algorithms. The casual reader, after reading the overview section, can safely pass over the two more detailed sections that follow. Finally, Section 5.4 presents PADS experimental results.

5.1 Overview

The goal of the PADS algorithm is to calculate, for each activity in a problem's activity network, a *probability profile*, which includes the activity's probability of meeting its objective (po) and its expected utility (eu). The algorithm propagates probability profiles through the network, culminating in finding the probability profile of the taskgroup, which summarizes the expected outcome of execution and provides the overall expected utility.

For domains with temporal constraints like the two we consider, a method's finish time distribution (FT) is typically necessary for calculating the probability that the method meets the plan objective; in turn, its start time distribution (ST) and duration distribution are necessary for calculating its finish time distribution. Along with the method's probability of meeting its objective

and expected utility, we call these values the method's *probability profile*. In general, in this and following chapters we use uppercase variables to refer to distributions or sets and lowercase variables to refer to scalar values. We also use lowercase variables to refer to methods or activities, but uppercase variables to refer specifically to tasks.

Algorithm 5.1 provides the general, high-level structure for calculating a method's probability profile. It can be adjusted or instantiated differently for different domain characteristics, as we will see in Sections 5.2 and 5.3, when we discuss PADS in detail for the two domains.

Algorithm 5.1: Method probability profile calculation.

Function : `calcMethodProbabilityProfile` (m)

`//calculate` m`'s start time distribution`
1 $ST\,(m) =$ `calcStartTimes` (m)

`//calculate` m`'s finish time distribution`
2 $FT\,(m) =$ `calcFinishTimes` (m)

`//calculate` m`'s probability of meeting its objective`
3 $po\,(m) =$ `calcProbabilityOfObjective` (m)

`//calculate` m`'s expected utility`
4 $eu\,(m) =$ `calcExpectedUtility` (m)

The order in which activities' profiles are calculated is based on the structure of activity dependencies in the activity network. For example, a method is dependent on its predecessor and its prerequisites; correspondingly, its profile depends on the profiles of those activities and cannot be calculated until they are. Once these profiles are known, a method's start time distribution can be calculated (Line 1). Typically, calculating a method's start time distribution includes finding the maximum of its earliest start time, `est`, and its predecessors and prerequisite activities' finish time distributions, as it cannot start until those activities finish. A method's finish time distribution is found next (Line 2), and generally includes summing its start time distribution and duration distribution.

At any point during these calculations, a method may be found to have a non-zero probability of violating a constraint or otherwise not meeting its objective, leading to a failure. These cases are collected and combined to find m's probability of meeting its objective (Line 3). Similarly, if a method is found to have a non-zero *a priori* probability of failing, such information should be incorporated into the $po\,(m)$ calculation.

Finally, a method's expected utility is calculated (Line 4). Its expected utility depends on $po\,(m)$ and its utility, as well as any other dependencies (such as soft NLEs) that may affect it.

Its utility depends on the plan objective. If the plan objective is to execute all scheduled methods, individual methods should all have the same utility; if the plan objective includes an explicit model of reward or utility, a method's utility is defined by that.

Probability profiles are found differently for tasks. The general, high-level algorithm is shown in Algorithm 5.2. For tasks, start time distributions are ignored as they are not necessary for calculating a task's finish time distribution; instead, a task's finish time distribution depends on the finish time distributions of its children. A task's finish time distribution may only be necessary to explicitly calculate if there are direct relationships between it or its ancestors and other activities; however, we do not show that check here. The algorithm calculates tasks' probability profiles based on whether the task is an AND, OR or ONE task.

Algorithm 5.2: Task probability profile calculation.

Function : `calcTaskProbabilityProfile` (T)

```
1  switch Type of task T do
2      case AND
3          FT(T) = calcANDTaskFinishTimes(T)
4          po(T) = calcANDTaskProbabilityOfObjective(T)
5          eu(T) = calcANDTaskExpectedUtility(T)
6      case OR
7          FT(T) = calcORTaskFinishTimes(T)
8          po(T) = calcORTaskProbabilityOfObjective(T)
9          eu(T) = calcORTaskExpectedUtility(T)
10     case ONE
11         if |scheduledChildren(T)| == 1 then
12             c = sole-element(scheduledChildren(T))
13             FT(T) = FT(c)
14             po(T) = po(c)
15             eu(T) = eu(c)
16         else
17             po(T) = eu(T) = 0
18         end
19 end
```

For AND and OR tasks, the algorithm does not explicitly check the number of children that are scheduled. That is because we assume that unscheduled tasks have a po and eu of 0. This property means that the equations in the algorithm evaluate to the desired values without such a check. In this algorithm, the function `scheduledChildren` returns the scheduled children of a task.

An AND task's finish time distribution includes calculating the `max` of its children's, as it does not meet its plan objective until all children do (Line 3). Similarly, its probability of meeting the plan objective should equal the probability that all children do (Line 4). An AND task's expected utility is generally a function of the utility of its children, multiplied by the probability that it meets its objective; this function depends on the domain specifics (such as what the task's *uaf* is) (Line 5).

An OR task's finish time distribution includes calculating the `min` of its scheduled children's, as it meets its plan objective when its first child does (Line 7). Its probability of meeting the plan objective is the probability that at least one child does (which is equivalent to the probability that not all scheduled children do not) (Line 8). An OR task's expected utility is generally a function of the utility of its children; this function depends on the domain specifics (such as what the task's *uaf* is) (Line 9).

Trivially, ONE tasks' profiles equal those of their sole scheduled child. Typically, planners do not schedule more than one child under a ONE task. Therefore, we omit considering, for cases where more than one child is scheduled, the probability that only one succeeds. Instead, we say instead that in such a case the task has an expected utility of 0. Algorithms 5.1 and 5.2, put together, enable the expected utility of the entire schedule to be found. In subsequent equations, we refer to this as $eu\left(TG\right)$, where TG is the taskgroup.

5.1.1 Single-Agent PADS

Recall the representation of the single-agent domain: at the root is the AND taskgroup, and below it are the ONE tasks, each with their possible child methods underneath. That is, the plan objective is to execute one child of each task within its imposed temporal constraints. Trivially, therefore, $po\left(a\right)$ equals the probability that an activity a meets this objective. We define plan utility for this domain to be 1, if all tasks have one child method that executes within its constraints, or 0 if all do not. Due to this simple plan structure, PADS for this domain is fairly intuitive. It moves linearly through the schedule, calculating the probability profiles of each method, and passing them up to their parents, until all tasks' profiles have been found and the expected utility of the schedule is known.

5.1.2 Multi-Agent PADS

The overall objective of our multi-agent domain is to maximize utility earned. While calculating probability profiles for this domain, we define $po\left(a\right)$ to equal the probability that an activity a earns positive utility; its utility is trivially its domain-defined utility.

This domain is inherently more complex than our single-agent domain, leading to a number of extensions to the simple algorithms and equations presented above. First, the calculations of tasks' expected utilities depend on the tasks' *uaf* types, not just node types (*i.e.*, AND, OR or ONE). Second, the disables, facilitates and hinders relationships need to be factored in at various stages in the calculations as appropriate: disablers affect whether a method starts; and facilitators and hinderers affect its finish time and expected utility calculations. The analysis begins at the lowest method level, and propagates profiles both forward and upward through agents' schedules and the activity hierarchy until all activities' (including the taskgroup's) probability profiles have been calculated and the schedule's expected utility is known.

We have developed two versions of PADS for this domain. The first is intended to be performed offline and centrally in order to perform plan management before the initial global schedule is distributed to agents. The second version is used for probabilistic plan management during distributed execution. When PADS is performed globally, calculations begin with the first method(s) scheduled to execute, and later activities' profiles can be calculated as soon as the profiles of all activities they are dependent on have been determined. In the distributed case, the same basic propagation structures takes place; however, agents can update only portions of the activity network at a time and must communicate (via message passing) relevant activity probability profiles to other agents as the propagation progresses. Figure 5.1 shows a schedule, the resulting activity dependencies and profile propagation order for the small example shown in Figure 3.5, on page 39. Figure 5.2 shows how the profile propagation is done in a distributed way. Although it is not illustrated in the diagram for visual clarity, after agent Buster finish its calculations, it would pass the profiles of $T2$ and the taskgroup to agent Jeremy, as Jeremy has remote copies of them.

5.2 Single-Agent Domain

We have implemented PADS for a single-agent domain [Hiatt and Simmons, 2007]. Recall the characteristics of this domain: the plan's objective is that all tasks, each with exactly one method underneath them, execute without violating any temporal constraints. The only source of uncertainty is in method duration; and temporal constraints are in the form of minimal and maximal time lags between tasks, as well as a final deadline. We use a fixed times planner to generate schedules, and we expand solutions to be flexible times by translating them into an STN.

Recall from our discussion on STNs (Section 3.2.2, on page 31) that if a method does not begin by its latest start time, `lst`, the STN will become inconsistent. In the STN representing the deterministic schedules of this domain, time bounds are based only on methods' scheduled durations, *i.e.*, the edge representing the duration constraint specifies lower and upper bounds of

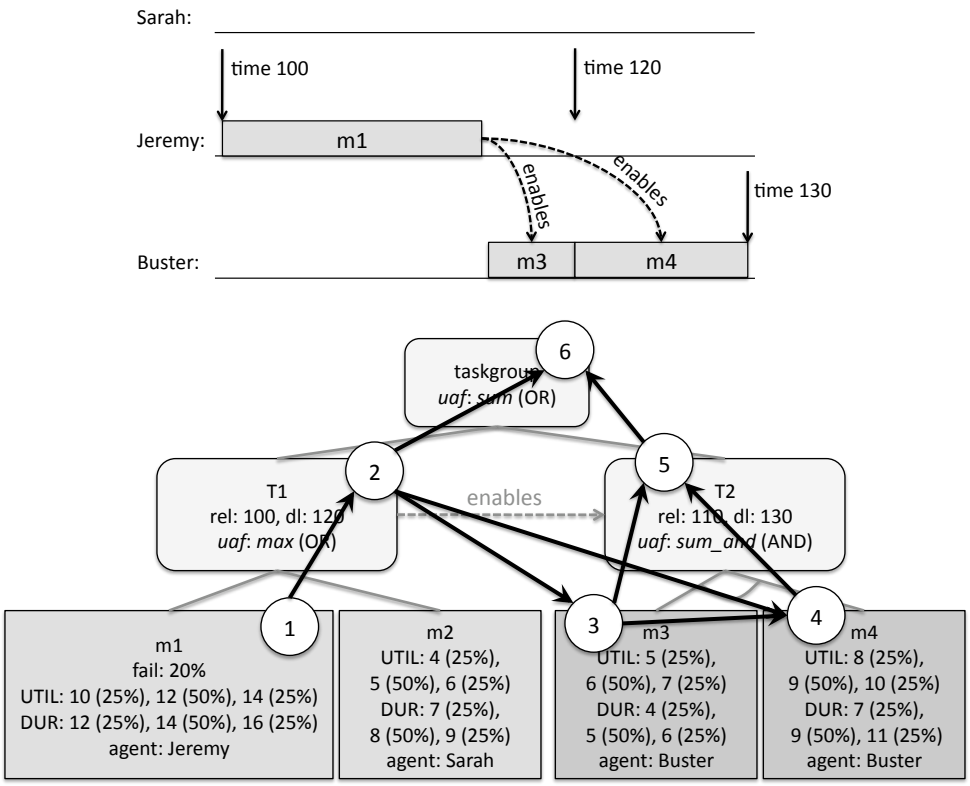


Figure 5.1: The activity dependencies (denoted by solid arrows) and propagation order (denoted by circled numbers) that result from the schedule shown for the problem in Figure 3.5.

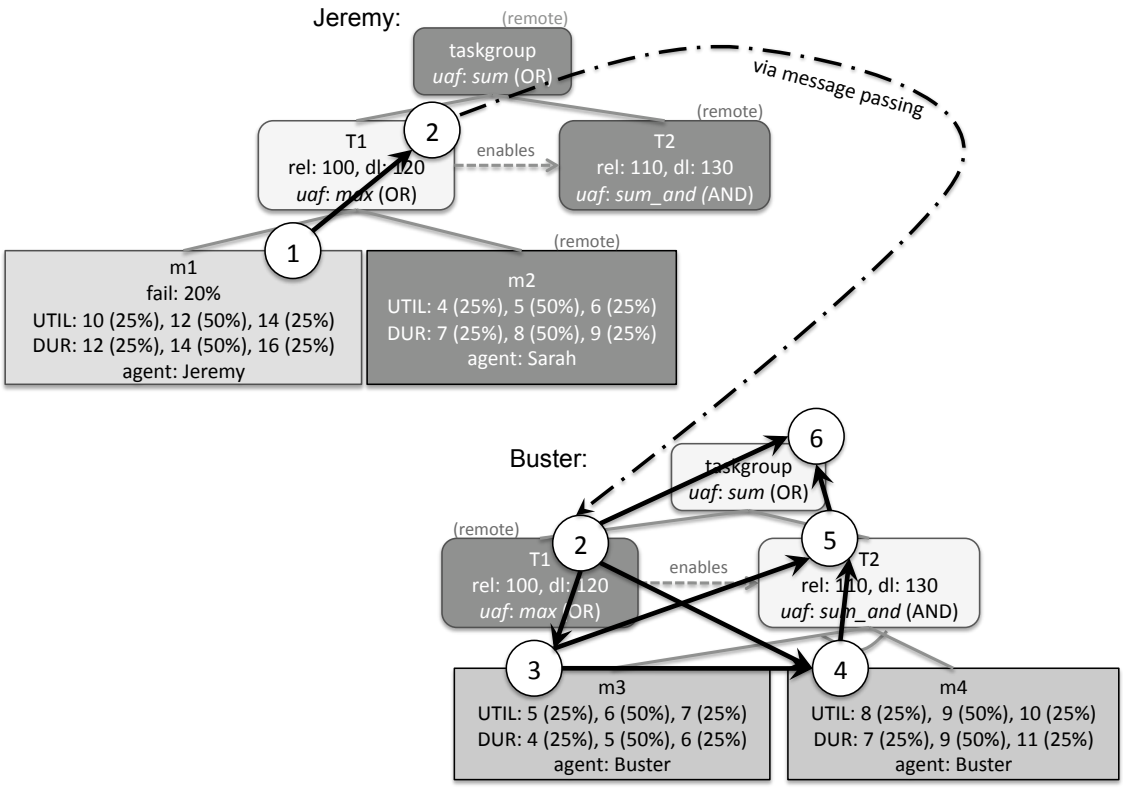


Figure 5.2: The same propagation as Figure 5.1, but done distributively. Jeremy first calculates the profiles for its activities, m1 and T1, and passes the profile of T1 to Buster; Buster can then finish the profile calculation for the rest of the activity network.

$[\texttt{dur}_m, \texttt{dur}_m]$ (see Section 3.2.2, on page 31, for this description). One implication of this is that, if a method misses its `lst`, the STN will become inconsistent; however, the schedule may still execute successfully, such as if the method begins after its `lst` but executes with a duration *less* than $\texttt{dur}_m$. Ideally, this contingency would be captured by the probabilistic analysis.

To do so, before performing its analysis, PADS expands the duration edges in the STN to include the range of possible integer durations for methods bounded by

$$[\texttt{max}\left(1, \mu\left(\texttt{DUR}\left(m\right)\right) - 3 \cdot \sigma\left(\texttt{DUR}\left(m\right)\right)\right), \mu\left(\texttt{DUR}\left(m\right)\right) + 3 \cdot \sigma\left(\texttt{DUR}\left(m\right)\right)]$$

This covers approximately 99.7% of methods' duration distributions. We use integer durations because method's executed durations are rounded to the nearest integer during simulation (Section 3.3.1). By expanding method durations in the STN, we can interpret the resulting time bounds differently. Now, if a method does not start by its revised `lst`, it is almost certain that the schedule cannot successfully be executed as-is. This allows PADS to more accurately calculate the schedule's probability of succeeding. We use a " $'$ " symbol to denote that we are using this modified STN: we refer to it as STN$'$, and time bounds generated from it are $\texttt{est}'$, $\texttt{lst}'$, etc.

5.2.1 Activity Dependencies

In this domain, since all tasks need to be executed within their constraints, the probability that the schedule meets its objective for a problem with T tasks is $po\left(TG\right) = po\left(m_1, ..., m_T\right)$, where $po\left(m_i\right)$ is the probability that method m_i executes without violating any constraints.

The probabilities $po\left(m_1\right), ..., po\left(m_T\right)$ are not independent. The successful execution of any method is dependent on the successful execution of its predecessors, because if a predecessor does not execute successfully (for example by violating its $\texttt{lft}'$) then execution of the current schedule no longer has any possibility of succeeding. Conditional probability allows us to rewrite the equation as:

$$po\left(m_1, ..., m_T\right) = \prod_{i=1}^{T} po\left(m_i \mid m_1, ..., m_{i-1}\right)$$

That is, a method's po value should equal the probability that it executes within all its constraints given that all previous ones do, as well. Then, the schedule's po value is calculated by multiplying the po values for the individual ONE tasks, which equal the po values of their sole scheduled child, as shown in Section 5.1. Note that we can avoid explicit consideration of prerequisite relationships; they are implicitly considered by the above consideration that all of a method's predecessors must successfully execute before it can, since the agent can execute only one thing at a time.

5.2.2 Probability Profiles

As the structure of the entire activity network is standard for all problems in this domain, we are able to present a closed-form PADS algorithm. Algorithm 5.3 shows the pseudocode for this. Note how this algorithm provides domain-specific implementations of the functions in Algorithm 5.1, and reuses its calculation of ONE tasks' probability profiles. We do not need to calculate the finish time distributions of the ONE tasks, because neither they nor their ancestors have any direct dependencies with other activities. Since the distributions in this domain are represented as normal distributions, we present them in terms of their mean and variance. Distributions are combined according to the equations presented in Section 3.3.1 (on page 35).

The function `schedule` in this and later algorithms returns the agent's current schedule, with methods sorted by their scheduled start time, and `children` and `scheduledChildren` return the children and scheduled children of a task, respectively. The algorithm iterates through each scheduled method and finds its probability of violating temporal constraints given that all previous methods successfully complete. As with Algorithm 5.1, it first calculates the method's start time distribution. We use the `truncate` function to approximate the `max` function (see Section 3.3.1 for specifics). It follows that its approximate start time distribution is its predecessor's finish time distribution, given it succeeds, truncated at its lower bound by $\texttt{est}'(m_i)$ (Line 2); as stated above, we can avoid explicit consideration of prerequisite relationships in our calculations.

The method's finish time distribution is the sum of its start time and duration distributions (Line 3). The probability of the method ending at a valid time equals the probability of it ending in the interval defined by its earliest finish time $\texttt{eft}'$ and its latest finish time $\texttt{lft}'$ (Line 4), as reported by the STN$'$. This value is m's po value, assuming all prior methods have finished successfully.

Assigning each method an arbitrary constant utility of 1, m_i's expected utility equals $po(m_i)$. Next, the method's finish time distribution is truncated to approximate its finish time distribution given it succeeds, FT_s, in order to pass that forward to future methods (Line 6). It is not necessary to find m_i's failure finish time distribution, as if any method fails the schedule cannot execute successfully as-is.

After the profiles of all methods have been found, their profiles are passed up to the task level. All tasks are ONE tasks, so their profiles equal that of their sole scheduled child (Lines 9-11); if no child (or more than one child) is scheduled, the tasks' po and eu values are 0 (Lines 12-14). Finally, after this has been done for all tasks, the individual task po values can be multiplied to find the overall probability of the schedule meeting its objective (Line 17). This, trivially, equals the schedule's expected utility (Line 18).

Algorithm 5.3: Taskgroup expected utility calculation for the single-agent domain with normal duration distributions and temporal constraints.

Function : calcExpectedUtility

foreach method $m_i \in$ schedule **do**

//calculate m_i's start time distribution

$ST(m_i) = \texttt{truncate}(\texttt{est}'(m_i), \infty, \mu(FT_s(m_{i-1})), \sigma(FT_s(m_{i-1})))$

//calculate m_i's finish time distribution

$FT(m_i) = ST(m_i) + \texttt{DUR}(m_i)$

//calculate m_i's probability of meeting its objective

$po(m_i) = \texttt{normcdf}(\texttt{lft}'(m_i), \mu(FT(m_i)), \sigma(FT(m_i))) - \texttt{normcdf}(\texttt{eft}'(m_i), \mu(FT(m_i)), \sigma(FT(m_i)))$

//calculate m_i's expected utility

$eu(m_i) = po(m_i)$

//adjust the FT distribution to be m_i's FT distribution given it succeeds

$FT_s(m_i) = \texttt{truncate}(\texttt{eft}'(m_i), \texttt{lft}'(m_i), \mu(FT(m_i)), \sigma(FT(m_i)))$

end

//calculate ONE tasks' profiles

foreach $T \in \texttt{children}(TG)$ **do**

  if $|\texttt{scheduledChildren}(T)| == 1$ **then**

    $po(T) = po(\texttt{sole-element}(T))$

    $eu(T) = eu(\texttt{sole-element}(T))$

  else

    $po(T) = 0$

    $eu(T) = 0$

  end

end

$po(TG) = \prod_{T \in \texttt{children}(TG)} po(T)$

$eu(TG) = po(TG)$

Before execution begins, this algorithm is used to find the probability profiles of the entire schedule. During execution, feedback of execution results, *i.e.*, the methods' actual, executed durations, are asserted into the STN as they become known. As PADS continues to update its probability values, it skips over activities that have already been executed and finds only the profiles for activities that have not yet completed. These profiles will be used in as part of a probabilistic meta-level control algorithm, which will be described in Section 7.2.

5.3 Multi-Agent Domain

This section discusses the implementation of PADS for the multi-agent domain we consider [Hiatt et al., 2008, 2009]. Recall that the goal of this domain is to maximize utility earned. We say, therefore, that the probability of meeting the plan objective equals the probability of earning positive utility. The multi-agent domain has many sources of uncertainty and a complicated activity structure. We assume a partial-order deterministic planner exists, which outputs schedules represented as STNs. Unlike the single-agent domain, however, when performing PADS we do not expand the STN to an STN′ that includes all possible method durations; this design decision was made to facilitate integration with the existing architecture [Smith et al., 2007].

The distributions we work with in this domain are discrete. In its algorithms, PADS combines the distributions according to the equations defined in Section 3.3.2 (on page 44). As part of the algorithms it also iterates through distributions. We use the **foreach** loop construct to designate this; *e.g.*, "**foreach** $ft \in FT(m)$," which iterates through each item ft in the distribution $FT(m)$. Inside the loop, PADS will also refer to $p(ft)$; this is the probability of ft occurring according to FT.

5.3.1 Activity Dependencies

Activity dependencies and po calculations need to be handled differently for this domain than in the single-agent domain. First, there is a more complicated dependency structure, due to more types of NLEs and, more specifically, NLEs between activities on different agents' schedules. There are also more ways for activities to fail than violating their temporal constraints, including other activities explicitly disabling them. Finally, because of the varying activity network structure and dependencies between activities, we do not present a closed-form formula representing $po(TG)$.

When calculating activity finish time profiles for this domain, we make the simplifying assumption that a method's non-local source NLEs finish times are independent of one another (*e.g.*, a method's enabler is different from its disabler or facilitator); this assumption is rarely broken

in practice (*e.g.*, an enabler rarely also serves as a facilitator). For tasks, we also assume that all children's finish time distributions are independent of one another. This assumption fails if, for example, both children are enabled by the same enabler. This approximation was found to be accurate enough for our purposes, however, and the extra computational effort required to sort out these dependencies was deemed unnecessary (see Section 5.4); we also show an example supporting this decision at the end of this section.

We do not, however, make this type of independence assumption when calculating po values, as it is too costly with respect to accuracy. To see why, consider the activity network and associated schedule shown in Figure 5.1 (on page 64). It is clear from this example that $po\,(T1) = 0.8$ because its only scheduled child is $m1$, and $m1$ only fails with a 20% likelihood. Both $m3$ and $m4$ are enabled by $T1$. Therefore, if neither has a probability of missing its deadline, $po\,(m3) = 0.8$ and $po\,(m4) = 0.8$ as they both can earn utility only if $T1$ does. Assuming that these values are independent means that $po\,(T2) = 0.8 \cdot 0.8 = 0.64$, when in fact it should equal 0.8; essentially, we are double-counting $T1$'s probability of failing in $po\,(T2)$. We deem this error unacceptably high, and so put forth the computational effort to keep track of these dependencies.

To track the dependencies, our representation of po for all activities is based on a *utility formula* $UF\,(a)$, which indicates what *events* must occur during execution for activity a to earn utility and how to combine the events (*e.g.*, at least one needs to happen, or all must happen, etc.) into an overall formula of earning utility. We define events as possible contingencies of execution, such as a method ending in a failure outcome, a method's disabler ending before the method begins, or a method missing its deadline. The formula assigns probabilities to each individual event, and can be evaluated to find an overall probability of earning utility.

As an example of how the utility formula is handled, consider again the example in Figure 5.1. We define $UF\,(T1) = (util\text{-}m1, (util\text{-}m1 \leftarrow 0.8))$. This formula says, intuitively, that "$m1$ must execute with positive utility for $T1$ to earn utility; this happens with a probability of 0.8." Formally, UF formulas have two parts: the *event formula* component, UF_{ev}, and the *value assignment* component, UF_{val}. In the example above, $UF_{ev}\,(T1) = util\text{-}m1$ and $UF_{val}\,(T1) = (util\text{-}m1 \leftarrow 0.8)$. The event formula component specifies which events must come true in order for $T1$ to earn utility; here, just $m1$ must earn utility. The *event variable* $util\text{-}m1$ designates "the event of $m1$ earning utility" and is used in place of $m1$ to avoid confusion. The value assignment component specifies probability assignments for the events; here, it says that the event of $m1$ earning utility will occur with probability 0.8 (due to $m1$'s possible failure outcome). $UF\,(T1)$ can be evaluated to find $po\,(T1)$, which here, trivially, is 0.8.

Since neither $m3$ nor $m4$ will fail on their own, but depend on $T1$ earning utility, their formulas mimic that of $T1$: $UF\,(m3) = UF\,(m4) = (util\text{-}m1, (util\text{-}m1 \leftarrow 0.8))$. Now we can

define $UF\left(T2\right)$. For $T2$ to earn utility, both $m3$ and $m4$ need to. Therefore, $UF_{ev}\left(T2\right)$ is the and of $UF_{ev}\left(m3\right)$ and $UF_{ev}\left(m4\right)$, as both $m3$ and $m4$ must earn utility for $T2$ to earn utility. $UF_{val}\left(T2\right)$ is the union of $UF_{val}\left(m3\right)$ and $UF_{val}\left(m4\right)$, ensuring all events have probability value assignments. So, $UF\left(T2\right) = \left(\left(and\ util\text{-}m1\ util\text{-}m1\right), \left(util\text{-}m1 \leftarrow 0.8,\ util\text{-}m1 \leftarrow 0.8\right)\right)$. This can be simplified to $UF\left(T2\right) = \left(util\text{-}m1, \left(util\text{-}m1 \leftarrow 0.8\right)\right)$, which evaluates to $po\left(T2\right) = 0.8$, the correct value.

More generally, there are five events that affect whether a method earns utility: (1) whether it is disabled before it begins to be executed; (2) whether its enablers must earn positive utility; (3) whether it begins executing by its `lst`; (4) whether it meets its deadline; and (5) whether its execution yields a successful outcome. These events have event variables associated with them (*e.g.*, the $suc\text{-}m$ variable represents a method m executing with a successful outcome), which can be combined together as an event formula that represents what must happen for a method to earn utility; as we saw above, that event formula can then be evaluated in conjunction with value assignments to find the method's probability of earning utility.

In this domain, if a method ends in a failure outcome, violates a domain constraint or cannot be executed, executing the remainder of the schedule as-is could still earn utility. Therefore, in contrast to the single-agent domain, it is necessary to explicitly calculate methods' failure finish time distributions to propagate forward. In this domain, a method's finish time is either when it actually finishes (whether it earns utility or not) or, if it is not able to be executed, when the next activity could begin executing. For example, if while a method's predecessor is executing, the agent learns that the method has been disabled by an activity d, the "finish time" of the disabled method is $\texttt{max}\left(FT\left(\texttt{pred}\left(m\right)\right),\ ft\left(d\right)\right)$, as that is the earliest time that the subsequently scheduled method (on the same agent's schedule) could start. The function `pred` returns the predecessor of a method (the method scheduled before it on the same agent's schedule).

We represent $FT\left(a\right)$ in terms of two components: $FT_u\left(a\right)$, or a's finish time distribution *when it earns utility*, and $FT_{nu}\left(a\right)$, or a's finish time distribution *when it does not earn utility*. Together, the probabilities of $FT_u\left(a\right)$ and $FT_{nu}\left(a\right)$ sum to 1; when we refer to $FT\left(a\right)$ alone, it is the concatenation of these distributions.

We now return back to our assumption that a task's children's finish time distributions are independent of one another. When calculating the finish time distribution of task $T2$ from the above example, we would double-count the probability that $T1$ fails and does not enable either $m3$ or $m4$. Say, hypothetically, that we have already calculated that $FT_u\left(m3\right) = \{\{118, 0.3\},\ \{120, 0.5\}\}$, and $FT_u\left(m4\right) = \{\{127, 0.6\},\ \{129, 0.2\}\}$. Because $T2$ is an AND task, $FT_u\left(T2\right)$ equals the maximum of its children's FT_u distributions, since it does not earn utility until both children have. When we find the maximum of these distributions as if they were independent, we get $FT_u\left(T2\right) =$

$\{\{127, 0.48\},\ \{129, 0.16\}\}$. Intuitively, however, the finish time distribution of $T2$ should be $\{\{127, 0.6\},\ \{129, 0.2\}\}$, because $m3$ and $m4$ always either both earn utility, or both do not earn utility. If, however, we scale $FT_u\,(T2)$ by our previously calculated $po\,(T2)$, so that the distribution of $T2$'s finish time when it earns utility sums to $po\,(T2)$, we get the desired value of $FT_u\,(T2) = \{\{127, 0.6\},\ \{129, 0.2\}\}$. Although this approach does not always result in the exactly correct finish time distributions, this example illustrates why we make the independence assumption for tasks' finish time distributions, and introduces the extra step we take of scaling the distributions by activities' po values.

5.3.2 Probability Profiles

We define the algorithms of PADS recursively, with each activity calculating its profile in terms of the profiles of other activities it is connected to. The complex activity dependency structure for this domain also means that the high-level algorithms presented in Algorithms 5.1 and 5.2 (on pages 60 and 61) must be elaborated upon in order to calculate probability profiles for activities in this domain. We present much of the probability profile calculation in this domain in terms of providing appropriate individual functions to instantiate these algorithms.

The following algorithms are used in both the global and distributed versions of PADS for this domain. Then, Sections 5.3.3 and 5.3.4 addresses the global and distributed versions, specifically.

Method Probability Profile Calculations

The calculation of methods' profiles for this domain is the most complicated, as all activity dependencies (*e.g.*, predecessors, enablers, disablers, facilitators and hinderers) are considered at this level.

Recall again the five events determining whether a method earns utility: (1) it is not disabled; (2) it is enabled; (3) it meets its `lst` and begins executing; (4) it meets its deadline; and (5) it does not end in a failure outcome. Of these events, (5) is independent of the others; we also, as previously stated, assume that (1) and (2) are independent. Event (3), however, depends on whether it is able to begin execution and is contingent on (1) and (2); similarly, (4) is dependent on (3). To find the overall event formula of m earning utility, then, we find the probability value assignments of these individual events given these dependencies. We list them explicitly below along with the associated event variable:

1. $dis\text{-}m$ - The event of m being disabled before it starts.

2. en-m - The event of m being enabled; we assume this is independent of dis-m.
3. st-m - The event of m beginning execution given that it is not disabled and is enabled.
4. ft-m - The event of m meeting its deadline given that it begins execution.
5. suc-m - The event of m not ending in a failure outcome; due to its *a priori* likelihood, it is independent of the other events.

Overall, $UF_{ev}(m) = (and\ (not\ dis\text{-}m)\ en\text{-}m\ st\text{-}m\ ft\text{-}m\ suc\text{-}m)$.

Two of these events, dis-m and en-m are actually event formulas, in that they are determined by the outcome of other events. For example, $en\text{-}m = (and\ UF_{ev}(en1)\ UF_{ev}(en2)\ ...)$, where $en1$, $en2$, etc. are the enablers of m. The variable dis-m equals a similar formula. These formulas appear in the place of dis-m and en-m in m's event formula.

Of the five events determining whether a method earns utility, the probability values (and event formulas, if applicable) of two – all enablers earning utility, and the method having a successful outcome – can be found initially. The values of the other three, however, depend on the method's start and finish times. Therefore as we calculate the method's start time and finish time distributions, we also calculate the value assignments (and event formulas, for the case of disablers) for these three cases in order to avoid duplicate work.

In the algorithms presented below, we assume that variables are global (*e.g.*, $ST(m)$, $FT(m)$, etc.); therefore, these functions in general do not have return values.

Start Time Distributions We assume in our calculations that methods begin at their `est`, as executives have incentive to start methods as early as possible. This assumption, in general, holds; we discuss one example of a method not being able to start at its `est` in our PADS experiments in Section 5.4.2.

When calculating a method's starting distribution, as execution continues in this domain past a method failure, we must include in our implementation of Line 1 of Algorithm 5.1 consideration for what happens when a method's enablers fail. We must also check for disablers. And, as execution continues even if a constraint is violated, we must explicitly check to see if methods' will be able to start by their `lst`.

The algorithm for `calcStartTimes` is shown in Algorithm 5.4. In this domain, a method's start time distribution is a function of three things: (1) the finish time distribution of the method's predecessor, $FT(\texttt{pred}(m))$; (2) the finish time distributions of its enablers *given that they earn utility*, $FT_u(e)$ (recall that m will not execute until all enablers earn utility); and (3) the method's

release (earliest possible start) time. Along the way, several types of failures can occur: (1) the method might be disabled; (2) the method might not be enabled; and (3) the method might miss its `lst`. Before a method's start time distribution can be calculated, the profiles of m's predecessor and all enablers must have been calculated. If all enablers are not scheduled, then $po\,(m)$ and $eu\,(m)$ both equal 0.

The algorithm begins by generating a distribution of the maximum of its enablers' finish times (Line 1). The first component shown, $\{\max_{e \in \mathtt{en}(m)} FT_u\,(e)\}$ is the distribution representing when all enablers will successfully finish. The second component, $\{\{\infty, 1 - \prod_{e \in \mathtt{en}(m)} (1 - po\,(e))\}\}$, represents the group's finish time if not all enablers succeed; in this case, the group as a whole never finishes and so has an end time of ∞.

The algorithm iterates through this distribution, as well as the distribution of m's predecessor's finish time (Line 4), to consider individual possible start times st. It first checks to see if the method might be disabled by the time it starts; if it is, the method would immediately be removed from the schedule without being executed. To calculate the probability that m is disabled, the algorithm considers the distribution of the first of m's disablers to successfully finish with utility (Lines 8 - 9). If there is a chance a disabler will finish with utility by st, it adds this probability (scaled by the probability of st) to $FT_{nu}\,(m)$, along with m's finish time in that case (Lines 10-11).

If m might be disabled it also makes note of it for $UF\,(m)$. For each potential disabler the algorithm creates a unique event variable representing the disabling (Line 12). The variable is of the form d-dis-d', where d' is either m or an ancestor of m, whichever is directly disabled by d. The algorithm increments the probability of this event and adds it to m's disable formula (Lines 13-14). It also increments the probability that m will be disabled at this start time for later use (Line 15).

Next, the algorithm checks ft_e. If ft_e equals ∞, then m is not enabled in this iteration. In such cases, m is not removed from the schedule until its `lst` has passed; the planner will wait to remove it just in case the failed enabler(s) can be rescheduled. Therefore we add $\mathtt{lst}\,(m) + 1$ to $FT_{nu}\,(m)$ along with the probability $p_{st} - p_{disbl}$, which is the probability of this iteration minus the probability that m has already been disabled (Line 20).

Finally, if m is not disabled and is enabled, but $st > \mathtt{lst}\,(m)$, then this is an invalid start time (Line 23). In this case m will miss its latest start time with probability $p_{st} - p_{disbl}$. The associated finish time is the maximum of the predecessor's finish time and $\mathtt{lst}\,(m) + 1$. We add these to $FT_{nu}(m)$ (Line 24). Otherwise, m can start at time st and so we add $p_{st} - p_{disbl}$ and st to $ST\,(m)$ (Line 26). This process is repeated for all possible st's. Note that $ST\,(m)$ does not sum to 1; instead, it sums to the probability that m successfully begins execution.

At the end of the algorithm we collect information for $UF\,(m)$. The event formula representing

Algorithm 5.4: Method start time distribution calculation with discrete distributions, temporal constraints and hard NLEs.

Function : `calcStartTimes` (m)

$E = \{\max_{e \in \text{en}(m)} FT_u(e)\} \bigcup \{\{\infty, 1 - \prod_{e \in \text{en}(m)} po(e)\}\}$
$p_{norm} = 0$
foreach $ft_e \in E$ **do**
  foreach $ft_{pred} \in FT(\text{pred}(m))$ **do**
    $st = \max(\text{rel}(m), ft_{pred}, ft_e)$
    $p_{st} = p(ft_e) \cdot p(ft_{pred})$
    `//might the method be disabled in this combination?`
    $p_{disbl} = 0$
    foreach $D \in \mathcal{P}(\text{dis}(m))$ **do**
      foreach $ft_d \in \min_{d \in \text{dis}(m)} (d \in D \; ? \; FT_u(d) \,|FT_{nu}(d) + \infty)$ **do**
        if $ft_d \leq st \; and \; ft_d \leq \text{lst}(m)$ **then**
          `//the method is disabled`
          $\text{push}(\{\max(ft_{pred},\ ft_d),\ p_{st} \cdot p(ft_d)\}, FT_{nu}(m))$
          $a = \text{directlyDisabledAncestor}(m, d)$
          $p(d\text{-}dis\text{-}a) = p(d\text{-}dis\text{-}a) + p_{st} \cdot p(ft_d)$
          $dis_{ev} = dis_{ev} \cup \{d\text{-}dis\text{-}a\}$
          $p_{disbl} = p_{disbl} + p_{st} \cdot p(ft_d)$
        end
      end
    end
    `//is the method enabled for this start time?`
    if $ft_e == \infty$ **then**
      $\text{push}(\{\max(\text{lst}(m) + 1,\ ft_{pred}),\ p_{st} - p_{disbl}\},\ FT_{nu}(m))$
    else
      $p_{norm} = p_{norm} + p_{st} - p_{disbl}$
      if $st > \text{lst}(m)$ **then**
        `//invalid start time`
        $\text{push}(\{\max(\text{lst}(m) + 1,\ ft_{pred}),\ p_{st} - p_{disbl}\},\ FT_{nu}(m))$
      else
        $\text{push}(\{st,\ p_{st} - p_{disbl}\},\ ST(m))$
      end
    end
  end
end
`//collect the probability formulas for` $UF(m)$
$dis\text{-}m = (or\ dis_{ev})$
$p(st\text{-}m) = \Sigma_{st \in ST(m)} p(st) / p_{norm}$

m being disabled is $(not\ (or\ dis_{ev}))$ where dis_{ev} and the corresponding value assignments of it were found earlier (Line 31). The event variable representing m meeting its `lst` and starting is st-m, which happens with probability $\Sigma_{st \in ST(m)} p\left(st\right) / p_{norm}$; it is normalized in this way so that it equals the probability that it starts given that it is not disabled and is enabled (Line 32).

Soft NLE Distributions In order to calculate m's finish time and expected utility, we first need to know m's facilitating and hindering effects. Algorithm 5.5 presents the algorithm for this. The algorithm returns a set of distributions representing the effects on both m's duration NLE_d and utility NLE_u. Further, it indexes the distribution NLE_d by m's possible start times; each entry is the distribution of possible NLE coefficients for that start time.

Algorithm 5.5: Method soft NLE duration and utility effect distribution calculation with discrete distributions.

Function : `calcNLEs` (m)
1 **foreach** $st \in ST\left(m\right)$ **do**
2 $DC = \{1, 1\}$
3 $UC = \{1, 1\}$
4 **foreach** $f \in$ `fac` (m) **do**
5 $p\text{-}nle = \sum_{ft \in FT_u(f)} ft \leq st\ ?\ p\left(ft\right)\ :\ 0$
6 `push` $(\{\{(1 - df \cdot prop\text{-}util\left(f, st\right)), p\text{-}nle\},\ \{1, 1 - p\text{-}nle\}\}, DC)$
7 `push` $(\{\{(1 + uf \cdot prop\text{-}util\left(f, st\right)), p\text{-}nle\},\ \{1, 1 - p\text{-}nle\}\}, UC)$
8 **end**
9 **foreach** $h \in$ `hind` (m) **do**
10 $p\text{-}nle = \sum_{ft \in FT_u(h)} ft \leq st\ ?\ p\left(ft\right)\ :\ 0$
11 `push` $(\{\{(1 + df \cdot prop\text{-}util\left(h, st\right)), p\text{-}nle\},\ \{1, 1 - p\text{-}nle\}\}, DC)$
12 `push` $(\{\{(1 - uf \cdot prop\text{-}util\left(h, st\right)), p\text{-}nle\},\ \{1, 1 - p\text{-}nle\}\}, UC)$
13 **end**
14 **foreach** $dc \in \prod_{c_i \in DC} c_i$ **do**
15 `push` $(\{dc,\ p\left(dc\right)\},\ NLE_d\left(m, st\right))$
16 **end**
17 **foreach** $uc \in \prod_{c_i \in UC} c_i$ **do**
18 `push` $(\{uc,\ p\left(uc\right)\},\ NLE_u\left(m\right))$
19 **end**
20 **end**

At a high level, the algorithm iterates through m's start times, facilitators and hinderers. For each start time st in $ST\left(m\right)$ and facilitator (hinderer), it finds the probability that the facilitator (hinderer) will earn utility by time st (Line 5). It then adds a small distribution to the lists DC and

UC representing the possible coefficients of what the facilitator's (hinderer's) effect on m will be: with probability $p\text{-}nle$ it will have an effect, with probability $1-p\text{-}nle$ it will not (Lines 6-7). Then, the algorithm combines these small distributions via multiplication to find an overall distribution of what the possible NLE coefficients for m are at time st (Line 14), and the result is added to m's NLE distributions (Line 15). This is repeated for each possible start time until the full distributions are found.

Finish Time Distributions Next, $FT_u\left(m\right)$ is calculated. The implementation of Line 2 of Algorithm 5.1 (on page 60) is complicated due to the presence of NLEs and the separation of FT_u and FT_{nu}. The algorithm is illustrated in Algorithm 5.6. It iterates through $st \in ST\left(m\right)$, and for each start time multiplies m's duration distribution with $NLE_d\left(m, st\right)$ to find m's effective duration distribution when it starts at time st. Iterating through the effective duration distribution, the algorithm considers each finish time ft that results from adding each possible start time and duration together (Lines 2-3). If $ft > \texttt{dl}(m)$, the method earns zero utility and the algorithm adds the probability of this finish time $p\left(ft\right)$ and ft to $FT_{nu}(m)$ (Line 7). Otherwise, it is a valid finish time. At this point, the only other thing to consider is that m might end with a failure outcome. Therefore $p\left(ft\right) \cdot \texttt{fail}\left(m\right)$ and the time ft are added to $FT_{nu}(m)$ (Line 10), and $p\left(ft\right) \cdot \left(1 - \texttt{fail}\left(m\right)\right)$ and the time ft are added to $FT_u(m)$ (Line 11).

The last line of the algorithm collects the probability value assignment for $ft\text{-}m$ (Line 15). Its value is the probability that it does not miss a deadline given that it begins execution.

Probability of Earning Utility To calculate m's probability of meeting the plan objective (which equals its probability of earning positive utility), all of the event probabilities that have been calculated in preceding algorithms are combined. This is shown in Algorithm 5.7. In the preceding algorithms the event formula for $dis\text{-}m$ and the corresponding value assignments were found, as well as $p\left(st\text{-}m\right)$ and $p\left(ft\text{-}m\right)$; additionally, the event formulas and value assignments for $en\text{-}m$ and $suc\text{-}m$ are already known. With these formulas and value assignments in hand, the algorithm can put it all together to get m's final utility formula. As all of the events must happen for m to earn utility, the event formula is the and of these five individual event formulas (Line 1). Trivially, the value assignments are the union of the individual value assignments (Line 2).

To calculate $po\left(m\right)$ from $UF\left(m\right)$, there are two options: reduce the event formula $UF_{ev}\left(m\right)$ such that each event appears only once and evaluate it using the variable probability values in $UF_{val}\left(m\right)$; or consider all possible assignments of truth values to those events and, for each assignment, calculate the probability of that assignment occurring as well as whether it leads to an eventual success or failure. The former, however, is not guaranteed to be possible, so we chose the

Algorithm 5.6: Method finish time distribution calculation with discrete distributions, temporal constraints and soft NLEs.

Function : $\texttt{calcFinishTimes}\,(m)$
1 $p_{dl} = 0$
2 **foreach** $st \in ST\,(m)$ **do**
3 **foreach** $dur \in \lceil \texttt{DUR}\,(m) \cdot NLE_d\,(m, st) \rceil$ **do**
4 $ft = st + dur$
5 $p\,(ft) = p\,(st) \cdot p\,(dur)$
6 **if** $ft > \texttt{dl}\,(m)$ **then**
 //the method did not meet its deadline
7 $\texttt{push}\,(\{ft, p\,(ft)\},\ FT_{nu}\,(m))$
8 $p_{dl} = p_{dl} + p\,(ft)$
9 **else**
 //the method might finish with a failure outcome
10 $\texttt{push}\,(\{ft,\ p\,(ft) \cdot \texttt{fail}\,(m)\},\ FT_{nu}\,(m))$
 //otherwise it ends with positive utility
11 $\texttt{push}\,(\{ft,\ p\,(ft) \cdot (1 - \texttt{fail}\,(m))\},\ FT_{u}\,(m))$
12 **end**
13 **end**
14 **end**
 //collect the probability for m meeting its deadline
15 $p\,(ft\text{-}m) = \left(\Sigma_{st \in ST(m)} p\,(st) - p_{dl}\right) / \Sigma_{st \in ST(m)} p\,(st)$

Algorithm 5.7: Method probability of earning utility calculation with utility formulas.

Function : $\texttt{calcProbabilityOfObjective}\,(m)$
 //first put it all together to get $UF\,(m)$
1 $UF_{ev}\,(m) = \left(and\ \ (not\ dis\text{-}m)\ \bigcup_{e \in \texttt{en}(m)} UF_{ev}\,(e)\ \ st\text{-}m\ ft\text{-}m\ suc\text{-}m\right)$
2 $UF_{val}\,(m) = \{\bigcup_{d\text{-}dis\text{-}a \in dis\text{-}m} d\text{-}dis\text{-}a \leftarrow p\,(d\text{-}dis\text{-}a)\,,\ \bigcup_{e \in \texttt{en}(m)} UF_{val}\,(e)\,,\ st\text{-}m \leftarrow p\,(st\text{-}m)\,,\ ft\text{-}m \leftarrow p\,(ft\text{-}m)\,,\ suc\text{-}m \leftarrow (1 - \texttt{fail}\,(m))\}$
 //evaluate $UF\,(m)$
3 $po\,(m) = \texttt{evaluateUtilityFormula}\,(UF\,(m))$

latter to use in our approach, shown as Algorithm 5.8.

The algorithm creates different truth value assignments by looking at all possible subsets of the events and assigning "true" to the events in the set and "false" to those not in the set. We use the powerset function $\mathcal{P}$ to generate the set containing all subsets of the events (Line 1). For each subset S, the algorithm evaluates whether $UF_{en}(a)$ is true when all variables in S are true and all that are not are false (Line 3). It does so by substituting all variables in $UF_{en}(a)$ that are in S with **true** and all that are not with **false**, and evaluating the formula. If it is true, it calculates the probability of that subset and adds it to $po(a)$. Once all subsets are considered we have the final value of $po(a)$.

Finally, PADS normalizes $FT_u(a)$ and $FT_{nu}(a)$ so that they sum to the values $po(a)$ and $1 - po(a)$, respectively, as discussed in Section 5.3.1. This algorithm is the same for all activities in the hierarchy.

Algorithm 5.8: Activity utility formula evaluation.

Function : `evaluateUtilityFormula` (a)
1 **foreach** $S \in \mathcal{P}(UF_{var}(a))$ **do**
2 $\quad L = UF_{var}(a) - S$
3 $\quad$ **if** `evaluate` $(UF_{en}/\forall s \in S \leftarrow$ **true**$/\forall l \in L \leftarrow$ **false**$)$ **then**
4 $\quad\quad p(S) = \prod_{s \in S} p(s) \cdot \prod_{l \in L} (1 - p(l))$
5 $\quad\quad po(a) = po(a) + p(S)$
6 $\quad$ **end**
7 **end**
8 `normalize` $(FT_u(a), po(a))$
9 `normalize` $(FT_{nu}(a), 1 - po(a))$

Expected Utility Algorithm 5.9 shows how $eu(m)$ is calculated for methods, providing an implementation for Line 4 of Algorithm 5.1 (on page 60). The algorithm iterates through `UTIL` (m) and $NLE_u(m)$ to find the method's effective average utility, after soft NLEs, and then multiplies that by m's probability of earning utility.

OR Task Probability Profile Calculations

An OR task's utility at any time is a direct function of its children's: if any child has positive utility, the task does as well; similarly if no child has positive utility, nor will the task. Therefore its FT_u distribution represents when the first child earns utility, and its FT_{nu} distribution represents when

Algorithm 5.9: Method expected utility calculation with discrete distributions and soft NLEs.

```
   Function : calcExpectedUtility(m)
1  eu(m) = 0
2  foreach util ∈ UTIL(m) do
3      enle = 0
4      foreach nle ∈ NLE_u(m) do
5          enle = enle + nle · p(nle)
6      end
7      eu(m) = eu(m) + (enle · util · p(util))
8  end
9  eu(m) = eu(m) · po(m)
```

the last child fails to earn utility. To find these distributions, it is necessary to consider all possible combinations of children succeeding and failing. As a simple example of this, if a task T has two scheduled children a and b, to calculate the finish time distribution of T we must consider:

- The min of $FT_u(a)$ and $FT_u(b)$, to find when T earns utility if both a and b earn utility.
- $FT_u(a)$, scaled by $1 - po(b)$, to find when T earns utility if only a earns utility.
- $FT_u(b)$, scaled by $1 - po(a)$, to find when T earns utility if only b earns utility.
- The max of $FT_{nu}(a)$ and $FT_{nu}(b)$, to find when T fails to earn utility if neither a nor b earn utility.

We do this algorithmically by calculating the powerset of T's children, where each subset represents a hypothetical set of children earning utility. A task's probability profile cannot be calculated until the profiles of all scheduled children have been found; if there are no scheduled children, its eu and po values equal 0.

Algorithm 5.10 shows how to calculate these distributions, providing implementation for Line 7 of Algorithm 5.2 (on page 61). As described in the above paragraph, the algorithm considers each possible set of task T's children earning utility by taking the powerset of T's scheduled children (Line 1). Each set S in the powerset represents a set of children that hypothetically earns utility; the set difference between T's scheduled children and S is L, a set of children that hypothetically does not earn utility. If the size of set S is at least one, meaning at least one child earns utility in this hypothetical case, the algorithm finds the distribution representing the minimum finish time of the children in S and iterates through it, considering each member ft in turn (Line 4). The probability of this finish time under these circumstances is the probability of ft multiplied by the probability that all children in L do not earn utility (*i.e.*, the product of $1 - po(l)$ for each $l \in L$) (Line 5).

The algorithm adds each possible finish time and its probability to $FT_u(T)$ (Line 6). Otherwise, if no child earns utility in the current hypothetical case (*i.e.*, S is the empty set), the algorithm finds the latest finish time distribution of the hypothetically failed children and adds it to $FT_{nu}(T)$ (Line 10).

Algorithm 5.10: OR task finish time distribution calculation with discrete distributions and *uafs*.

```
   Function : calcORTaskFinishTimes(T)
 1 foreach S ∈ P(scheduledChildren(T)) do
 2     L = scheduledChildren(T) − S
 3     if |S| ≥ 1 then
           //at least one child earned utility
 4         foreach ft ∈ min_{s∈S} FT_u(s) do
 5             p_ft = p(ft) · ∏_{l∈L}(1 − po(l))
 6             push({ft, p_ft}, FT_u(T))
 7         end
 8     else
           //no child earned utility
 9         foreach ft ∈ max_{l∈L} FT_nu(l) do
10             push({ft, p(ft)}, FT_nu(T))
11         end
12     end
13 end
```

An OR task's utility formula is simple to calculate. The event formula is the *or* of its children's formulas, and the value assignments are the union of its children's assignments. It can be evaluated by using the function `evaluateUtilityFormula` from Algorithm 5.8.

The expected utility of an OR task depends on its *uaf*; its calculation is shown in Algorithm 5.11. For a *sum* task, it is the sum of the expected utilities of its children, as the expectation of a sum is the sum of the expectation (Line 3). For a *max* task, the algorithm must consider all possible combinations of children succeeding or failing and find what the maximum utility is for each combination. As with an OR task's finish time distribution, to do so the algorithm considers the powerset of all scheduled children, where each set in the powerset represents a hypothetical combination of children earning utility. The probability that any set S in that powerset occurs is the probability that the children in S earn utility and the children in L do not (Line 10). As the task's expected utility is the max of only the children that earn utility, its expected utility is a function of the expected utilities of its children *provided they earn positive utility*. This number can be easily calculated for activities as part of their probability profiles. We call this value epu (Line 11).

Algorithm 5.11: OR task expected utility calculation with discrete distributions and *uafs*.

```
   Function : calcORTaskExpectedUtility(T)
 1 switch Type of task T do
 2     case sum
 3         eu(T) = Σ_{s∈scheduledChildren(T)} eu(s)
 4     case max
 5         foreach S ∈ P(scheduledChildren(T)) do
 6             if |S| == 0 then
 7                 eu(T) = 0
 8             else
 9                 L = scheduledChildren(T) − S
10                 p(S) = ∏_{s∈S} po(s) ∏_{l∈L} (1 − po(l))
11                 eu(T) = eu(T) + p(S) · max_{s∈S} epu(s)
12             end
13         end
14     end
15 end
```

AND Task Probability Profile Calculations

An AND task's utility at any time depends on whether all children have earned utility. If any child has zero utility, then the task does as well. Therefore its FT_u distribution represents when the last child earns utility, and its FT_{nu} distribution represents when the first child fails to earn utility. As with an OR task's finish time distribution, to do so it is necessary to consider all possible combinations of children succeeding and failing, which can be generated algorithmically by the powerset operator. A task's probability profile cannot be calculated until the profiles of all scheduled children have been found; if not all children are scheduled, its eu and po values equal 0 and no further calculation is necessary.

The finish time distribution calculation is shown in Algorithm 5.12. The algorithm considers each possible set of task T's children *not* earning utility by taking the powerset of T's scheduled children (Line 1). Each set L in the powerset represents a hypothetical set of children that do not earn utility; the set difference between T's scheduled children and L is S, the children that do earn utility in this hypothetical case. If at least one child does not earn utility in the current set, the algorithm finds the distribution representing the minimum finish time of the children who do not earn utility and consider each possible value ft in the distribution (Line 4). The probability of this finish time under these circumstances is the probability of ft times the probability that all children

in S do earn utility (Line 5). We add each possible finish time and its probability to $FT_{nu}(T)$ (Line 10). Otherwise, if all children earn utility, then we find the latest finish time of the children and add it to $FT_u(T)$ (Line 10).

Algorithm 5.12: AND task finish time distribution calculation with discrete distributions and *uafs*.

```
   Function : calcANDTaskFinishTimes(T)
 1 foreach L ∈ P(scheduledChildren(T)) do
 2     S = scheduledChildren(T) − L
 3     if |L| ≥ 1 then
           //at least one child did not earn utility
 4         foreach ft ∈ min_{l∈L} FT_nu(l) do
 5             p_ft = p(ft) · ∏_{s∈S} po(s)
 6             push({ft, p_ft}, FT_nu(T))
 7         end
 8     else
           //all children earned utility
 9         foreach ft ∈ max_{s∈S} FT_u(s) do
10             push({ft, p(ft)}, FT_u(T))
11         end
12     end
13 end
```

An AND task's event formula is the and of its children's formulas, and the value assignments are the union of its children's assignments. It can be evaluated by using the function `evaluateUtilityFormula` from Algorithm 5.8.

The algorithm for calculating the expected utility of an AND task is shown in Algorithm 5.13. Because an and task earns utility only when all children do, its expected utility is a function of the expected utilities of its children *provided they earn positive utility*. As mentioned in the discussion of OR tasks' expected utility calculations, this value, epu, can easily be calculated as part of an activity's probability profile. If the activity is a *sum_and* task, it is the sum over all children of this estimate (Line 3); if the activity is a *min* task, it is the minimum of these estimates (Line 5).

ONE Task Probability Profile Calculation

A ONE task's probability profile is that of its sole scheduled child; if the number of scheduled children is not one, the task will not earn utility and its profile is trivial. Its implementation is the same as in Algorithm 5.2 (on page 61).

Algorithm 5.13: AND task expected utility calculation with discrete distributions and *uafs*.

Function : `calcANDTaskExpectedUtility` (T)
switch *Type of task* T **do**
    case *sum_and*
        $eu(T) = po(T) \cdot \Sigma_{c \in \texttt{children}(T)} epu(c)$
    case *min*
        $eu(T) = po(T) \cdot \texttt{min}_{c \in \texttt{children}(T)} epu(c)$
    end
end

Expected Utility Loss / Gain

As part of its analysis, PADS also calculates two heuristics which are used to rank activities by their importance by probabilistic schedule strengthening (Chapter 6). It measures this importance in terms of *expected utility loss* (eul) and *expected utility gain* (eug). Both of these metrics are based on the expected utility of the taskgroup assuming that an activity a earns positive utility, minus the expected utility of the taskgroup assuming that a earns zero utility:

$$eu(TG \mid po(a) = 1) - eu(TG \mid po(a) = 0)$$

In this and subsequent chapters, we rely on the | operator, which signifies a change to the structure or profiles of the taskgroup. For example, $eu(TG \mid po(a) = 1)$ denotes the expected utility of the taskgroup if $po(a)$ were to equal 1. The above equation, then, specifies by how much the expected utility of the taskgroup would differ if the activity were to definitely succeed versus if it were to definitely fail; we call this *utility difference* (ud). The utility difference is, in general, positive; however, in the presence of disablers, it may be negative. The utility difference can be interpreted as representing two different measures: how much utility the taskgroup would lose if the activity were to fail; or, how much utility the taskgroup would gain if the activity were to earn positive utility. These two interpretations are the bases for the ideas behind expected utility loss and expected utility gain.

To find expected utility loss, we scale ud by $1 - po(a)$; this turns the measure into the utility loss expected to happen based on a's probability of failing. For example, when considering eul we do not care if an activity has a huge utility difference if it will never fail; instead, we would want to prioritize looking at activities with medium utility differences but large probabilities of failing. The overall equation for expected utility loss is:

$$eul(a) = (1 - po(a)) \cdot (eu(TG \mid po(a) = 1) - eu(TG \mid po(a) = 0))$$

Expected utility gain also scales ud, but by a's probability of earning utility, so that it measures the gain in utility expected to happen based on a's probability of earning utility. This ensures that eug prioritizes activities with, for example, a medium utility difference but high probability of earning utility over an activity with a high utility difference but a very low probability of earning utility. The equation for expected utility gain is:

$$eug\,(a) = po\,(a) \cdot (eu\,(TG \mid po\,(a) = 1) - eu\,(TG \mid po\,(a) = 0))$$

Although we do not to so in this work, utility difference can also be used as a metric without considering po values, depending on what the goals of the programmer are. Using ud alone is suitable for systems that want to prioritize, for example, eliminating the possibility of a catastrophic failure, no matter how unlikely; $(1 - po) \cdot ud$ and $po \cdot ud$ are suitable for systems with the goal of maximizing expected utility, as ours. These heuristics are not specific to this domain, as utility difference can be calculated for any activity in any activity network.

Because these calculations ideally require knowledge of the entire activity hierarchy, they are implemented differently depending on whether it is global or distributed. We discuss details of this in the following two sections.

5.3.3 Global Algorithm

When initially invoked, global PADS goes through the activity network and calculates probability profiles for each activity. PADS begins with the method(s) with the earliest `est`(s); it then propagates profiles through the network until profiles for the entire tree have been found. Profiles are stored with each activity, and are updated by PADS if the information associated with upstream (or child) activities changes.

To calculate $eul\,(a)$ and $eug\,(a)$, PADS temporarily sets $po\,(a)$ to be 1 and 0, in turn, and propagates that information throughout the activity network until the $eu\,(TG)$ has been updated and provides the values of $eu\,(TG \mid po\,(a) = 1)$ and $eu\,(TG \mid po\,(a) = 0)$. PADS can then find the utility difference, $ud = eu\,(TG \mid po\,(a) = 1) - eu\,(TG \mid po\,(a) = 0)$, and multiply it by $1 - po\,(a)$ or $po\,(a)$ to find $eul\,(a)$ or $eug\,(a)$, respectively.

Probability Profile Propagation Example

In this section, we show global probability profile calculations for a simple example (Figure 5.3) to give further intuition for how PADS works . The problem that we show is similar to Figures 3.5 and 5.1, but is simplified here to promote understanding; it has deterministic utilities and smaller

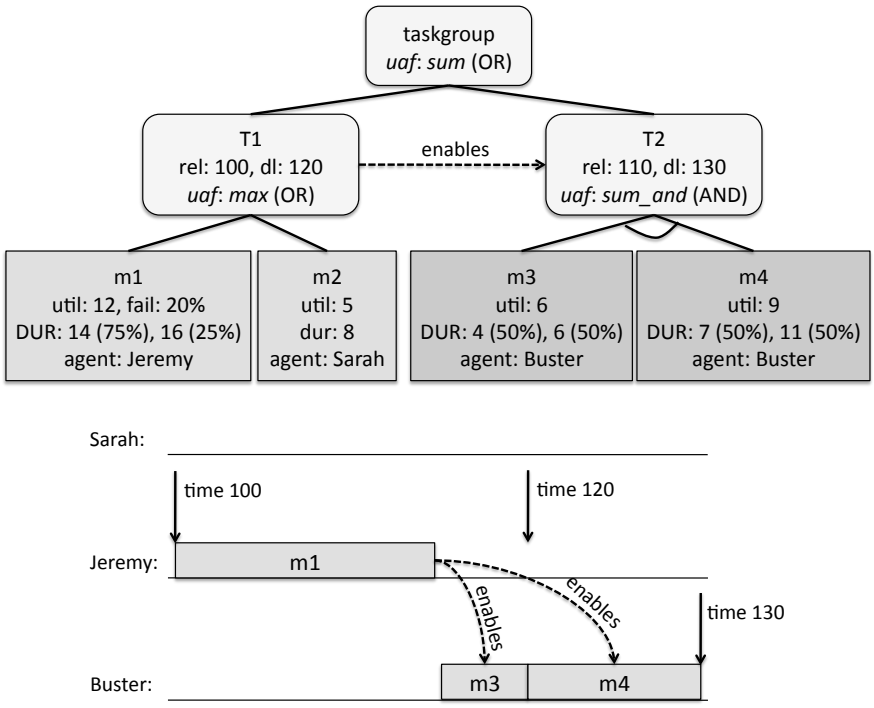


Figure 5.3: A simple 3-agent problem and a corresponding schedule.

duration distributions. We also show the schedule for this problem. The time bounds for this schedule generated by the deterministic STN (and thus based on the methods' average, scheduled durations) are:

- $\texttt{est}\,(m1) = 100, \texttt{lst}\,(m1) = 101, \texttt{eft}\,(m1) = 115, \texttt{lft}\,(m1) = 116$
- $\texttt{est}\,(m3) = 115, \texttt{lst}\,(m3) = 116, \texttt{eft}\,(m3) = 120, \texttt{lft}\,(m3) = 121$
- $\texttt{est}\,(m4) = 120, \texttt{lst}\,(m4) = 121, \texttt{eft}\,(m4) = 129, \texttt{lft}\,(m4) = 130$

Our calculation begins with method $m1$, as it is the earliest scheduled method. It has no enablers and no predecessor, so its start time distribution depends on nothing but its release time (rel). Therefore, since methods are always started as early as possible, $ST\,(m1) = \{\{100, 1.0\}\}$; this implies that $p\,(st\text{-}m1) = 1$ because it never misses its $\texttt{lst}$ of 101. It has no disablers, so $p\,(dis\text{-}m1) = 0$. It also has no soft NLEs, so $NLE_u\,(m1) = \{\{1, 1\}\}$ and $NLE_d\,(m1) = \{\{1, 1\}\}$ for all start times.

The finish time of $m1$ is a combination of $ST\,(m1) = \{\{100, 1.0\}\}$ and its duration distribution, $\{\{14, 0.75\}, \{16, 0.25\}\}$. Iterating through these, PADS first gets $ft = 100 + 14 = 114$ which occurs with $p\,(ft) = 1.0 \cdot 0.75 = 0.75$. This meets $m1$'s deadline of 120. So, PADS pushes $\{114, 0.75 \cdot 0.2\}$ onto $FT_{nu}\,(m1)$ to account for its failure outcome likelihood, and $\{114, 0.75 \cdot 0.8\}$ onto $FT_u\,(m1)$. The method's other possible finish time is $ft = 100 + 16 = 116$ which occurs with $p\,(ft) = 1.0 \cdot 0.25 = 0.25$, which also meets $m1$'s deadline. Again, however, $m1$ might fail so PADS pushes $\{116, 0.25 \cdot 0.2\}$ onto $FT_{nu}\,(m1)$ and $\{116, 0.25 \cdot 0.8\}$ onto $FT_u\,(m1)$. Trivially, $p\,(ft\text{-}m1) = 1$.

The utility event formula for $m1$ is $UF_{ev}\,(m1) = (and\ st\text{-}m1\ ft\text{-}m1\ suc\text{-}m1)$; we omit $dis\text{-}m1$ and $en\text{-}m1$ because $m1$ does not have enablers or disablers. The utility value assignment is $UF_{val}\,(m1) = \{st\text{-}m1 \leftarrow 1,\ ft\text{-}m1 \leftarrow 1,\ suc\text{-}m1 \leftarrow 0.8\}$. To evaluate this, PADS creates different truth value assignments to the variables in $UF_{ev}\,(m1)$ by looking at all possible subsets of the events and assigning "true" to the events in the set and "false" to those not in the set. We list below the different sets, generated by the powerset operator. and how the utility formula evaluates for those sets, in the context of Algorithm 5.8.

1. $S = \{\}$:

$$T = \{st\text{-}m1,\ ft\text{-}m1,\ suc\text{-}m1\}$$
$$p\,(S) = (1 - p\,(st\text{-}m1)) \cdot (1 - p\,(ft\text{-}m1)) \cdot (1 - p\,(suc\text{-}m1)) = 0$$
$$UF_{en}/\forall s \in S \leftarrow true/\forall t \in T \leftarrow false = (and\ false\ false\ false) = false$$

2. $S = \{st\text{-}m1\}$:

$$T = \{ft\text{-}m1,\ suc\text{-}m1\}$$
$$p\,(S) = p\,(st\text{-}m1) \cdot (1 - p\,(ft\text{-}m1)) \cdot (1 - p\,(suc\text{-}m1)) = 0$$
$$UF_{en}/\forall s \in S \leftarrow true/\forall t \in T \leftarrow false = (and\ true\ false\ false) = false$$

3. $S = \{ft\text{-}m1\}$: not shown, evaluates to false
4. $S = \{suc\text{-}m1\}$: not shown, evaluates to false
5. $S = \{st\text{-}m1,\ ft\text{-}m1\}$: not shown, evaluates to false
6. $S = \{st\text{-}m1,\ suc\text{-}m1\}$: not shown, evaluates to false
7. $S = \{ft\text{-}m1,\ suc\text{-}m1\}$: not shown, evaluates to false

8. $S = \{st\text{-}m1,\ ft\text{-}m1,\ suc\text{-}m1\}$:

$$T = \{\}$$
$$p(S) = p(st\text{-}m1) \cdot p(ft\text{-}m1) \cdot p(suc\text{-}m1) = 0.8$$
$$UF_{en}/\forall s \in S \leftarrow true/\forall t \in T \leftarrow false = (and\ true\ true\ true) = true$$
$$po(m1) = po(m1) + 0.8$$

Based on these iterations, $po(m1) = 0.8$. As $util(m1) = 12$ and $NLE_u(m1) = \{\{1, 1\}\}$, $eu(m1) = 0.8 \cdot 12 = 9.6$. To put together $m1$'s probability profile:

$$ST(m1) = \{\{100, 1.0\}\}$$
$$FT_u(m1) = \{\{114, 0.6\}, \{116, 0.2\}\}$$
$$FT_{nu}(m1) = \{\{114, 0.15\}, \{116, 0.05\}\}$$
$$UF_{ev}(m1) = (and\ st\text{-}m1\ ft\text{-}m1\ suc\text{-}m1)$$
$$UF_{val}(m1) = \{st\text{-}m1 \leftarrow 1,\ ft\text{-}m1 \leftarrow 1,\ suc\text{-}m1 \leftarrow 0.8\}$$
$$po(m1) = 0.8$$
$$eu(m1) = 9.6$$

Next we can calculate $T1$'s probability profile. Because it only has one scheduled child, its profile is the same as $m1$'s:

$$FT_u(T1) = \{\{114, 0.6\}, \{116, 0.2\}\}$$
$$FT_{nu}(T1) = \{\{114, 0.15\}, \{116, 0.05\}\}$$
$$UF_{ev}(T1) = (and\ st\text{-}m1\ ft\text{-}m1\ suc\text{-}m1)$$
$$UF_{val}(T1) = \{st\text{-}m1 \leftarrow 1,\ ft\text{-}m1 \leftarrow 1,\ suc\text{-}m1 \leftarrow 0.8\}$$
$$po(T1) = 0.8$$
$$eu(T1) = 9.6$$

Because $m3$ is enabled by $T1$, its start time depends on $T1$'s finish time. The algorithm begins by iterating through the distribution E (Line 1 of Algorithm 5.4), which here equals $FT_u(T1)$ appended by $\{\infty, 1 - po(T1)\}$, so $E = \{\{114, 0.6\}, \{116, 0.2\}, \{\infty, 0.2\}\}$. It can skip Line 4 as $m3$ has no predecessors. First, then, $st = 114$ with probability 0.6. Since this is a valid start time, PADS pushes $\{114, 0.6\}$ onto $ST(m3)$. Next, $st = 116$ with probability 0.2 and PADS pushes $\{116, 0.2\}$ onto $ST(m3)$.

Next, $st = \infty$ with probability 0.2. This represents the case when $T1$ does not earn utility. Because the method is not enabled in this case, the method fails at $\texttt{max}(\texttt{lst}(m3) + 1, ft_{pred}) =$

117; recall that in cases where a method's enabler fails, the method is not removed from the schedule until its `lst` has passed in case the planner is able to reschedule the enabler. So, PADS pushes $\{117, 0.2\}$ onto $FT_{nu}(m3)$. Finally, $p(st\text{-}m3) = 0.8/0.8$ and $p(dis\text{-}m3) = 0$. As $m3$ has no soft NLEs, $NLE_u(m3) = \{\{1, 1\}\}$ and $NLE_d(m3) = \{\{1, 1\}\}$ for all possible start times.

To find $m3$'s finish time distribution, $ST(m3)$ is combined with $\texttt{DUR}(m3)$. The first combination is $st = 114$ and $dur = 4$, which happens with probability $0.6 \cdot 0.5 = 0.3$; the finish time 118 is less than $m3$'s deadline, and so PADS pushes $\{118, 0.3\}$ onto $FT_u(m3)$. The next combination is $st = 114$ and $dur = 6$, which happens with probability $0.6 \cdot 0.5 = 0.3$; the finish time 120 is less than $m3$'s deadline, and so we push $\{120, 0.3\}$ onto $FT_u(m3)$. The next combination is $st = 116$ and $dur = 4$, which happens with probability $0.2 \cdot 0.5 = 0.1$; the finish time 120 is less than $m3$'s deadline, and so PADS pushes $\{120, 0.1\}$ onto $FT_u(m3)$. The final combination is $st = 116$ and $dur = 6$, which happens with probability $0.2 \cdot 0.5 = 0.1$; the finish time 122 is less than $m3$'s deadline, and so PADS pushes $\{122, 0.1\}$ onto $FT_u(m3)$. Trivially, $p(ft\text{-}m3) = 0.8/0.8 = 1$.

The utility event formula for $m3$ is (omitting $dis\text{-}m3$, as $m3$ has no disablers):

$$UF_{ev}(m3) = (and\ (and\ st\text{-}m1\ ft\text{-}m1\ suc\text{-}m1)\ st\text{-}m3\ ft\text{-}m3\ suc\text{-}m3)$$

and the utility value assignment is

$$\begin{aligned} UF_{val}(m3) = \{&st\text{-}m1 \leftarrow 1,\ ft\text{-}m1 \leftarrow 1,\ suc\text{-}m1 \leftarrow 0.8,\\ &st\text{-}m3 \leftarrow 1,\ ft\text{-}m3 \leftarrow 1,\ suc\text{-}m3 \leftarrow 1\} \end{aligned}$$

$UF(m3)$ is evaluated the same as is $m1$'s utility formula. The calculation is not shown, but it evaluates to 0.8, and $m3$'s expected utility is $0.8 \cdot 6 = 4.8$. Thus $m3$'s full profile is:

$$\begin{aligned} ST(m3) &= \{\{114, 0.6\}, \{116, 0.2\}\}\\ FT_u(m3) &= \{\{118, 0.3\}, \{120, 0.4\}, \{122, 0.1\}\}\\ FT_{nu}(m3) &= \{\{117, 0.2\}\}\\ UF_{ev}(m3) &= (and\ (and\ st\text{-}m1\ ft\text{-}m1\ suc\text{-}m1)\ st\text{-}m3\ ft\text{-}m3\ suc\text{-}m3)\\ UF_{val}(m3) &= \{st\text{-}m1 \leftarrow 1,\ ft\text{-}m1 \leftarrow 1,\ suc\text{-}m1 \leftarrow 0.8,\\ &\qquad st\text{-}m3 \leftarrow 1,\ ft\text{-}m3 \leftarrow 1,\ suc\text{-}m3 \leftarrow 1\}\\ po(m3) &= 0.8\\ eu(m3) &= 4.8 \end{aligned}$$

Method $m4$'s profile is calculated next as per Algorithm 5.4. Its start time depends on $T1$'s and $m3$'s finish times. As with method $m4$, the distribution $E = \{\{114, 0.6\}, \{116, 0.2\}, \{\infty, 0.2\}\}$.

PADS iterates through this distribution and $m4$'s predecessor's ($m3$'s) finish time distribution, calculating $m4$'s start time distribution along the way. Recall that $\texttt{lst}\,(m4) = 121$. We collapse the cases where ft_e equals ∞ and do not bother showing the iteration through $m4$'s predecessor's distribution for that case.

1. $ft_e = 114, ft_{pred} = 117, st = 117, p_{st} = 0.6 \cdot 0.2 = 0.12$
   Method starts, push $\{117, 0.12\}$ onto $ST\,(m4)$
2. $ft_e = 114, ft_{pred} = 118, st = 118, p_{st} = 0.6 \cdot 0.3 = 0.18$
   Method starts, push $\{118, 0.18\}$ onto $ST\,(m4)$
3. $ft_e = 114, ft_{pred} = 120, st = 120, p_{st} = 0.6 \cdot 0.4 = 0.24$
   Method starts, push $\{120, 0.24\}$ onto $ST\,(m4)$
4. $ft_e = 114, ft_{pred} = 122, st = 114, p_{st} = 0.6 \cdot 0.1 = 0.06$
   Method misses $\texttt{lst}$, push $\{122, 0.06\}$ onto $FT_{nu}\,(m4)$
5. $ft_e = 116, ft_{pred} = 117, st = 117, p_{st} = 0.2 \cdot 0.2 = 0.04$
   Method starts, push $\{117, 0.04\}$ onto $ST\,(m4)$
6. $ft_e = 116, ft_{pred} = 118, st = 118, p_{st} = 0.2 \cdot 0.3 = 0.06$
   Method starts, push $\{118, 0.06\}$ onto $ST\,(m4)$
7. $ft_e = 116, ft_{pred} = 120, st = 120, p_{st} = 0.2 \cdot 0.4 = 0.08$
   Method starts, push $\{120, 0.08\}$ onto $ST\,(m4)$
8. $ft_e = 116, ft_{pred} = 122, st = 122, p_{st} = 0.2 \cdot 0.1 = 0.02$
   Method misses $\texttt{lst}$, push $\{122, 0.02\}$ onto $FT_{nu}\,(m4)$
9. $ft_e = \infty, st = \infty, p_{st} = 0.2$
   Method is not enabled, push $\{122, 0.2\}$ onto $FT_{nu}\,(m4)$

PADS ultimately ends up with $ST\,(m4) = \{\{117, 0.16\}, \{118, 0.24\}, \{120, 0.32\}\}$. As it is enabled with probability 0.8 but starts with probability 0.72, $p\,(st\text{-}m4) = 0.72/0.8 = 0.9$. As $m4$ has no soft NLEs, $NLE_u\,(m4) = \{\{1, 1\}\}$ and $NLE_d\,(m4) = \{\{1, 1\}\}$.

To find $m4$'s finish time distribution, $ST\,(m4)$ is combined with $\texttt{DUR}\,(m4)$. Since $\texttt{DUR}\,(m4)$ has only two possible values, $m4$ has six possible finish time combinations; we do not show the calculation. The latest one, when $m4$ starts at time 120 and has duration 11, is the only one where $m4$ misses its deadline; so $p\,(ft\text{-}m4) = \frac{0.72-0.16}{0.72} = 0.77778$.

The utility event formula for $m4$ is

$$UF_{ev}\,(m4) = (and\ (and\ st\text{-}m1\ ft\text{-}m1\ suc\text{-}m1)\ st\text{-}m4\ ft\text{-}m4\ suc\text{-}m4)$$

The utility value formula is $UF_{val}\,(m4) = \{st\text{-}m1 \leftarrow 1,\ ft\text{-}m1 \leftarrow 1,\ suc\text{-}m1 \leftarrow 0.8,\ st\text{-}m4 \leftarrow 0.9,\ ft\text{-}m4 \leftarrow 0.77778,\ suc\text{-}m4 \leftarrow 1\}$. We evaluate this as we evaluated $m1$'s utility formula.

The calculation is not shown but it evaluates to 0.56, and $m4$'s expected utility is $0.56 \cdot 9 = 5.04$. Thus $m4$'s full profile is:

$$
\begin{aligned}
ST\,(m4) &= \{\{117, 0.16\}, \{118, 0.24\}, \{120, 0.32\}\} \\
FT_u\,(m4) &= \{\{124, 0.08\}, \{125, 0.12\}, \{127, 0.16\}, \{128, 0.08\}, \{129, 0.12\}\} \\
FT_{nu}\,(m4) &= \{\{122, 0.28\}, \{131, 0.16\}\} \\
UF_{ev}\,(m4) &= (and\ (and\ st\text{-}m1\ ft\text{-}m1\ suc\text{-}m1)\ st\text{-}m4\ ft\text{-}m4\ suc\text{-}m4) \\
UF_{val}\,(m4) &= \{st\text{-}m1 \leftarrow 1,\ ft\text{-}m1 \leftarrow 1,\ suc\text{-}m1 \leftarrow 0.8, \\
&\qquad st\text{-}m4 \leftarrow 0.9,\ ft\text{-}m4 \leftarrow 0.77778,\ suc\text{-}m4 \leftarrow 1\} \\
po\,(m4) &= 0.56 \\
eu\,(m4) &= 5.04
\end{aligned}
$$

$T2$ is calculated next. PADS begins by iterating through the superset of $T2$'s children to consider the possible combinations of $T2$'s children not earning utility, as per Algorithm 5.12. When $L = \{\}$, all children earn utility; PADS finds the distribution representing the `max` of their FT_u distributions and adds it to $FT_u\,(T2)$. Because $m4$ always finishes later, $FT_u\,(T2) = \{\{124, 0.064\}, \{125, 0.096\}, \{127, 0.128\}, \{128, 0.064\}, \{129, 0.096\}\}$ (note that this is the same as $FT_u\,(m4)$ times 0.8, the probability that $m3$ earns utility). When $L = \{m3\}$, $T2$ will fail at the same time $m3$ does and so PADS adds $\{117, 0.2 \cdot 0.56 = 0.112\}$ to $FT_{nu}\,(T2)$. When $L = \{m4\}$, $T2$ will fail at the same time $m4$ does and so PADS adds $\{122, 0.28 \cdot 0.8 = 0.224\}$ and $\{131, 0.16 \cdot 0.8 = 0.128\}$ to $FT_{nu}\,(T2)$. Finally, when $L = \{m3, m4\}$, $T2$ will fail at the same time $m3$ does, as it always fails earlier, and so PADS adds $\{117, 0.2 \cdot 0.44 = 0.088\}$ to $FT_{nu}\,(T2)$.

$UF_{ev}\,(T2)$ equals the and of its children's event formulas, and $UF_{val}\,(T2)$ equals the union of its children's value assignments. $UF\,(T2)$ evaluates to 0.56. Note here the discrepancy between the sum of the members of FT_u and po: $FT_u\,(T2)$ sums to 0.448, but $po\,(T2) = 0.56$. This highlights the difference between assuming that the enablers of $m3$ and $m4$ are independent, as in the FT_u calculation, and not making that assumption, as for po. $T2$'s expected utility, as described in Algorithm 5.13, is $0.56 \cdot (6 + 9) = 8.4$, where 6 and 9 are utilities of $m3$ and $m4$, respectively. The last part of the calculation is to normalize $T2$'s FT distributions so that they sum to $po\,(T2)$ and $1 - po\,(T2)$.

$T2$'s final probability profile is:

$$
\begin{aligned}
FT_u\left(T2\right) &= \{\{124, 0.08\}, \{125, 0.12\}, \{127, 0.16\}, \{128, 0.08\}, \{129, 0.12\}\} \\
FT_{nu}\left(T2\right) &= \{\{117, 0.16\}, \{122, 0.18\}, \{131, 0.1\}\} \\
UF_{ev}\left(T2\right) &= (and\ (and\ (and\ st\text{-}m1\ ft\text{-}m1\ suc\text{-}m1)\ st\text{-}m3\ ft\text{-}m3\ suc\text{-}m3) \\
&\qquad\qquad (and\ (and\ st\text{-}m1\ ft\text{-}m1\ suc\text{-}m1)\ st\text{-}m4\ ft\text{-}m4\ suc\text{-}m4)) \\
UF_{val}\left(T2\right) &= \{st\text{-}m1 \leftarrow 1,\ ft\text{-}m1 \leftarrow 1,\ suc\text{-}m1 \leftarrow 0.8, \\
&\qquad\quad st\text{-}m3 \leftarrow 1,\ ft\text{-}m3 \leftarrow 1,\ suc\text{-}m3 \leftarrow 1, \\
&\qquad\quad st\text{-}m4 \leftarrow 0.9,\ ft\text{-}m4 \leftarrow 0.77778,\ suc\text{-}m4 \leftarrow 1\} \\
po\left(T2\right) &= 0.56 \\
eu\left(T2\right) &= 8.4
\end{aligned}
$$

Finally we can find the taskgroup's probability profile. We first find its finish time distribution as per Algorithm 5.10. In the first iteration of the algorithm, $S = \{\}$, which means that in this hypothetical case no child earns utility. As $T2$'s failure finish time distribution is strictly later than $T1$'s, we can add $FT_{nu}\left(T2\right) \cdot \left(1 - po\left(T1\right)\right)$, or $\{\{117, 0.032\}, \{122, 0.036\}, \{131, 0.02\}\}$, to $FT_{nu}\left(TG\right)$.

Next, $S = \{T1\}$, meaning that in this hypothetical case $T1$ earns utility and $T2$ does not. Therefore, we add $FT_u\left(T1\right) \cdot \left(1 - po\left(T2\right)\right)$, or $\{\{114, 0.264\}, \{116, 0.088\}\}$, to $FT_u\left(TG\right)$. S next equals $\{T2\}$. For the same reasoning, we add $FT_u\left(T2\right) \cdot \left(1 - po\left(T1\right)\right)$, which equals $\{\{124, 0.016\}, \{125, 0.024\}, \{127, 0.032\}, \{128, 0.016\}, \{129, 0.024\}\}$, to $FT_u\left(TG\right)$. Finally, $S = \{T1, T2\}$. As $FT_u\left(T1\right)$ is strictly less than $FT_u\left(T1\right)$, we add $\{\{114, 0.336\}, \{116, 0.112\}\}$ to $FT_u\left(TG\right)$.

$UF_{ev}\left(TG\right)$ equals the *or* of its children's event formulas, and $UF_{val}\left(TG\right)$ equals the union of its children's value assignments. Together, they evaluate to 0.8. Given this value, we can normalize

$FT_u\,(TG)$ and $FT_{nu}\,(TG)$ so they sum to 0.8 and 0.2, respectively. Overall:

$$\begin{aligned}
FT_u\,(TG) &= \{\{114, 0.53\}, \{116, 0.175\}, \{124, 0.014\}, \{125, 0.021\}, \{127, 0.028\},\\
&\quad \{128, 0.014\}, \{129, 0.021\}\}\\
FT_{nu}\,(TG) &= \{\{117, 0.073\}, \{122, 0.082\}, \{131, 0.045\}\}\\
UF_{ev}\,(TG) &= (or\ (and\ st\text{-}m1\ ft\text{-}m1\ suc\text{-}m1)\\
&\qquad (and\ (and\ (and\ st\text{-}m1\ ft\text{-}m1\ suc\text{-}m1)\ st\text{-}m3\ ft\text{-}m3\ suc\text{-}m3)\\
&\qquad\quad (and\ (and\ st\text{-}m1\ ft\text{-}m1\ suc\text{-}m1)\ st\text{-}m4\ ft\text{-}m4\ suc\text{-}m4)))\\
UF_{val}\,(TG) &= \{st\text{-}m1 \leftarrow 1,\ ft\text{-}m1 \leftarrow 1,\ suc\text{-}m1 \leftarrow 0.8,\\
&\quad st\text{-}m3 \leftarrow 1,\ ft\text{-}m3 \leftarrow 1,\ suc\text{-}m3 \leftarrow 1,\\
&\quad st\text{-}m4 \leftarrow 0.9,\ ft\text{-}m4 \leftarrow 0.77778,\ suc\text{-}m4 \leftarrow 1\}\\
po\,(TG) &= 0.8\\
eu\,(TG) &= 18
\end{aligned}$$

Therefore the probability that the schedule with earn positive utility during execution is 0.8, and its expected utility is 18.

5.3.4 Distributed Algorithm

The distributed version of PADS (D-PADS) is used by agents locally during execution. It is intuitively very similar to global PADS. In fact, the algorithms for calculating activities start times, NLE information, finish times and expected utility are for the large part the same. However, not all activity profiles can be found in a single pass, as no agent has a complete view of the activity network. Recall from Section 3.3.2 (on page 36) that when execution begins, agents have a fixed period of time to perform initial computations, such as calculating initial probability profiles, before they actually begin to execute methods. When execution begins, during this time, the probability profiles of the activity network are computed incrementally, with each agent calculating as many profiles as it can at a time. Agents then pass on the newly-found profiles to other agents in order to allow them to calculate further portions of the network. This process repeats until the profiles of the entire network have been found.

Once execution begins and methods begin executing, probability profiles are continuously and asynchronously being updated to reflect the stochasticity of the domain and the evolving schedule. Whenever an agent receives updates concerning an activity's execution, its enablers, disablers, facilitators, hinderers, or (for tasks) children, it updates that activity's profile; if the activity's profile does in fact change, D-PADS passes that information on to other agents interested in that activity.

Agents do not explicitly wait for or keep track of the effects these changes have on the rest of the hierarchy, even though changes resulting from their update may eventually propagate back to them; in a domain with high executional uncertainty such as this one, doing so would be very difficult because typically many probability profiles are changing concurrently.

The run-time aspect of the approach also necessitates an additional notion of the $current\text{-}time$, which is used as a lower bound on when a method will start (assuming its execution has not already begun); similarly, after an activity has been executed, its profile is changed to the actual outcome of its execution. Since handling these issues involve very small changes to the PADS algorithms presented in Section 5.3.2, we will not explicitly show the changes needed to those algorithms.

The distributed version, however, does make several approximations of expected utility at the taskgroup level. For example, in the global version, while calculating ud, which is necessary for eul and eug, $eu(TG \mid po(a) = 1)$ and $eu(TG \mid po(a) = 0)$ can be calculated exactly by setting $po(a)$ to the appropriate value and updating the entire network. In the distributed case, a different approach must be taken since no agent has a complete view of the network.

There are three options for calculating ud in a distributed way. The first is to exactly calculate such values on demand. This involves agents querying other agents for information to use in their calculations, which would introduce a large delay in the calculation. The second option, which can also calculate the values exactly, avoids this delay by always having such values cached. This requires every agent to calculate, for each of its activities, how the success or failure of every other activity in the network would affect it, and communicate that to other agents. Such computations would take long to compute due to the large number of contingencies to explore, and would also require a large increase in the amount of communication; overall, therefore, we consider them intractable given the real-time constraints of the domain.

Instead, we utilize a third option in which these values are estimated locally by agents as needed. During D-PADS, agents estimate ud based on an estimation of the expected utility of the taskgroup if an activity a were to have a hypothetical probability value h of earning utility, $\widehat{eu_{TG}}(a, h)$; we use the $\widehat{\ }$ symbol to designate a number as an estimate.

The algorithm for calculating $\widehat{eu_{TG}}$ is shown in Algorithm 5.14. The algorithm begins at some local activity a and ascends the activity hierarchy one parent at a time until it reaches the taskgroup (Line 5). At each level it estimates the po and eu that the current task T would have if $po(a) = h$, notated as $\widehat{po}(T \mid po(a) = h)$ and $\widehat{eu}(T \mid po(a) = h)$. To begin, $\widehat{po}(a \mid po(a) = h) = h$ and $\widehat{eu}(a \mid po(a) = h) = h \cdot \texttt{averageUtil}(a)$ (where `averageUtil` is defined in Section 3.3.2) (Lines 1-2). At higher levels the algorithm estimates these numbers according to tasks' *uafs*. As a simple example, if a's parent is a *max* task T, its estimated expected utility is the maximum of a's new estimated expected utility and the expected utility of T's other children (Lines 20-21).

Algorithm 5.14: Distributed taskgroup expected utility approximation with enablers and *uafs*.

Function : $\texttt{approximateEU}\,(a, h)$

$\widehat{po}\,(a \mid po\,(a) = h) = h$
$\widehat{eu}\,(a \mid po\,(a) = h) = h \cdot \texttt{averageUtil}\,(a)$
$c = a$
$\widehat{nle_\Delta} = 0$
while $T = \texttt{parent}\,(c)$ **do**
  foreach $c_e \in \texttt{enabled}\,(c)$ **do**
    $\widehat{po}\,(c_e \mid \hat{po}\,(c)) = po\,(c_e) \cdot \frac{\widehat{po}(c|po(a)=h)}{po(c)}$
    $\widehat{nle_\Delta} = \widehat{nle_\Delta} + \texttt{approximateEU}\,(c_e, \widehat{po}\,(c_e \mid \hat{po}\,(c)))$
  end
  foreach $c_d \in \texttt{disabled}\,(c)$ **do**
    $\widehat{po}\,(c_d \mid \hat{po}\,(c)) = po\,(c_d) \cdot \frac{1-\widehat{po}(c|po(a)=h)}{1-po(c)}$
    $\widehat{nle_\Delta} = \widehat{nle_\Delta} + \texttt{approximateEU}\,(c_d, \widehat{po}\,(c_d \mid \hat{po}\,(c)))$
  end
  $children' = \texttt{scheduledChildren}\,(T) - \{c\}$
  switch *Type of task* T **do**
    case *sum*
      $\widehat{po}\,(T \mid po\,(a) = h) = 1 - \left((1 - po\,(T)) \cdot \frac{1-\widehat{po}(c|po(a)=h)}{1-po(c)}\right)$
      $\widehat{eu}\,(T \mid po\,(a) = h) = \widehat{eu}\,(c \mid po\,(a) = h) + \sum_{c' \in children'} eu\,(c')$
    case *max*
      $\widehat{po}\,(T \mid po\,(a) = h) = 1 - \left((1 - po\,(T)) \cdot \frac{1-\widehat{po}(c|po(a)=h)}{1-po(c)}\right)$
      $\widehat{eu}\,(T \mid po\,(a) = h) = \texttt{max}\,(\widehat{eu}\,(c \mid po\,(a) = h), \texttt{max}_{c' \in children'} eu\,(c'))$
    case *exactly_one*
      $\widehat{po}\,(T \mid po\,(a) = h) = \widehat{po}\,(c \mid po\,(a) = h)$
      $\widehat{eu}\,(T \mid po\,(a) = h) = \widehat{eu}\,(c \mid po\,(a) = h)$
    case *sum_and*
      $\widehat{po}\,(T \mid po\,(a) = h) = po\,(T) \cdot \frac{\widehat{po}(c|po(a)=h)}{po(c)}$
      $\widehat{eu}\,(T \mid po\,(a) = h) = \widehat{po}\,(T \mid po\,(a) = h) \cdot \left(\frac{eu(c)}{\widehat{po}(c|po(a)=h)} + \sum_{c' \in children'} \frac{eu(c')}{po(c')}\right)$
    case *min*
      $\widehat{po}\,(T \mid po\,(a) = h) = po\,(T) \cdot \frac{\widehat{po}(c|po(a)=h)}{po(c)}$
      $\widehat{eu}\,(T \mid po\,(a) = h) = \widehat{po}\,(T \mid po\,(a) = h) \cdot \texttt{min}\left(\frac{eu(c)}{\widehat{po}(c|po(a)=h)}, \texttt{min}_{c' \in children'} \frac{eu(c')}{po(c')}\right)$
    end
  end
  $c = T$
end
return $\widehat{eu}\,(TG \mid po\,(a) = h) + \widehat{nle_\Delta}$

Enabling and disabling relationships are also included in the estimation. If an activity c is reached that enables another activity c_e, c_e's probability of earning utility given $\widehat{po}\,(c \mid po(a) = h)$ is estimated. We abbreviate this for clarity as $\widehat{po}\,(c_e \mid \hat{po}\,(c))$. How this hypothetical po value for c_e would impact the overall taskgroup utility, $\widehat{eu_{TG}}\,(c_e, \widehat{po}\,(c_e \mid \hat{po}\,(c)))$, is then estimated by a recursive call (Line 8). The same process happens for disablers. We do not include facilitating or hindering relationships as they do not affect whether a method earns utility, and we expected little benefit in including them in the estimation. Our experimental results, showing the accuracy of this approximation, support this decision (Section 5.4.2). Overall, the algorithm returns the sum of the estimated taskgroup utilities (Line 35).

This algorithm, if it recurses, clearly returns a value not directly comparable to $eu\,(TG)$ as it includes potentially multiple summands of it (Lines 8, 35). Instead, it is meant to be compared to another call to the estimation algorithm. So, to estimate ud, agents use this algorithm twice, with $h = 1$ and $h = 0$:

$$ud = \widehat{eu_{TG}}\,(a, 1) - \widehat{eu_{TG}}\,(a, 0)$$

This value can then be used to provide estimates of eul and eug.

Typically, the estimate the calculation provides is good. The main source of inaccuracy is in cases where an activity c and its enabler would separately cause the same task to lose utility if they fail. For example, if c and an activity it enables c_e were crucial to the success of a common ancestral task, both the core algorithm as well as the recursive call would return a high utility difference stemming from that common ancestor. As we sum these values together, we end up double-counting the utility difference at this task and the overall estimate is higher than it should be. Ultimately, however, we did not deem this a critical error, as even though we are overestimating the impact c's failure would have on the schedule, in such cases it typically has a high impact, anyway.

5.3.5 Practical Considerations

When combining discrete distributions, the potential exists for generating large distributions: the size of a distribution $Z = X + Y$, for example, is bounded by $|X| + |Y|$. As described in Section 3.3.2 (on page 44), however, typically such a distribution has duplicate values which we collapse to reduce its size (*i.e.*, if the distribution already contained an entry $\{5, 0.25\}$ and PADS adds $\{5, 0.2\}$ to it, the resulting distribution is $\{5, 0.45\}$). In part because of this, we did not find distribution size to be a problem. Even so, we demonstrate how the size can be further reduced by eliminating very unlikely outcomes. We found that eliminating outcomes that occur with less than a 1% probability worked well for us; however, depending on what point in the trade-off of speed

versus accuracy is needed for a given domain, that threshold can be adjusted. The elimination did not much impair the calculation's accuracy, as is shown in the next section, and can readily be done for systems which are negatively affected by the distribution sizes.

We also memoized the results of function calls to `evaluateUtilityFormula`. This was found to greatly reduce the memory usage and increase the speed of updating probability profiles. It is effective because many activities have identical formulas or portions of formulas (*e.g.*, a parent task with only one scheduled child), and because many activities' profiles do not change over time.

In the distributed version of PADS, our approach has the potential to increase communication costs as compared to a system not using PADS, as more information is being exchanged. One simple way to decrease this effect is to communicate only those changes in probability profiles that are "large enough" to be important, based on a threshold value. We discuss other ways to reduce communication in our discussion of future work, Section 8.1.1.

5.4 Experiments and Results

We performed experiments in the single-agent domain to test the speed of PADS, as well as in the multi-agent domain to test the speed and accuracy of both the global and distributed versions of PADS.

5.4.1 Single-Agent Domain

We performed an experiment to test the speed of PADS in this domain. We used as our test problems instances PSP94, PSP100 and PSP107 of the mm-j20 benchmark suite of the resource-relaxed Multi-Mode Resource Constrained Project Scheduling Problem with Minimal and Maximal Time Lags (MRCPSP/max) [Schwindt, 1998]. The problems were augmented as described in Section 3.3.1.

In the experiment, we executed the three problems 250 times each, using PADS to calculate and maintain all activity probability profiles, and timed all PADS calculations. To execute the problems, we implemented a simple simulator. Execution starts at time zero, and methods are started at their `est`. To simulate method execution the simulator picks a random duration for the method based on its duration distribution, rounds the duration to the nearest integer, advances time by that duration, and updates the STN as appropriate. The planner is invoked both initially, to generate an initial plan, and then during execution whenever a method finishes at a time that caused an inconsistency in the STN. PADS is invoked to analyze the initial plan before execution begins, as well as to update

Table 5.1: Average time spent by PADS calculating and updating probability profiles while executing schedules in the single-agent domain.

	average PADS time (s)	standard deviation PADS time (s)	average makespan (s)
PSP94	8.65	2.25	130.99
PSP100	14.82	7.54	140.94
PSP107	6.43	1.68	115.41

probability profiles when execution results become known and when replanning occurs.

Table 5.1 shows the results. For each problem, it shows the average time taken to perform all PADS calculations, as well as the corresponding standard deviation. It also shows the average executed schedule makespan. All values are shown in seconds. Clearly, PADS incurs only a modest time cost in this domain, incurring an average overhead of 10.0 seconds, or 7.6% of the duration of execution.

5.4.2 Multi-Agent Domain

Test Suite

For these experiments, we used a test suite consisting of 20 problems taken from the Phase II evaluation test problem suite of the DARPA Coordinators program[1]. In this test suite, methods have deterministic utility (before the effects of soft NLEs), which promotes consistency in the utility of the results. Method durations were, in general, 3-point distributions. Table 5.2 shows the main features of each of the problems, roughly ordered by size. In general, three problems had 10 agents, 20 problems had 20 agents, and two problems had 25 agents. The problems had, on average, between 200 and 300 methods in the activity network, and each method had, on average, 2.4 possible execution outcomes. The table also shows how many max-leaf tasks, tasks overall, and NLEs of each type were present in the activity network. Recall that "max-leaf" tasks are *max* tasks that are the parents of methods. Typically, a deterministic planner would choose only one child of a max-leaf task to schedule. We include this information here because we also use this test suite in our probabilistic schedule strengthening experiments in the following chapter. In general, 20-30% of the total methods in the activity network were scheduled in the initial schedule of these problems, or roughly 50-90 methods.

[1] http://www.darpa.mil/ipto/programs/coor/coor.asp

Table 5.2: Characteristics of the 20 DARPA Coordinators test suite problems.

	agents	**methods**	**avg # method outcomes**	**max-leafs**	**tasks**	**enable NLEs**	**disable NLEs**	**facilitate NLEs**	**hinder NLEs**
1	10	189	3	63	89	18	12	0	18
2	10	225	2.4	45	67	0	0	13	0
3	10	300	2.66	105	140	50	36	33	33
4	20	126	2.95	42	59	21	2	13	10
5	20	135	2.67	52	64	22	0	11	0
6	20	135	2.67	45	64	23	0	12	0
7	20	198	2.67	66	93	38	12	21	10
8	20	225	2.64	75	105	43	10	28	8
9	20	225	5	75	106	23	12	14	11
10	20	225	5	75	106	23	12	20	19
11	20	225	5	75	106	28	12	8	13
12	20	225	5	75	106	29	13	10	13
13	20	225	5	75	106	33	11	15	17
14	20	384	1.55	96	137	33	10	20	17
15	20	384	1.58	96	137	38	23	16	15
16	20	384	1.59	96	137	32	18	17	16
17	20	384	1.59	96	137	35	23	19	14
18	20	384	1.62	96	137	47	16	19	23
19	25	460	3.01	115	146	106	28	42	47
20	25	589	3.06	153	194	81	12	60	26

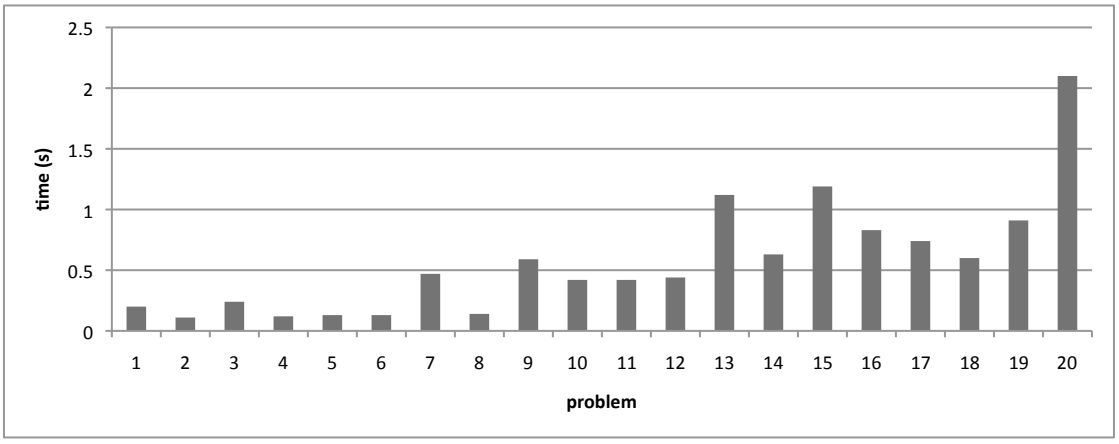


Figure 5.4: Time spent calculating 20 schedules' initial expected utilities.

Analysis of Multi-Agent Global PADS

We performed two experiments in which we characterized the running time and accuracy of global PADS. For each problem in the test suite, we timed the calculation of the expected utility of the initial schedule, using a 3.6 GHz Intel computer with 2GB RAM. Figure 5.4 shows how long it took to calculate the initial expected utility for the entire activity network for each of the 20 problems. The problems are numbered as presented in Table 5.2, and so their order is roughly correlated to their size. The time is very fast, with the largest problem taking 2.1 seconds, and well within acceptable range.

Next, we ran an experiment to characterize the accuracy of the calculation. We executed the initial schedules of each of the 20 test suite problems in a distributed execution environment to see what utility was actually earned by executing the schedules. Each problem in the test suite was run 25 times. In order to directly compare the calculated expected utility with the utility the schedule earns during execution, we executed the schedules as is, with no replanning or strengthening allowed. The schedule's flexible times representation does, however, allow the time bounds of methods to float in response to execution dynamics, and we did retain basic conflict-resolution functionality that naively repairs inconsistencies in the STN, such as removing methods from a schedule if they miss their `lst`.

In the distributed execution environment, when execution begins, agents are given their local view of the activity network and their portion of the pre-computed initial joint schedule, and are left to manage execution of their local schedules while trying to work together to maximize overall

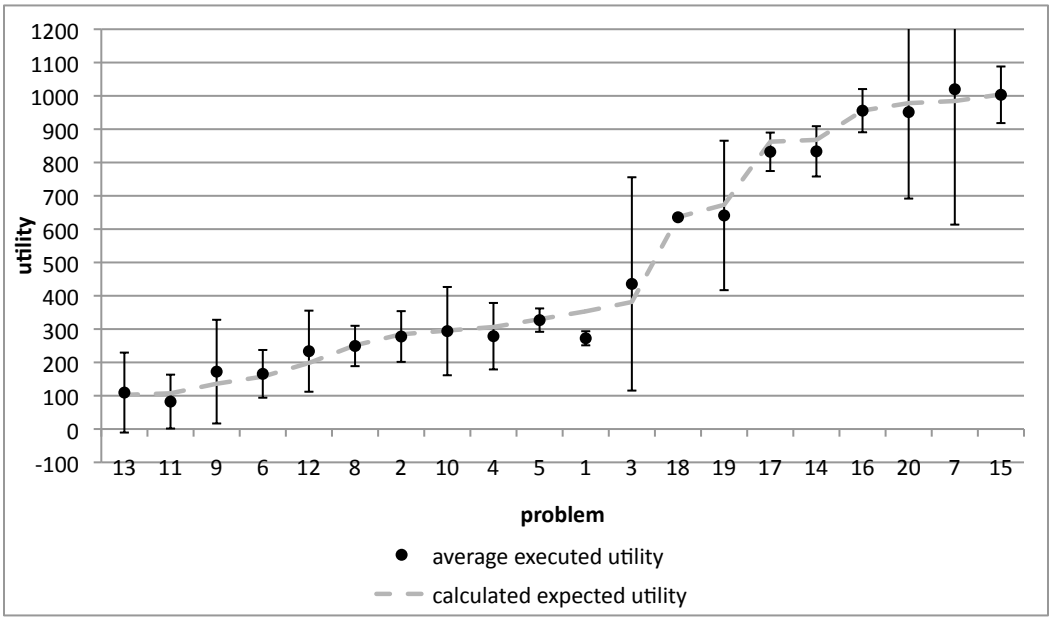


Figure 5.5: The accuracy of global PADS: the calculated expected utility compared to the average utility, when executed.

joint schedule utility. They communicate method start commands to the simulator, receive back execution results (based on the stochastics of the problem) as they become known, and share status information asynchronously. Each agent and the simulator run on their own processor and communicate via message passing. We assume communication is perfect and sufficient bandwidth is available.

Figure 5.5 shows the experimental results, and compares the calculated expected utility with the average utility that was actually earned during execution. The problems are numbered as presented in Table 5.2, and for visual clarity are sorted in the graph by their calculated expected utility. Error bars indicate the standard deviation of utility earned over the 25 execution trials.

Note that although the calculations are close to the average achieved utilities, they are not exactly the same. In addition to assumptions we make that affect the accuracy (Section 5.3.1), there are sources of inaccuracy due to aspects of the problem that we do not model. For example, methods may not start as early as possible (at their `est`) because of the asychronousness of execution; this accounts in large part for the inaccuracy of problem 3. Also, very rarely, when a method misses its `lst` it is left on the schedule and, instead, a later method is removed to increase schedule slack,

going against our assumption that the method that misses its `lst` is always removed. This occurs during the execution of problem 1, accounting for most of its inaccuracy.

Despite these differences, only 4 of the problems - 1, 14, 17, and 18 - have average executed utilities that are statistically different from the corresponding calculated expected utility. Problem 1 we have already discussed. Each of the other problems' 25 simulated executed utilities consist solely of two different values; this occurs, for example, when there is one high-level (and high utility) task that has a non-zero probability of failing but all others always succeed. In cases such as these, a slight error in the calculation of the probability of that task earns utility can lead to a reasonable error in the calculated expected utility, and this is what happens in for these three problems. Overall, however, as the results show, our PADS calculation is very accurate and there is an average error of only 7% between the calculations of the expected utility and the utility earned during execution for these schedules.

Analysis of Multi-Agent Distributed PADS

We also performed experiments to characterize the run-time and accuracy of D-PADS. Recall that when execution begins, agents have a fixed period of time to perform initial computations, such as calculating initial probability profiles, before they actually begin to execute methods. The first experiment measured how much time, during this period, agents spent on PADS from when they received their initial schedule until all probability profiles for the entire activity network were calculated. The experiment was performed on the 20 problem test suite described above. Agent threads were distributed among 10 3.6 GHz Intel computers, each with 2GB RAM. Figure 5.6 shows the average computation time each agent spent distributively calculating the initial expected utilities for each problem, as well as the total computation time spent across agents calculating profiles for that problem. On average, the clock time that passed between when execution began and when all profiles were finished being calculated was 4.9 seconds. In this figure, the problems are numbered as presented in Table 5.2.

Comparing this graph with Figure 5.4, we see that distributing the PADS algorithm does incur some overhead. This is due to probability profiles on an agent needing to be updated multiple times as profile information arrives asynchronously from other agents and affects already-calculated profiles. Even considering this, however, the average time spent per agent calculating probability profiles is quite low, and suitable for a run-time algorithm.

Once agents begin executing methods and execution progresses, the workload in general is not as high as these results, since profiles are only updated on an as-needed basis. On average across these problems, once method execution has begun, individual PADS updates take an average of

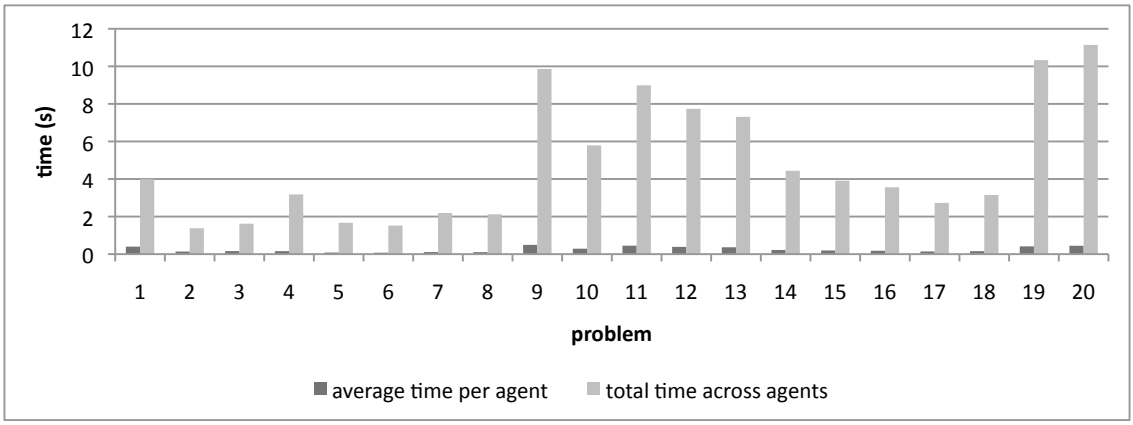


Figure 5.6: Time spent distributively calculating 20 schedule's initial expected utility; shown as both the average time per agent and total time across all agents.

0.003 seconds (with standard deviation $\sigma = 0.01$). Overall, each agent spends an average of 1.5 seconds updating probabilities. In contrast, each individual replan takes an average of 0.309 seconds ($\sigma = 0.39$) with agents spending an average of 47.7 seconds replanning. Thus the overhead of distributed PADS is insignificant when compared to the cost of replanning.

Because there are no approximations in our initial calculation of the probability profiles of the activity tree, we do not duplicate our accuracy results for D-PADS. We did, however, perform an experiment which tested the accuracy of our utility difference approximation. For each method scheduled for each of the 20 test suite problems, we calculated the utility difference exactly (using the method described in Section 5.3.3), and approximately (using the approximation algorithm described in Section 5.3.4). Overall, there were 1413 methods used in the experiment. A repeated measures ANOVA showed a weakly significant difference between the two calculations with $p < 0.07$; however, the utility difference approximations were, on average, within 4.58% of the actual utility difference.

Chapter 6

Probabilistic Schedule Strengthening

This chapter discusses how to effectively strengthen schedules by utilizing the information provided by PADS. The next section describes the approach at a high level. The two sections following that provide detailed descriptions of the algorithms, and can be skipped by the casual reader. Finally, Section 6.4 shows probabilistic schedule strengthening experimental results.

6.1 Overview

The goal of probabilistic schedule strengthening is to make improvements to the most vulnerable portions of the schedule. It relies on PADS to *target*, or focus strengthening efforts on, "at-risk" activities, or activities in danger of failing, whose failure impacts taskgroup utility the most. It then takes steps to strengthen the schedule by protecting those activities against failure. Probabilistic schedule strengthening also utilizes PADS to ensure that all strengthening actions raise the taskgroup expected utility. We implemented probabilistic schedule strengthening for the multi-agent domain [Hiatt et al., 2008, 2009]. In this domain, schedule strengthening is performed both on the initial schedule before it is distributed to agents, and during distributed execution in conjunction with replanning.

At a high level, schedule strengthening works within the structure of the current schedule to improve its robustness and minimize the number of activity failures that arise during execution. In this way is somewhat dependent on the goodness of the schedules generated by the deterministic planner. It is most effective if domain knowledge can be taken advantage of to find the best ways to make the schedule most robust. In the multi-agent domain, for example, a common structural feature of the activity networks is to have *max* tasks as the direct parents of several leaf node

methods. These sibling methods are typically of different durations and utilities, and often belong to different agents. For example, a search and rescue domain might have "Restore Power" modeled as a *max* task, with the option of fully restoring power to a town (higher utility and longer duration) or restoring power only to priority buildings (lower utility and shorter duration). We refer to these *max* tasks as "max-leafs." Typically, a deterministic planner would choose only one child of a max-leaf task to schedule. Probabilistic schedule strengthening, in contrast, might schedule an additional child, or substitute one child for another, in order to make the schedule more robust.

Given our knowledge of the multi-agent domain's characteristics, such as the presence of max-leaf tasks, we have investigated three strengthening tactics for this domain:

1. "Backfilling" - Scheduling redundant methods under a max-leaf task to improve the task's likelihood of earning utility (recall that *max* tasks accrue the utility of the highest utility child that executes successfully). This decreases schedule slack, however, which may lead to a loss of utility.

2. "Swapping" - Replacing a scheduled method with a currently unscheduled sibling method that has a lower *a priori* failure likelihood, a shorter duration or a different set of NLEs[1]. This potentially increases the parent's probability of earning utility by lowering the likelihood of a failure outcome, or potentially increases overall schedule slack and the likelihood that this (or another) method earns utility. Swapping may lead to a loss of utility, however, as the sibling may have a lower average utility, or a set of NLEs that is less desirable.

3. "Pruning" - Unscheduling a method to increase overall schedule slack, possibly strengthening the schedule by increasing the likelihood that remaining methods earn utility. This can also lead to utility loss, however, as the removed method will not earn utility.

Note that each of these tactics has an associated trade-off. PADS allows us to explicitly address this trade-off to ensure that all schedule modifications improve the taskgroup expected utility. *A priori*, we expect backfilling to improve expected utility the most, as it preserves the higher-utility option while still providing a back-up in case that higher-utility option fails. Swapping is expected to be the next most useful, as the task remains scheduled (although possibly at a lower utility). Pruning is expected to be the weakest individual tactic. Our results (Section 6.4.2) support these conjectures.

Activities to strengthen are identified and prioritized differently for the three tactics. For backfilling, max-leaf tasks are targeted which have a non-zero likelihood of failing, and methods are

[1]The reason to consider siblings with a different set of NLEs is because they may change the effective time bounds of the method (via different hard NLEs) or its effective duration (via soft NLEs).

added to reduce this failure probability. Parent tasks are prioritized for backfilling by their expected utility loss (Section 5.3.2, on page 84). Recall that this metric specifies by how much utility the overall taskgroup is expected to suffer because of a task earning zero utility. As schedules can support a limited amount of backfilling before they become too saturated with methods, prioritizing the tasks in this way ensures that tasks whose expected failure affects taskgroup utility the most are strengthened first and have the best chance of being successfully backfilled. While this does not always result in optimal behavior (for example, if backfilling the activity with the highest *eul* prevents backfilling two subsequent ones, which could have led to a higher overall expected utility), we found it to be an effective heuristic in prioritizing activities for strengthening.

For both swapping and pruning, "chains" of methods are first identified where methods are scheduled consecutively and threaten to cause a method(s) to miss its deadline. Methods within the chain are sorted by increasing expected utility gain (Section 5.3.2). Recall that this metric specifies by how much utility the overall taskgroup is expected to gain by a task earning positive utility. Sorting methods in this way ensures that low-importance methods, that do not contribute much to the schedule, are removed or swapped first to try increase overall schedule slack. The chains themselves are prioritized by decreasing expected utility loss so that the most important potential problems are solved first. Methods with non-zero likelihoods of a failure outcome are also considered candidates for swapping; such methods are identified and incorporated into the list of chains, above. As with backfilling, using the *eul* and *eug* heuristics are not guaranteed to result in optimal behavior, but we found them to be effective when used with our swapping and pruning tactics.

Every time that a tactic attempts to modify the schedule, it takes a *step*. A step of any tactic begins with sorting the appropriate activities in the ways specified above. The step considers strengthening each activity sequentially until a valid strengthening action is made. An action is valid if (1) it is successful (*i.e.*, a method is successfully scheduled, swapped or unscheduled) and (2) it raises the taskgroup's expected utility. When a valid action is found, the step returns without trying to modify the schedule further. A step will also return if no valid action was found after all possible actions were considered (*e.g.*, all max-leaf tasks have been looked at but no valid backfill is found). The main computational cost of schedule strengthening comes from updating probability profiles while performing strengthening during these steps.

Given the three tactics introduced above, we next address the question of the order in which they should be applied. The ordering of tactics can affect the effectiveness of probabilistic schedule strengthening because the tactics are not independent: performing a step of one tactic, such as unscheduling a method, may make another tactic, such as swapping that method, infeasible. We call an approach to determining how tactics can be applied to perform probabilistic schedule strength-

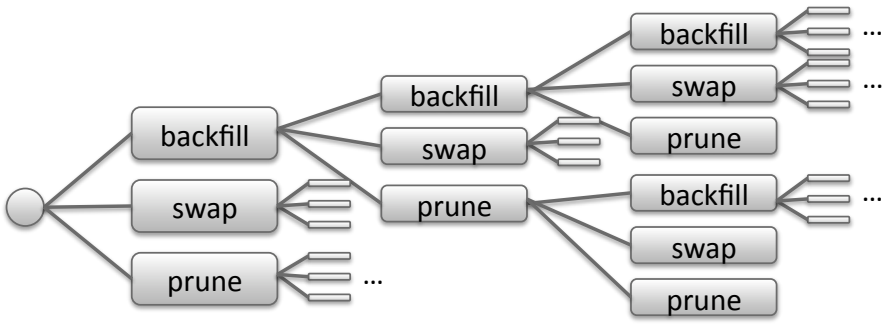


Figure 6.1: The baseline strengthening strategy explores the full search space of the different orderings of backfills, swapping and pruning steps that can be taken to strengthen a schedule.

ening a *strategy*. The baseline "pseudo-optimal" strategy that finds the best possible combination of these actions considers each possible order of backfilling, swapping and pruning steps, finds the taskgroup expected utility that results from each ordering, and returns the sequence of steps that results in the highest expected utility (Figure 6.1). This strategy, however, can take an unmanageable amount of time as it is exponential in the number of possible strengthening actions. Although its practical use is infeasible, we use it as a baseline to compare more efficient strategies against.

The high-level strategy we use is a variant of a round-robin strategy. Agents perform one backfill step, one swap step and one prune step, in that order, repeating until no more actions are possible. This strategy is illustrated in Figure 6.2. Two additional heuristics are incorporated in order to improve the strategy. The first prevents swapping methods for lower-utility methods (to lower the *a priori* failure likelihood) while backfilling the parent max-leaf task is still possible. The second prevents pruning methods that are a task's sole scheduled child, and thus pruning entire branches of the taskgroup, when a backfill or swap is still possible. These heuristics help to prioritize the tactics in order of their individual effectiveness, leading to a more effective probabilistic schedule strengthening strategy. If other strengthening actions were being used, analogous heuristics could be used to prioritize the new set in its order of effectiveness. Although we found these heuristics to be very effective at increasing the amount of expected utility gained by performing round-robin schedule strengthening, they are not guaranteed to be optimal. For example, if a backfill was performed instead of a prune to bolster the expected utility of a task, it is possible that then backfilling some other set tasks may not be possible, which may have led to a higher overall expected utility.

We also experimented with a greedy heuristic variant that performs one "trial" step of each tactic, and commits to the step that raises the expected utility the most, repeating until no more

Figure 6.2: The round-robin strengthening strategy performs one backfill, one swap and one prune, repeating until no more strengthening actions are possible.

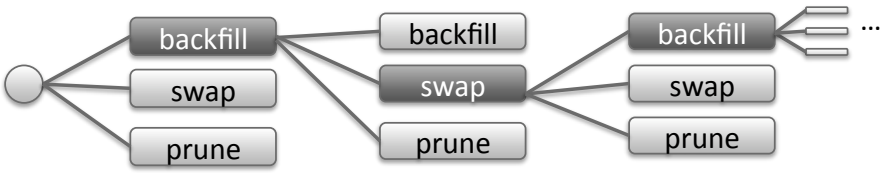


Figure 6.3: The greedy strengthening strategy repeatedly performs one trial backfill, one trial swap and one trial prune, and commits at each point to the action that raises the expected utility of the joint schedule the most.

steps are possible (Figure 6.3). It utilizes the same two heuristics as the round-robin strategy. It performs, in general, roughly as well as the round-robin strategy, but with a higher running time, as for each successful strengthening action it performs up to three trial actions; the round-robin strategy, on the other hand, only performs the one.

During distributed execution, probabilistic schedule strengthening and replanning work well when used together, as they ensure that the current schedule takes advantage of opportunities that arise while also being robust to execution uncertainty; ideally, therefore, replanning would not be performed without being followed by probabilistic schedule strengthening. Schedule strengthening may also be useful when performed alone if the schedule has changed since the last time it was performed, as the changes may have made it possible to strengthen activities that could not be strengthened before, or they may have weakened new activities that could be helped by strengthening.

Agents operate independently and asynchronously during execution. This means that actions such as replanning and probabilistic schedule strengthening are also performed asynchronously. Coordination is required, therefore, to prevent agents from simultaneously performing duplicate actions needlessly and/or undesirably (*e.g.*, two agents backfilling the same task when one backfill would suffice). To address this, agents communicate a "strengthening status" for each local method indicating whether it is scheduled, whether it is unscheduled and the agent has not tried to backfill it yet, or whether the agent has tried to backfill it but the backfill action was invalid. Agents use this

information to reduce these unwanted situations and to inform the two heuristics of round-robin strengthening described above.

Recall also that, as agents make changes to their own probability profiles, they do not explicitly wait for or keep track of the effects these changes have on the rest of the hierarchy, even though changes resulting from their update may eventually propagate back to them. In a domain with high executional uncertainty such as this one, doing so would be very difficult because typically many probability profiles are changing concurrently. Instead, agents replan and strengthen with as complete information as they have. Although this of course has the potential to lead to suboptimal behavior, this situation is alleviated since agents have the option of replanning and strengthening again later if their actions became unsuitable for the updated situation.

6.2 Tactics

The next three sections describe the tactics in detail, for both the global and distributed versions of our multi-agent domain.

6.2.1 Backfilling

Global

Algorithm 6.1 shows pseudocode for a step of the global backfilling tactic. The step performs one backfill, if possible, and then returns. It begins by sorting all scheduled max-leaf tasks T where $po\,(T) < 1$ by $eul\,(T)$ (Line 2), targeting improvements at max-leaf tasks which might fail. We define the function `maxLeafs` to return scheduled max-leaf tasks. Additionally, in this and subsequent algorithms we assume `sort` takes as its arguments the set of activities to sort, the sort order, and the field of the activity to sort by. Sorting in this way ensures that tasks whose expected failure leads to the highest utility loss are strengthened first.

Once the max-leaf tasks are sorted, the step stores the original $eu\,(TG)$ value (Line 3) to use later for comparison. Then, each child of each task is sequentially considered for backfilling. If a child is unscheduled, and the step is able to schedule it via a call to the planner `do-schedule` (Line 6), then the activity hierarchy's probability profiles are updated (Line 7). Next, the step compares the taskgroup's new eu value to the original (Line 8). If it is higher, the step returns the successfully backfilled child c. If not, it undoes the schedule (Lines 11-12) and continues looping through the tasks and children. If no successful backfills are found once all max-leaf tasks have been considered, the step returns **null** (Line 17).

Algorithm 6.1: Global backfilling step utilizing max-leaf tasks.

```
   Function : doOneGlobalBackfill
1  MLT = maxLeafs()
2  MLT_s = sort({T ∈ MLT | po(T) < 1}, >, eul)
3  eu_TG = eu(TG)
4  foreach T ∈ MLT_s do
5      foreach c ∈ children(T) do
6          if unscheduled(c) and do-schedule(c) then
7              updateAllProfiles()
8              if eu(TG) > eu_TG then
                   //expected utility increased, commit to backfill
9                  return c
10             else
                   //expected utility decreased or stayed the same, undo
                     backfill
11                 undo(do-schedule(c))
12                 updateAllProfiles()
13             end
14         end
15     end
16 end
17 return null
```

This step, as well as the steps for the swapping and pruning tactics (described below), can be modified in several simple ways to suit the needs of different domains. Threshold values for failure probabilities could be utilized, if desired, to limit computation. Also, if a domain has an effective way to sort child methods of a max-leaf task for backfilling, then that could be added as well at Line 5; in our domain, the order of the methods considered for backfilling and swapping was found to be unimportant.

Distributed

The distributed backfilling step has several complications. First, as agents no longer have access to the entire activity network in order to calculate eul values, it must use instead the $\widehat{eul}$ approximation (Section 5.3.4). Second, `updateAllProfiles` can no longer update profiles throughout the entire task tree, and the step must instead rely only on updating local probabilities via the function `updateLocalProfiles`. It follows that the change in taskgroup expected utility that results from a backfill can only be estimated, which can be done using the same approximation technique used to approximate eul and eug. Third, the `maxLeafs` function returns only max-leafs that the agent knows about (whether local or remote), and their children must be checked to see if they are local before they can be considered by an agent. Fourth, agents are strengthening their local schedule asynchronously, and so coordination is required to prevent agents from simultaneously performing duplicate actions needlessly and/or undesirably.

To address the fourth issue, agents communicate a "strengthening status" for each method:

- *no-attempt* : this method is unscheduled and the owning agent has not yet tried to schedule this method as part of a backfill
- *failed-backfill* : this method is unscheduled because the owning agent tried, but failed, to schedule this method as part of a backfill
- *scheduled* : this method is scheduled

The strengthening status is communicated along with probability profiles during execution. Agents use this information to implicitly and partially coordinate schedule strengthening during execution. Namely, given a pre-defined sort order for max-leaf children, agents will not try to backfill a task with their own methods if there is a method earlier in the list of the task's children whose owner has not yet attempted to use it to backfill the task. This helps to prevent the duplicate actions mentioned above, as the first time that a max-leaf task is distributively strengthened all of its children are scheduled (or attempted to be scheduled) in order. It is still possible for duplicate strengthening actions to be performed, such as if two agents simultaneously revisit the max-leaf task on a later

schedule strengthening pass and both succeed in backfilling it; however, in practice we rarely see this occur. We define the function `status` to return a method's strengthening status. The strengthening status is also used to inform the heuristics for some of our strengthening strategies, described in Section 6.3.

Algorithm 6.2 presents the pseudocode for a distributed backfill step. It begins by sorting all max-leaf tasks by their estimated eul values (Line 2). The step next stores the original $eu\,(TG)$ value (Line 3) to use later for comparison. Then, each child of each task is sequentially considered for backfilling. If a child is not local and its owner has not yet attempted to backfill it (*i.e.*, its strengthening status is *no-attempt*), then it skips this max-leaf task and continue to the next. This ensures that methods are (attempted to be) backfilled in order, discouraging concurrent backfilling. If the method is local, unscheduled, and the algorithm is able to schedule it via a call to the planner `do-schedule` (Line 8), then the agent's local activity profiles are updated (Line 9).

Next, the step looks at each method in the agent's local schedule. It estimates the change in expected utility of the taskgroup by summing the estimated change resulting from differences in each method's profiles, if any; this ensures that any effects a backfill has on other methods in the schedule (*e.g.*, pushing them towards a deadline) are accounted for (Line 11). If the expected utility is estimated to be higher, the step returns the successfully backfilled child c. If not, it undoes the schedule (Lines 16-17) and continues looping through the tasks and their children. If no successful backfill is found once all max-leaf tasks have been considered, the step returns **null** (Line 22).

6.2.2 Swapping

Global

A step of the swapping tactic performs one swap, if possible. The step begins by identifying methods that are candidates for swapping. Swapping can serve to strengthen the schedule in two different ways: it can include methods with lower *a priori* failure outcome likelihoods and can increase schedule slack to increase the likelihood that this (or another) method meets a deadline.

Algorithm 6.3 shows pseudocode for a step of the global swapping tactic. The step first finds chains of methods, scheduled back-to-back on the same agent, that have overlapping execution intervals, where at least one method in the chain has a probability of missing a deadline (Line 1). It sorts the methods m within the chains by increasing $eug\,(m)$, so that methods that are expected to contribute the least to the overall utility are swapped first (Line 2).

To this set of chains, the step adds scheduled methods that have an *a priori* likelihood of a failure outcome (Lines 3-5). It identifies the max eul value of the methods in the group, and sorts

Algorithm 6.2: Distributed backfilling step utilizing max-leaf tasks.

```
   Function : doOneDistributedBackfill
 1 MLT = maxLeafs() //all max-leafs the agent knows about, whether
     local or remote
 2 MLT_s = sort({T ∈ MLT | po(T) < 1}, >, êul)
 3 eu_TG = eu(TG)
 4 foreach T ∈ MLT_s do
 5     foreach c ∈ children(T) do
 6         if not(local(c)) and status(c) == no-attempt then
 7             continue //continue to next task
 8         else if local(c) and unscheduled(c) and do-schedule(c) then
 9             updateLocalProfiles()
10             foreach m ∈ localSchedule() do
                   //estimate how m's new po value impacts TG expected
                     utility
11                 Δêu = Δêu + (approximateEU(m, po(m)) − eu_TG)
12             end
13             if Δêu > 0 then
                   //expected utility increased, commit to backfill
14                 return c
15             else
                   //expected utility decreased, undo backfill
16                 undo(do-schedule(c))
17                 updateLocalProfiles()
18             end
19         end
20     end
21 end
22 return null
```

the resulting groups by the decreasing maximum eul, so that groups whose worst expected failure leads to the most utility loss are considered first (Line 9). Next, the step iterates through the groups and methods in the groups to find candidates for swapping (Line 11-12).

A method and its sibling are a candidate for swapping if they are owned by the same agent and the sibling is unscheduled. In addition, the sibling must be considered a good candidate for swapping. If the original method was considered for swapping because it has a failure likelihood, the sibling method must have a lower failure likelihood. If the original method was identified because it was part of a chain, the sibling must meet one of two criteria to be considered a candidate: (1) the sibling has a shorter average duration, so swapping may increase schedule slack; or (2) the sibling has a different set of NLEs, so swapping might increase schedule slack due to the different temporal constraints (Lines 13-18). If the two methods meet this criteria, the planner is called to attempt to swap them. Then, if the swap is successful, the activity hierarchy's probability profiles are updated, and the taskgroup's new eu value is compared to the original (Line 19-20). If it is higher, the step returns the successfully swapped methods (Line 21). If not, it undoes the swap and continues looping through the methods and chains (Line 23-24). If no successful swaps are found once all chains have been considered, the step returns the empty set $\{\}$ (Line 30).

Distributed

The adaptation of this step to make it distributed is analogous to changes made for the distributed version of the backfilling step, where `localSchedule` is called instead of `jointSchedule`, not all profiles can be updated at once, and eul, eug and the change in expected utility of the taskgroup have to be approximated.

6.2.3 Pruning

Global

Algorithm 6.4 shows pseudocode for a step of the global pruning tactic. The step performs one prune, if possible. It begins by finding chains of methods on agents' schedules that have overlapping execution intervals, exactly as in the swapping algorithm (Line 1). Also as with swapping, it sorts the methods m within the chains by increasing eug, and the chains themselves by decreasing eul (Lines 2-3). Next, the step iterates through the chains and each method in the chains to find methods that can be unscheduled (Lines 5-7). When it reaches a method that can be unscheduled, then the activity hierarchy's probability profiles are updated, and the new $eu\,(TG)$ value is compared to the original (Lines 8-9). If it is higher, the step returns the successfully pruned method

Algorithm 6.3: Global swapping tactic for a schedule with failure likelihoods and NLEs.

```
   Function : doOneGlobalSwap
 1 C = findChains(jointSchedule())
 2 foreach c ∈ C do c ← sort(c, <, eug)
 3 foreach m ∈ jointSchedule() do
 4     if fail(m) > 0 then
 5         push({m}, F)
 6     end
 7 end
 8 G = C ∪ F
 9 G = sort(G, >, eul)
10 eu_TG = eu(TG)
11 foreach g ∈ G do
12     foreach m ∈ g do
13         foreach m' ∈ siblings(m) do
14             if owner(m') = owner(m) and unscheduled(m')
15                 and ((g ∈ F and fail(m') < fail(m))
16                     or (g ∈ C and
17                         (averageDur(m') < averageDur(m) or NLEs(m') ≠ NLEs(m))))
18                 and do-swap(m, m') then
19                  updateAllProfiles()
20                  if eu(TG) > eu_TG then
21                      return {m, m'}
22                  else
23                      undo(do-swap(m, m'))
24                      updateAllProfiles()
25                  end
26              end
27          end
28      end
29 end
30 return {}
```

(Line 10). If not, it undoes the unschedule and continues looping through the chains and methods (Lines 12-13). If no successful swaps are found once all chains have been considered, the step returns **null** (Line 18).

Algorithm 6.4: Global pruning step.

```
   Function : doOneGlobalPrune
 1 C = findChains(jointSchedule())
 2 foreach c ∈ C do c ← sort(c, <, eug)
 3 G = sort(C, >, eul)
 4 eu_TG = eu(TG)
 5 foreach g ∈ G do
 6     foreach m ∈ g do
 7         if do-unschedule(m) then
 8             updateAllProfiles()
 9             if eu(TG) > eu_TG then
10                 return m
11             else
12                 undo(do-unschedule(m))
13                 updateAllProfiles()
14             end
15         end
16     end
17 end
18 return null
```

Distributed

The adaptation of this step to make it distributed is analogous to changes made for the distributed version of the backfilling step, where `localSchedule` is called instead of `jointSchedule`, not all profiles can be updated at once, and eul, eug and the change in expected utility of the taskgroup have to be approximated.

6.3 Strategies

Given these three strengthening tactics, the next step is to find a *strategy* that specifies the order in which to apply them. The ordering of tactics can affect the effectiveness of probabilistic schedule

strengthening because the tactics are not independent: performing an action of one tactic, such as unscheduling a method, may make another tactic, such as swapping that method, infeasible. Similarly, performing an action of one tactic, such as swapping sibling methods, may increase schedule slack and make another tactic, such as backfilling a max-leaf task, possible.

In order to develop a good strategy to use for probabilistic schedule strengthening, we first consider a search-based strategy that optimizes the application of the tactics. This baseline, "pseudo-optimal" solution considers each possible order of backfilling, swapping and pruning steps, finds the taskgroup expected utility that results from each ordering, and returns the sequence of steps that results in the highest expected utility. This strategy is not optimal, since the ordering of activities within each tactic is just a heuristic; however, as this strategy is exponential in the number of possible strengthening actions, it still quickly grows to take an unmanageable amount of time. Although this makes its own use infeasible, we use it as a baseline against which to compare more efficient strategies.

Algorithm 6.5 shows the pseudocode for globally finding the best possible ordering of tactics for the baseline strategy. It performs a depth-first search to enumerate all possible orderings, recursing until no more strengthening actions are possible. It returns the highest expected utility for the schedule it found as well as the ordering of tactics that produced that schedule. As we are using this strategy as a baseline, it is not necessary to show how it would be implemented distributively.

We implemented two more efficient strategies to compare against the baseline strategy. One is a variant of a round-robin strategy, shown in Algorithm 6.6. The global version of the strategy performs one backfilling step, one swapping step and one pruning step, repeating until no more steps are possible. Two additional heuristics are incorporated. The first concerns swapping methods to find a lower *a priori* failure likelihood. The heuristic specifies that a swap cannot take place for a method unless the method's parent is *not eligible* for backfilling, where not eligible means that either its po value equals 1 or backfilling has already been attempted on all of the method's siblings (*i.e.*, no siblings have a strengthening status of *no-attempt*). This heuristic prevents swapping methods for lower-utility methods while backfilling a parent max-leaf task (and preserving a higher utility option) is still possible. This is incorporated in as part of the swapping tactic; we do not show the revised algorithm.

The second heuristic is that a prune cannot take place unless either: (1) the candidate for pruning is not the only scheduled child of its parent; or (2) swapping or backfilling any method is not possible at the current point in the algorithm. This heuristic prevents pruning methods, and thus entire branches of the taskgroup, when a backfill or swap is still possible and would better preserve the existing task structure. Again this is added in as part of the pruning tactic; we do not show the revised algorithm.

Algorithm 6.5: Global baseline strengthening strategy utilizing backfills, swaps and prunes.

```
   Function : baselineStrengthening
 1 b = doOneBackfill()
 2 if b then
 3     eu_b, orderings_b = baselineStrengthening()
 4     undo(do-schedule(b))
 5     updateAllProfiles()
 6 end
 7 {s, s'} = doOneSwap()
 8 if {s, s'} then
 9     eu_{s,s'}, orderings_{s,s'} = baselineStrengthening()
10     undo(do-swap(s, s'))
11     updateAllProfiles()
12 end
13 p = doOnePrune()
14 if p then
15     eu_p, orderings_p = baselineStrengthening()
16     undo(do-unschedule(p))
17     updateAllProfiles()
18 end
19 if b or {s, s'} or p then
20     switch max(eu_b, eu_{s,s'}, eu_p) do
21         case eu_b return eu_b, {b} ∪ orderings_b
22         case eu_{s,s'} return eu_{s,s'}, {{s, s'}} ∪ orderings_{s,s'}
23         case eu_p return eu_p, {p} ∪ orderings_p
24     end
25 end
26 return eu(TG), {}
```

Algorithm 6.6: Global round-robin strengthening strategy utilizing backfills, swaps and prunes.

```
   Function : roundRobinStrengthening
1 while true do
2     b = doOneBackfill()
3     {s, s'} = doOneSwap()
4     p = doOnePrune()
5     if not (b or {s, s'} or p) then return
6 end
```

The distributed version of this strategy differs only in how the distributed versions of backfilling, swapping and pruning functions are invoked. The overall structure and heuristics remain the same. As the details are purely implementational, we do not provide them here.

The other strategy is a greedy variant, shown in Algorithm 6.7. This strategy is similar to hill-climbing: it performs one 'trial' step of each tactic, records for each the increase in expected utility that resulted (Lines 2-7), and commits to the step that raises the expected utility the most (Lines 9-20), repeating until no more steps are possible. This greedy strategy also incorporates in the same two heuristics as does the round-robin strategy. In general, it performed the same as the round-robin strategy in our preliminary testing, but at a much higher computational cost; therefore we omit discussion of its distributed counterpart.

Algorithm 6.7: Global greedy strengthening strategy utilizing backfills, swaps and prunes.

```
   Function : greedyStrengthening
 1 while true do
 2     b = doOneBackfill(), eu_b = eu(TG)
 3     if b then undo(do-schedule(b)), updateAllProfiles()
 4     {s, s'} = doOneSwap(), eu_{s,s'} = eu(TG)
 5     if {s, s'} then undo(do-swap(s, s')), updateAllProfiles()
 6     p = doOnePrune(), eu_p = eu(TG)
 7     if p then undo(do-unschedule(p)), updateAllProfiles()
 8     if b or {s, s'} or p then
 9         switch max(eu_b, eu_{s,s'}, eu_p) do
10             case eu_b
11                 do-schedule(b)
12             end
13             case eu_{s,s'}
14                 do-swap(s, s')
15             end
16             case eu_p
17                 do-unschedule(p)
18             end
19         end
20         updateAllProfiles()
21     else
22         return
23     end
24 end
```

6.4 Experimental Results

We used the DARPA test suite of problems described in Section 5.4.2 (on page 98) for the probabilistic schedule strengthening experiments. We ran four experiments: one testing the effectiveness of using PADS to target and prioritize activities for schedule strengthening; one testing the effectiveness of the various schedule strengthening strategies; one testing how schedule utility improves when executing pre-strengthened schedules; and one measuring the effect of probabilistic schedule strengthening when performed in conjunction with replanning during execution.

6.4.1 Effects of Targeting Strengthening Actions

We ran an experiment to test the effectiveness of using PADS to target and prioritize activities when performing probabilistic schedule strengthening [Hiatt et al., 2008]. For simplicity, in this experiment the only tactic considered was backfilling. We compared three different classes of schedules: (1) the original schedule generated by the deterministic, centralized planner; (2) the schedule generated by performing backfilling on the original schedule, using PADS to target and prioritize activities (for clarity we refer to this as *probabilistic backfilling*); and (3) the schedule generated by performing *unconstrained backfilling* on the original schedule. The unconstrained backfilling tactic is deterministic, and loops repeatedly through the set of all scheduled max-leaf tasks and attempts to add one additional method under each task in each pass. It will continue until it can no longer backfill under any of the max-leafs due to lack of space in agents' schedules. This third mode provides the schedule strengthening benefits of backfilling without either the algorithmic overhead or the probabilistic focus of probabilistic backfilling.

We ran the pre-computed schedules for each condition through a multi-agent simulation environment in order to test their relative robustness. All agents and the simulator ran on separate threads on a 2.4 GHz Intel Core 2 Due computer with 2GB RAM. For each problem in our test suite, and each condition, we ran 10 simulations, each with their own stochastically chosen durations and outcomes, but purposefully seeded such that the same set of durations and outcomes were executed by each of the conditions. In order to directly compare the relative robustness of schedules, we executed the schedules as is, with no replanning allowed. The schedule's flexible times representation does, however, allow the time bounds of methods to float in response to execution dynamics, and we did retain basic conflict-resolution functionality that naively repairs inconsistencies in the STN, such as removing methods from a schedule if they miss their `lst`.

The average utility earned by the three conditions for each of the 20 problems is shown in Figure 6.4. The utility is expressed as a function of the scheduled utility of the original initial

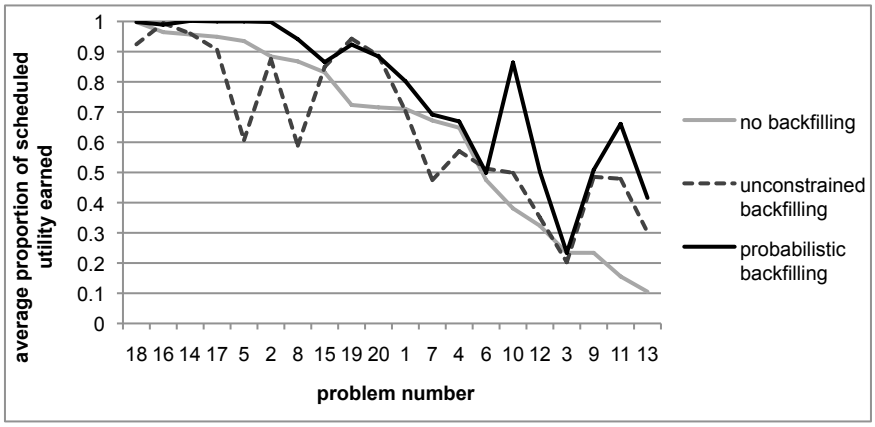


Figure 6.4: Utility earned by different backfilling conditions, expressed as the average proportion of the originally scheduled utility earned.

schedule: it is represented as the average proportion of that scheduled utility earned during execution. Problem numbers correspond to the numbers in Table 5.2, and problems are sorted by how well the condition (1) did; this roughly correlates to sorting the problems by their level of uncertainty. On all but 4 of the problems (16, 19, 20 and 6), probabilistic backfilling earned the highest percentage of the scheduled utility. In the four problems that unconstrained backfilling wins, it does so marginally. On average across all problems, no backfilling earned a proportional utility of 0.64, unconstrained backfilling earned a proportional utility of 0.66, and probabilistic backfilling earned a proportional utility of 0.77. Using a repeated measures ANOVA, probabilistic 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.

Looking more closely at the 4 problems where unconstrained backfilling outperformed probabilistic backfilling, we observe that these results in some cases are due to occasional sub-par planning by the deterministic planner. For example, under some circumstances, the method allocated by the planner under a max-leaf may not be the highest utility child, even though there is enough schedule space for the latter. Unconstrained backfilling may correct this coincidentally due to its undirected nature, while probabilistic backfilling will not even consider this max-leaf if the originally scheduled method is not in danger of earning no utility. This highlights our earlier supposition that the benefit of probabilistic schedule strengthening is affected by the goodness of the original, deterministic schedule.

Figure 6.5 shows the percent increase in the number of methods in the initial schedule after

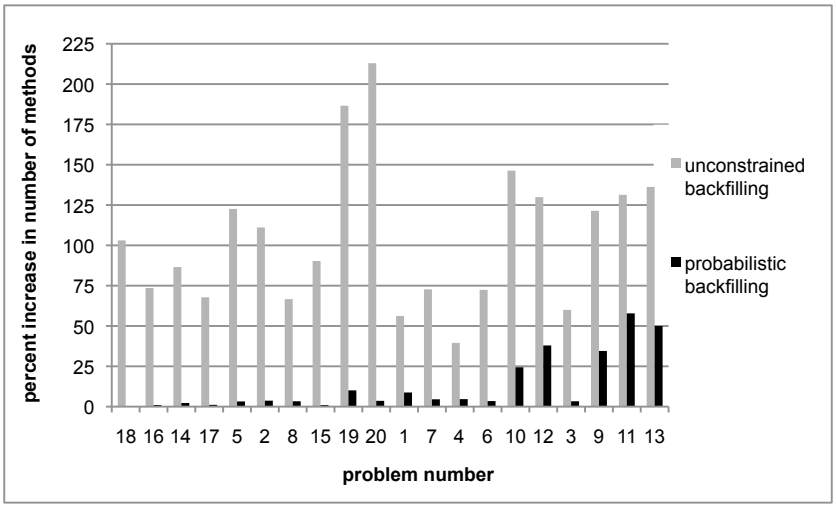


Figure 6.5: The percent increase of the number of methods in the initial schedule after backfilling for each mode.

backfilling for the two modes that performed backfilling. Problem numbers correspond to the numbers in Table 5.2, and problems are sorted to be consistent with Figure 6.4. In all cases but one, probabilistic backfilling added at least 1 method. Despite resulting in less utility earned during execution, unconstrained backfilling far exceeds probabilistic backfilling in the number of backfills.

Comparing Figures 6.4 and 6.5, the effective, probabilistic nature of our schedule strengthening approach is emphasized. In general, for the more certain problems to the left, targeted backfilling does not add many methods because not many are needed. For the more uncertain, difficult problems to the right, it adds methods in order to hedge against that uncertainty. Unconstrained backfilling, in contrast, has no discernible pattern in the graph from left to right. These figures again highlight how schedule strengthening can be limited by the goodness of the original schedule and how it is unable to fix all problems: for problems 6 and 3, for example, probabilistic backfilling does not much affect the utility earned as it is able to make only a small number of effective strengthening actions.

Overall, these results illustrate the advantage to be gained by our probabilistic schedule strengthening approach. There is an inherent trade-off associated with reinforcing schedules in resource-constrained circumstances: there is advantage to be gained by adding redundant methods, as seen by comparing the probabilistic backfilling results to the no backfilling results; however, this advantage can be largely negated if backfilling is done without guidance, as seen by comparing the no backfilling results to the relatively similar results of unconstrained backfilling. Our explicit

use of probabilities to target and prioritize activities provides a means of effectively balancing this trade-off.

6.4.2 Comparison of Strengthening Strategies

We compared the effectiveness of the different strengthening strategies by strengthening initial schedules for the 20 problems in the DARPA test suite in 6 ways: (1) pruning only; (2) swapping only; (3) backfilling only; (4) the round-robin strategy; (5) the greedy strategy; and (6) the baseline strategy [Hiatt et al., 2009]. Strengthening was performed on a 3.6 GHz Intel computer with 2GB RAM. For 8 of the 20 problems the baseline strategy did not finish by 36 hours; for these cases we considered the run time to be 36 hours and the utility earned to be the best found when it was stopped. The resulting average expected utility after the probabilistic schedule strengthening, as well as the average running time for each of the strategies, is shown in Table 6.1. The table also shows, on average, how many strengthening actions each strategy successfully performed; recall, for comparison, that these problems initially had roughly between 50-90 methods initially scheduled.

Table 6.1: A comparison of strengthening strategies.

	avg expected utility (% of baseline)	avg running time (in seconds)	avg num actions
initial	74.5	–	–
prune	77.2	6.7	0.8
swap	85.1	32.9	3.5
backfill	92.8	68.5	10.4
round-robin	99.5	262.0	13.1
greedy	98.5	354.5	13.0
baseline	100	32624.3	13.0

Our supposition of the effectiveness of individual tactics is supported by these results: namely, that backfilling is the strongest individual tactic, followed by swapping and then pruning. Further, despite the limitations of each individual tactic due to each one's associated trade-off, their different properties can also have synergistic effects (*e.g.*, swapping can increase schedule slack, making room for backfilling), and using all three results in stronger schedules than using any one individually.

The round-robin strategy uses, on average, fewer strengthening actions than the sum of the number of actions performed by the pruning, swapping and backfilling strategies independently (13.1 compared to 14.6). In contrast, its running time is much more than the sum of the running times of the three independent strategies (262.0 seconds compared to 108.1 seconds). This is because of the increased search space: when attempting to perform a valid action, instead of only trying to backfill, the round-robin strategy must consider the space of backfills, swaps and prunes. Therefore, in situations where only backfills are possible, for example, the round-robin strategy incurs a higher overhead than the pure backfilling strategy.

Clearly the round-robin strategy outperforms each of the individual strategies it employs at a modest run-time cost. When compared to the greedy strategy, it performs mostly the same. There are, however, a couple problems where round-robin outperforms greedy. An example of such a situation is when the greedy strategy backfilled a max-leaf task twice, but round-robin backfilled it and then swapped two of its children; the round-robin strategy turned out to be better, as it resulted in more schedule slack and left room for other backfills. Although we do not see the difference between the two conditions to be very meaningful, the difference in run time leads us to choose round-robin as the dominant strategy. Additionally, round-robin is almost as effective as the baseline strategy and, with a run-time that is over two orders of magnitude faster, is the preferred choice.

6.4.3 Effects of Global Strengthening

We performed an experiment to compare the utility earned when executing initial deterministic schedules with the utility earned when executing strengthened versions of the schedules. We ran simulations comparing the utility earned by executing three classes of schedules: (1) the schedule generated by the deterministic, centralized planner; (2) the schedule generated by performing backfilling (the best individual tactic) only on the schedule in condition (1); and (3) the schedule generated by performing round-robin schedule strengthening on the schedule in condition (1). Experiments were run on a network of computers with Intel Pentium D CPU 2.80GHz and Intel Pentium 4 CPU 3.60GHz processors; each agent and the simulator ran on its own machine.

For each problem in our DARPA test suite, and each condition, we ran 25 simulations, each with their own stochastically chosen durations and outcomes, but purposefully seeded such that the same set of durations and outcomes were executed by each of the three conditions. As with previous experiments, in order to directly compare the relative robustness of schedules, we executed the schedules as is, with no replanning allowed. Recall that the schedule's flexible times representation does, however, allow the time bounds of methods to float in response to execution dynamics, and

we did retain basic conflict-resolution functionality that naively repairs inconsistencies in the STN, such as removing methods from a schedule if they miss their `lst`.

Figure 6.6 shows the average utility earned by the three sets of schedules for each of the 20 problems. The utility is expressed as the fraction of the originally scheduled utility earned during execution. Problem numbers correspond to the numbers in Table 5.2, and problems are sorted by how well condition (1) did; this roughly correlates to sorting the problems by their level of uncertainty. Note although this graph and Figure 6.4 share two conditions, the data are slightly different; this is due to the previous experiment running only 10 simulations per problem, in contrast to the 25 simulations this experiment performs.

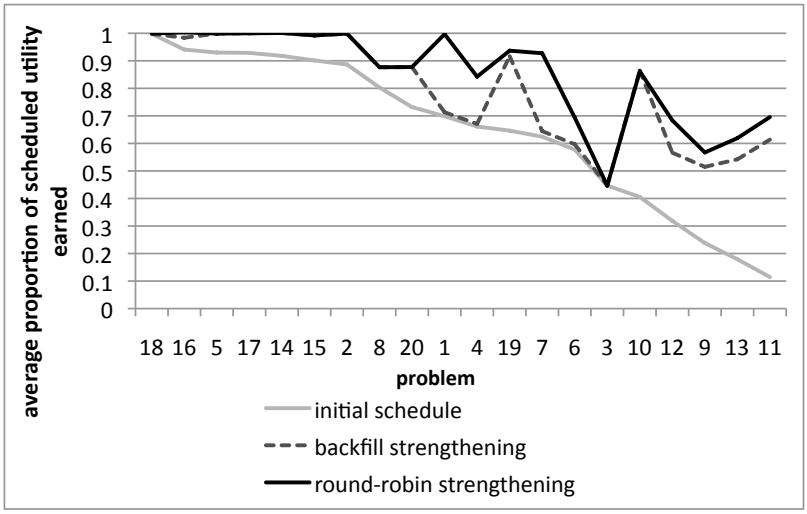


Figure 6.6: Utility for each problem earned by different strengthening strategies, expressed as the average proportion of the originally scheduled utility earned.

For problems 14, 15, 2, 8, 3 and 10, round-robin strengthening identified only backfill steps to be of value, and as a result produced results identical to the backfilling only strategy. Problems 18, 5, 17 and 20, while not identical, are on average very close for these conditions. For individual simulations of these four problems, we see that round-robin strengthening usually performs slightly worse than backfilling alone, but occasionally far outperforms it. This interesting behavior arises because round-robin can hedge against method failure by swapping in a lower utility, shorter duration method for a method with a small chance of earning zero utility. While for many runs this tactic sacrifices a few utility points when the original method succeeds, on average round-robin avoids the big utility loss that occurs when the method fails. For problem 3, strengthening is un-

able to provide much help in making the schedule more robust, and so all three conditions earn roughly the same utility.

As the figure shows, round-robin schedule strengthening outperforms backfilling, which in turn outperforms the initial schedule condition. Round-robin schedule strengthening, overall, outperforms the initial schedule condition by 31.6%. The differences between the conditions are significant with $p < 0.01$, when tested for significance with a repeated measures ANOVA.

6.4.4 Effects of Run-Time Distributed Schedule Strengthening

We next ran an experiment to quantify the effect that schedule strengthening has when performed in conjunction with replanning during execution. We used a 16 problem subset of the 20 problems described above. Twelve of the problems involved 20 agents, 2 involved 10 agents, and 2 involved 25 agents. Three of the problems had fewer than 200 methods, 6 had between 200 and 300 methods, 5 between 300 and 400 methods, and 2 had more than 400 methods. We initially avoided 4 of the problems due to difficulties getting the planner we were using to work well with them; as the results were significant with only 16 problems, we did not spend the time necessary to incorporate them into the test suite for these experiments.

We ran simulations comparing three conditions: (1) agents are initially given their portion of the pre-computed initial joint schedule and during execution replan as appropriate; (2) agents are initially given their portion of a strengthened version of the initial joint schedule and during execution replan as appropriate; (3) agents are initially given their portion of a strengthened version of the initial joint schedule and replan as appropriate during execution; further, agents perform probabilistic schedule strengthening each time a replan is performed. Experiments were run on a network of computers with Intel(R) Pentium(R) D CPU 2.80GHz and Intel(R) Pentium(R) 4 CPU 3.60GHz processors; each agent and the simulator ran on its own machine. For each problem in our test suite we ran 25 simulations, each with their own stochastically chosen durations and outcomes, but purposefully seeded such that the same set of durations and outcomes were executed by each of the three conditions.

Figure 6.7 shows the average utility earned by the three conditions for each of the 16 problems. The utility is expressed as the fraction of the "omniscient" utility earned during execution. We define the omniscient utility for a given simulation to be the utility produced by a deterministic, centralized version of the planner on omniscient versions of the problems, where the actual method durations and outcomes generated by the simulator are given as inputs to the planner. As the planner is heuristic, the fraction of omniscient utility earned can be greater than one. Problem numbers correspond to the numbers in Table 5.2, and problems are sorted by how well the condition (1) did;

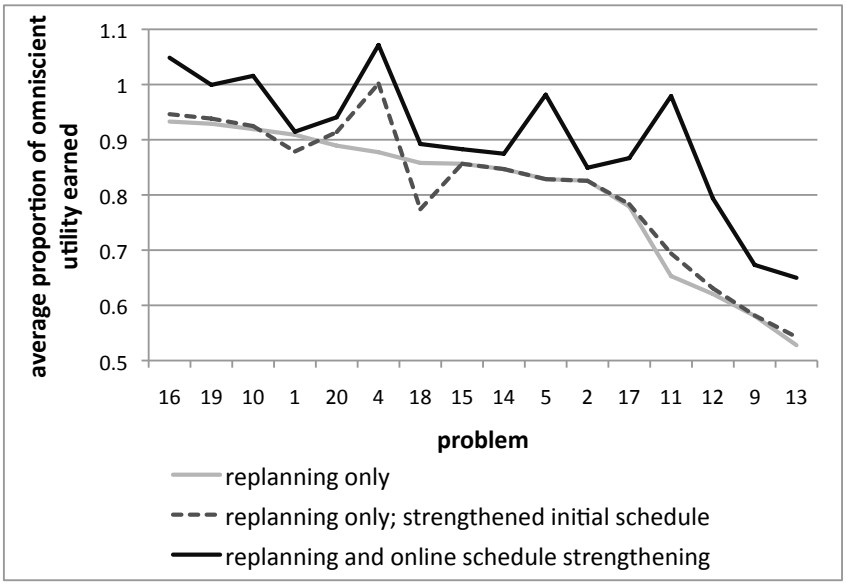


Figure 6.7: Utility for each problem for the different online replanning conditions, expressed as the average proportion of omniscient utility earned.

this roughly correlates to sorting the problems by their level of uncertainty.

Online schedule strengthening results in higher utility earned for all problems. For some the difference is considerable; for others it results in only slightly more utility on average as replanning is able to sufficiently respond to most unexpected outcomes. On average, probabilistic schedule strengthening outperforms condition (1) by 13.6%. Note that this difference is less dramatic than the difference between the probabilistic schedule strengthening and initial schedule conditions of the previous experiment of global schedule strengthening. This difference is because replanning is able to recover from some of the problems that arise during execution. Online probabilistic schedule strengthening does, however, have an additional benefit of protecting schedules against an agent's planner going down during execution, or an agent losing communication so that it can no longer send and receive information to other agents. This is because the last schedule produced will likely be more effective in the face of unexpected outcomes. Note, though, that our results do not reflect this because we assume that communication is perfect and never fails, and so these communication problems are not present in our experiments. Overall, the differences between the online schedule strengthening condition and the other two conditions are significant with $p < 0.01$, when tested for significance with a repeated measures ANOVA.

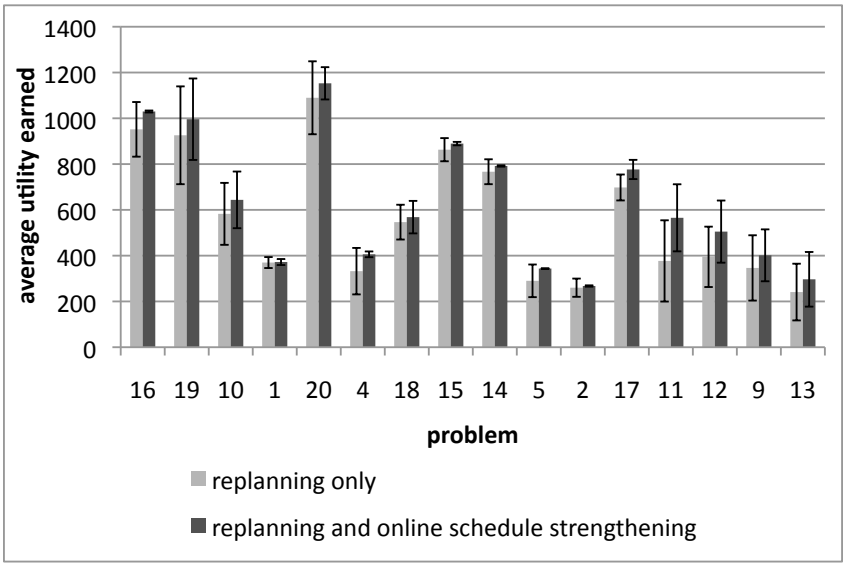


Figure 6.8: Utility earned across various conditions, with standard deviation shown.

Interestingly, conditions (1) and (2), which differ in which joint schedule they begin with, have very similar results for many of the problems. Ultimately this is because when an agent replans for the first time, it effectively "undoes" much of the schedule strengthening: the deterministic planner believes that redundant methods are unnecessary, and that as long as all methods fit in the schedule, the amount of slack is acceptable. This indicates that pre-strengthening schedules is not critical to the success of probabilistic schedule strengthening; still, we include it in future experiments for completeness.

Figure 6.8 shows, for the same problems, the utility earned when replanning only (condition (1)) to the utility earned when both replanning and schedule strengthening (condition (3)). Error bars indicate one standard deviation above and below the mean. A notable feature of this graph is how probabilistic schedule strengthening, in general, reduces the standard deviation of the utility earned across the 25 simulations. These results emphasize how probabilistic schedule strengthening improves the average utility by helping to protect against activity failures, lessening the impact of uncertainty on the outcome of execution.

Timing measurements gathered during these experiments show that each individual call to update probability profiles and to strengthen a schedule was very fast. Individual profile updates took on average 0.003 seconds (with standard deviation $\sigma = 0.01$) and individual schedule strengthenings took on average 0.105 seconds ($\sigma = 0.27$). Overall, each agent spent an average of 1.7

seconds updating probabilities and 37.6 seconds strengthening schedules for each problem. In contrast, each individual replan took an average of 0.309 seconds ($\sigma = 0.39$) with agents spending an average of 47.7 seconds replanning. Even with the fast-paced execution of our simulations, with methods lasting between 15 and 30 seconds, these computations are well within range to keep pace with execution.

Ultimately, the results clearly show that the best approach utilizes both replanning and schedule strengthening to generate and maintain the best schedule possible: replanning provides the means to take advantage of opportunities for earning higher utility when possible, while schedule strengthening works to protect against activity failure and minimize the problems that arise to which the planner cannot sufficiently respond.

Chapter 7

Probabilistic Meta-Level Control

This chapter discusses how to perform effective meta-level control of planning actions by utilizing the information provided by PADS. The first section discusses the control strategy at a high-level, and describes how our overall probabilistic plan management approach comes together. The two following sections provide a more detailed description of probabilistic meta-level control, and can be skipped by the casual reader. Finally, Section 7.4 shows probabilistic meta-level control and PPM experimental results.

7.1 Overview

We developed versions of probabilistic meta-level control for both the single-agent and multi-agent domains. The goal of these control strategies is to intelligently decide, based on a probabilistic analysis, when replanning is likely to be helpful and when it is not. Ideally, agents would replan only if it would be useful either in avoiding failure or in taking advantage of new opportunities. Similarly, they ideally would not replan if it would not be useful, either because: (1) there was no potential failure to avoid; (2) there was a possible failure but replanning could not fix it; (3) there was no new opportunity to leverage; or (4) replanning would not be able to exploit the opportunities that were there. Many of these situations are difficult to explicitly predict, in large part because it is difficult to predict whether a replan is beneficial without actually doing it. As an alternative, probabilistic meta-level control uses the probabilistic analysis to identify times where there is potential for replanning to be useful, either because there is likely a new opportunity to leverage or a possible failure to avoid. The probabilistic meta-level control strategy is the final component of our probabilistic plan management approach.

7.1.1 Single-Agent Domain

The probabilistic meta-level control strategy has two components in the single agent domain. First, there is a probability-based planning horizon, called the *uncertainty horizon*, which limits how much of the schedule is replanned at any given time. A planning horizon is an effective meta-level strategy for this domain due to its relatively flat activity network, its lack of AND nodes, and the presence of activity dependencies only at the method level. The uncertainty horizon is placed where the probability of a replan occurring before execution reaches that point in the schedule rises above a threshold. The threshold can be adjusted via trial and error as necessary to achieve the right balance between expected utility and computation time. The portion of the schedule within the horizon is considered for further PPM; the portion outside is ignored, lowering the computational overhead of managing the plan during execution.

The second meta-level control component determines when replanning is performed during execution. Because this domain has a clear maximal and minimal utility, an effective meta-level control strategy is to replan based on a threshold of expected utility. As execution progresses, as long as the expected utility of the schedule within the uncertainty horizon remains above a threshold, execution continues without replanning, because there is likely enough flexibility to successfully absorb unexpected method durations during execution and avoid failure. If the expected utility ever goes below the threshold, a replan is performed to improve the schedule. The threshold can be adjusted via trial and error as necessary to achieve the right balance between expected utility and computation time. We term this strategy *likely replanning*.

7.1.2 Multi-Agent Domain

The probabilistic meta-level control strategy for our oversubscribed, distributed, multi-agent domain determines when to replan and perform probabilistic schedule strengthening during execution. Because the domain is oversubscribed and the maximal feasible utility is unknown, a likely replanning strategy thresholding on expected utility is not a good fit for this domain. Instead, meta-level control is based on learning when it is effective to replan or strengthen based on the probabilistic state of the schedule: it assumes the initial schedule is as robust as possible and then learns when to replan and strengthen, or only strengthen, based on how execution affects probability profiles at the method level.

There are several reasons for considering only method-level profiles. First, since this is a distributed scenario, looking at the probabilistic state of the taskgroup or other high-level tasks is not always helpful for determining whether locally replanning could be of use. Second, due to the presence of OR nodes in this scenario, the probability profile of the taskgroup or another high-level

task does not always accurately represent the state of the plan due to the different *uafs* at each level of the tree. For example, if a task has two children, one of which will always earn utility and one of which will almost never earn utility, the task itself will have a probability of earning utility of 100%; this probability does not adequately capture the idea that one of its children will probably fail. Finally, we wanted also to learn when probabilistic schedule strengthening would be a good action to take, and lower-level profiles provide a better indication of when this might be useful.

The probabilistic meta-level control strategy for this domain learns when to replan using a classifier; specifically, decision trees with boosting (described in Section 3.4, on page 44). The classifier is learned offline, based on data from previous execution traces. Once the classifier has been learned, it is used to decide what planning action to take at any given time during execution.

7.1.3 Probabilistic Plan Management

With a meta-level control strategy in hand, our overall probabilistic plan management approach can be used to manage plans during execution. For the purposes of this discussion, we assume agent execution is a message-based system, where during execution agents take planning actions in response to update messages, either from other agents providing information such as changes in probability profiles, or from the simulator providing feedback on method execution such as a method's actual start or end time. Such a message is referred to as `updateMessage`. We assume this solely for clarity of presentation.

High-level, general pseudocode for the run-time approach is shown in Algorithm 7.1. During execution, PPM waits for updates to be received, such as about the status of a method the agent is currently executing or from other agents communicating status updates (Line 2). When a message is received, PADS is used to update all (or local for distributed cases) probability profiles (Line 3). Once the profiles have been updated, probabilistic meta-level control is used to decide what planning action to take, if any (Line 4). Once that action has been taken, probability profiles are updated again (Line 7), and any actions the agent needs to take to process the changes (*e.g.*, communicate changes to other agents) are done (Line 9). This process repeats until execution halts.

This algorithm can be instantiated depending on the specifics of probabilistic plan management for a domain. For example, for the single-agent domain, the overall structure of probabilistic plan management for this domain is: during execution, each time a method finishes and the agent gets a message from the simulator about its end time (corresponding to Line 2 of Algorithm 7.1), the agent first updates the uncertainty horizon and then updates all profiles within the horizon to reflect the current state of execution (Line 3). At this point, the probabilistic meta-level control strategy is invoked (Line 4). To decide whether to replan, it considers the current expected utility

Algorithm 7.1: Probabilistic plan management.

```
   Function : ProbabilisticPlanManagement()
 1 while execution continues do
 2     if updateMessage() then
 3         updateAllProfiles()
 4         act = metaLevelControlAction()
 5         if act then
 6             plan(act)
 7             updateAllProfiles()
 8         end
 9         process()
10     end
11 end
```

of the schedule. If it is within the acceptable range, it returns **null** and so the agent does nothing. Otherwise, it returns `replan`, and so the agent replans all activities within the uncertainty horizon, ignoring all others, and updates all profiles within the horizon to reflect the new schedule (Lines 6-7). There is no processing necessary in this domain (Line 9). This process continues until execution has completed.

For the multi-agent domain, any time an agent receives an update message, whether from the simulator specifying method execution information or from another agent communicating information about remote activities (corresponding to Line 2 of Algorithm 7.1), the agent updates its local probabilities (Line 3). It then passes the probabilistic state to the probabilistic meta-level control classifier, which indicates whether the agent should replan and strengthen, only strengthen, or do nothing (Line 4). The agent acts accordingly, and communicates changes to interested agents (Lines 6-9). This process continues until execution has finished.

7.2 Single-Agent Domain

We developed a probabilistic meta-level control strategy for our single-agent domain [Hiatt and Simmons, 2007]. Recall once again that the objective of the domain is that all tasks have exactly one method underneath them execute without violating any temporal constraints, and the only source of uncertainty is in method duration. The schedule is represented as an STN.

The probabilistic meta-level control strategy for this domain has two components: an uncertainty horizon and likely replanning. We begin by discussing a baseline, deterministic plan manage-

ment strategy for this domain, and then describe our more informed, probability-driven strategies.

7.2.1 Baseline Plan Management

In the baseline plan management strategy for this domain, ASPEN is invoked both initially, to generate an initial plan, and then during execution whenever a method finishes at a time that causes an inconsistency in the STN, such as a method missing a deadline or running so early or so late that a future method is unable to start in its specified time window.

7.2.2 Uncertainty Horizon

In general, we would like to spend time managing only methods that are near enough to the present to warrant the effort, and not waste time replanning methods far in the future when they will only have to be replanned again later. The problem, then, is how to determine which methods are "near enough." We pose this as how to find the first method at which the probability of a replan occurring before that method is executed rises above a threshold, which we call the *horizon threshold*. We calculate the probability of a replan occurring before a method starts executing based on the baseline, deterministic plan management system.

The high-level equation for finding the probability that the baseline system will need to replan before the execution of some method m_i in the schedule is:

$$(1 - po_b(m_1, ..., m_i))$$

where $po_b(m_j)$ is defined as the probability that method m_j meets its plan objective by observing the constraints in the deterministic STN. We simplify the calculation by assuming this means it ends within its range of valid end times, $[\texttt{eft}(m_j), \texttt{lft}(m_j)]$, given that it starts at $\texttt{est}(m_j)$.

Because $po_b(m_i)$ is independent of $po_b(m_1, ..., m_{i-1})$, $po_b(m_1, ..., m_i) = \prod_{j=1}^{i} po_b(m_j)$. To begin calculating, we find the probability that each method m ends in the range $[\texttt{eft}(m), \texttt{lft}(m)]$ given that it starts at its $\texttt{est}$:

$$\begin{aligned} po_b(m) =& \texttt{normcdf}(\texttt{lft}(m), \mu(\texttt{DUR}(m)) + \texttt{est}(m), \sigma(\texttt{DUR}(m))) \\ & - \texttt{normcdf}(\texttt{eft}(m), \mu(\texttt{DUR}(m)) + \texttt{est}(m), \sigma(\texttt{DUR}(m))) \end{aligned}$$

Multiplying these together, we can find the probability of a replan occurring before execution reaches method m_i. The horizon is set at the method where this probability rises above the user-defined horizon threshold. The algorithm pseudocode is shown in Algorithm 7.2, where the variable max_horizon indicates that the horizon includes all scheduled methods. Recall that the function `schedule` returns the agent's current schedule, with methods sorted by their scheduled start time.

Algorithm 7.2: Uncertainty horizon calculation for a schedule with normal duration distributions.

Function : `calcUncertaintyHorizon`

$po_b = 1$

foreach method $m \in$ `schedule` **do**

  $po_b = po_b \cdot (\texttt{normcdf}(\texttt{lft}(m), \mu(\texttt{DUR}(m)) + \texttt{est}(m), \sigma(\texttt{DUR}(m)))$

   $- \texttt{normcdf}(\texttt{eft}(m), \mu(\texttt{DUR}(m)) + \texttt{est}(m), \sigma(\texttt{DUR}(m))))$

  if $(1 - po_b) >$ horizon_threshold **then return** m

end

return max_horizon

We find the uncertainty horizon boundary and use it to limit the scope of probabilistic plan management, ignoring methods, tasks and constraints outside of the horizon. A higher horizon threshold is a more conservative approach and results in a longer horizon, where more methods and tasks are considered during each replan. A lower threshold corresponds to a shorter horizon, and results in fewer methods being considered by PPM. The threshold can be adjusted via trial and error as necessary to achieve the right balance between expected utility and computation time. In our experimental results (Section 7.4.1), we show experiments run with various thresholds and demonstrate the impact they have on plan management.

When a replan is performed, the planner replans all methods and tasks within the uncertainty horizon. This may cause inconsistencies between portions of the schedule inside and outside of the horizon as constraints between them are not observed while replanning. As long as at least one activity involved in the constraint remains outside of the horizon, such constraint violations are ignored. As execution progresses, however, and the horizon correspondingly advances, these conflicts shift to be inside the uncertainty horizon. They are handled at that time as necessary by replanning.

7.2.3 Likely Replanning

The decision of when to replan is based on the taskgroup's expected utility within the uncertainty horizon. The algorithm to calculate this value is very similar to Algorithm 5.3 (on page 68), which calculates the expected utility of the entire schedule. The differences are that instead of iterating through all methods and tasks to update activity probability profiles, only the profiles of methods and tasks within the uncertainty horizon are updated. Similarly, when calculating $po(TG)$, only those tasks within the uncertainty horizon are considered. The expected utility calculation within the uncertainty horizon is shown in Algorithm 7.3.

Algorithm 7.3: Taskgroup expected utility calculation, within an uncertainty horizon, for the single-agent domain with normal duration distributions and temporal constraints.

Function : $\texttt{calcExpectedUtility}\,(uh)$

foreach method $m_i \in \texttt{schedule}$ **until** uh **do**

`//calculate` m_i`'s start time distribution`

$ST\,(m_i) = \texttt{truncate}\,(\texttt{est}'\,(m_i)\,, \infty, \mu\,(FT_s\,(m_{i-1}))\,, \sigma\,(FT_s\,(m_{i-1})))$

`//calculate` m_i`'s finish time distribution`

$FT\,(m_i) = ST\,(m_i) + \texttt{DUR}\,(m_i))$

`//calculate` m_i`'s probability of meeting its objective`

$po\,(m_i) = \texttt{normcdf}\,(\texttt{lft}'\,(m_i)\,, \mu\,(FT\,(m_i))\,, \sigma\,(FT\,(m_i))) -$
$\texttt{normcdf}\,(\texttt{eft}'\,(m_i)\,, \mu\,(FT\,(m_i))\,, \sigma\,(FT\,(m_i)))$

`//calculate` m_i`'s expected utility`

$eu\,(m_i) = po\,(m_i)$

`//adjust the FT distribution to be` m_i`'s FT distribution given it succeeds`

$FT_s\,(m_i) = \texttt{truncate}\,(\texttt{eft}'\,(m_i)\,, \texttt{lft}'\,(m_i)\,, \mu\,(FT\,(m_i))\,, \sigma\,(FT\,(m_i)))$

end

foreach $T \in \texttt{children}\,(TG)$ **until** uh **do**

if $|\texttt{scheduledChildren}\,(T)| == 1$ **then**

$po\,(T) = po\,(\texttt{sole-element}\,(T))$

$eu\,(T) = eu\,(\texttt{sole-element}\,(T))$

else

$po\,(T) = 0$

$eu\,(T) = 0$

end

end

$po\,(TG) = \prod_{T \in \texttt{children}(TG)\ \textbf{until}\ uh} po\,(T)$

$eu\,(TG) = po\,(TG)$

Note that, as before, we set $eu\,(TG)$ equal to $po\,(TG)$, where $po\,(TG)$ is the probability that the schedule within the horizon executes successfully. This means the range of values for $eu\,(TG)$ within the horizon is $[0, 1]$; this is ideal, as we are using the value to compare to a fixed threshold, called the *replanning threshold*. As long as the expected utility of the taskgroup remains above the replanning threshold, no replanning is performed; if the expected utility ever goes below the threshold, however, a replan is done to improve schedule utility. A high replanning threshold, close to 1, corresponds to a more conservative approach, rejecting schedules with high expected utilities and replanning more often. At very high thresholds, for example, even a conflict-free schedule just generated by the planner may not satisfy the criterion if its probability of successful execution, and thus its expected utility, is less than 1. Lower thresholds equate to more permissive approaches, allowing schedules with less certain outcomes and lower expected utility to remain, and so performing fewer replans. The threshold can be adjusted via trial and error as necessary to achieve the right balance between utility and replanning time. In our experimental results (Section 7.4.1), we show experiments run with various thresholds and demonstrate the impact they have on plan management.

If a replan is not performed, there may be a conflict in the STN due to a method running early or late. If that is the case, we adjust methods' scheduled durations to make the STN consistent. To do so, we first use the STN to find all valid integer durations for unexecuted methods. Then we iteratively choose for each method the valid duration closest to its original one as the method's new scheduled, deterministic duration. The durations are reset to their domain-defined values when the next replan is performed. The ability to perform this duration adjustment is why we are able to perform the extra step of extending the STN to STN′ for our analysis (see Section 5.2, on page 63), and is what allows us to assume that methods can violate their `lst` as long as they do not violate their `lst′`.

7.2.4 Probabilistic Plan Management

The overall PPM algorithm is derived from Algorithm 7.1. As before, the agent waits for update messages to be received. When one is, the agent updates all probability profiles. The function `updateAllProfiles` (Line 3 in Algorithm 7.1) for this domain is defined as

`calcExpectedUtility(calcUncertaintyHorizon())`

This equation finds the uncertainty horizon and updates all probability profiles within it. The agent next determines whether to replan. The meta-level control function (Line 4) is defined as:

if $eu\,(TG) <$ planning_threshold **then return** `replan` **else return null**

Finally, after agents have replanned (or not), they have nothing to process at this point, so `process` does not do anything.

7.3 Multi-Agent Domain

We implemented a probabilistic meta-level control strategy for the multi-agent domain, where the plan objective is to maximize utility earned. We begin by discussing baseline plan management for this domain, then describe our probabilistic meta-level control strategy and overall probabilistic plan management approach.

7.3.1 Baseline Plan Management

We use the planning system described in [Smith et al., 2007] as our baseline plan management system for this domain. During execution, agents replan locally in order to respond to updates from the simulator specifying the current time and the results of execution for their own methods, as well as updates passed from other agents communicating the results of their execution, changes due to replanning, or changes to their schedules that occur because of messages from other agents. Specifically, replans occur when:

- A method completes earlier than its expected finish time and the next method cannot start immediately (a replan here exploits potential schedule slack)
- A method finishes later than the `lst` of its successor or misses a deadline
- A method starts later than requested and causes a constraint violation
- A method ends in a failure outcome
- Current time advances and causes a constraint violation

No updates are processed while an agent is replanning or schedule strengthening, and so important updates must wait until the current replan or strengthening pass, if one is occurring, has finished before they can be incorporated into the agent's schedule and responded to. More detail on this planning system can be found in [Smith et al., 2007].

7.3.2 Learning When to Replan

Our probabilistic meta-level control strategy is based on a classifier which classifies the current state of execution and indicates what planning action, if any, the agent should take. We use decision trees with boosting (see Section 3.4) to learn the classifier. The learning algorithm takes training

examples that pair states (in our case, probabilistic information about the current schedule) and classifications (*e.g.*, "replan and strengthen," "do nothing," etc.), and learns a function that maps from states to classifications. States are based on changes in probability profiles at the method and max-leaf task level that result from update messages. Considering profiles at this level is appropriate for distributed applications like this one: higher-level profiles can be poor basis for a control algorithm because the state of the joint plan does not necessarily directly correspond to the state of an agents' local schedule and activities.

For each local method, we look at how its probability of meeting its objective (po), finish time distribution when it earns utility (FT_u) and finish time distribution when it does not earn utility (FT_{nu}) change. For each of these values, we consider the maximum increase and decrease in their value or, for distributions, their average and standard deviation. This allows the meta-level control strategy to respond to, for example, methods' projected finish times becoming earlier or later, or a large change in a method's probability of earning utility. We also look at the po value of max-leaf tasks and their remote children, as well as whether any of their remote children have been scheduled or unscheduled, since changes in these values are indicative of whether probabilistic schedule strengthening may be useful. The overall state consists of:

1. the largest increase in local method po
2. the largest decrease in local method po
3. the largest increase in local method $\mu\,(FT_u)$
4. the largest decrease in local method $\mu\,(FT_u)$
5. the largest increase in local method $\sigma\,(FT_u)$
6. the largest decrease in local method $\sigma\,(FT_u)$
7. the largest increase in local method $\mu\,(FT_{nu})$
8. the largest decrease in local method $\mu\,(FT_{nu})$
9. the largest increase in local method $\sigma\,(FT_{nu})$
10. the largest decrease in local method $\sigma\,(FT_{nu})$
11. the largest increase in max-leaf task po
12. the largest decrease in max-leaf task po
13. the largest increase in any max-leaf tasks' remote child po
14. the largest decrease in any max-leaf tasks' remote child po
15. whether any max-leaf task's remote children have been scheduled or unscheduled

To distinguish this state from the probabilistic state of the schedule as a whole, we call this state the "learning state."

We trained two different classifiers: one to decide whether to replan, and one to decide whether to replan and strengthen, only strengthen, or do nothing. We use the former in our experiments to test the effectiveness of the probabilistic meta-level control strategy when used alone. It learned when to replan based on the baseline system with no strengthening; its possible classifications were "replan only" or "do nothing." We use the latter to test the overall plan management approach, which also includes probabilistic schedule strengthening. Its possible classifications were "replan and strengthen," "strengthen only," and "do nothing." We do not consider "replan only" a condition here because, as previously argued (Section 6.1), replanning should always be used in conjunction with strengthening, if possible.

To generate our training data, we used 4 training problems that closely mirrored in structure those used in our experiments, described in Section 7.4.2. We executed the problems 5 times each. During execution, agents replanned in response to every update received (for the first classifier), or replanned and strengthened in response to every update received (for the second classifier). For the rest of this section, we describe the learning approach in terms of the more general case, which includes strengthening.

We analyzed the execution traces of these problems to generate training examples. Each time a replan and strengthening pass was performed by an agent in response to an update, we recorded the learning state that resulted from the changes in probabilities caused by that update. We next looked to see if either replanning or strengthening resulted in "important" changes to the schedule. Important changes were defined as changes that either: (1) removed methods originally in the schedule but not ultimately executed; or (2) added methods not originally in the schedule but ultimately executed. Note that a method could be removed from the schedule but still executed if, for example, it was added back onto the schedule at a later time; this would be considered a non-important change. If an important change was made, the learning state was considered a "replan and strengthen" or "strengthen only" example, depending on which action resulted in the change. If no important change was made, the learning state was considered a "do nothing" example.

Using this data, we learned a classifier of 100 decision trees, each of height 2. When given a state as an input, the classifier outputs a sum of the weighted votes for each of the three classes. Each sum represents the strength of the belief that the example is in that class according to the classifier. In classic machine learning, the classification with the maximal sum is used as the state's classification.

In our domain, however, the data are very noisy and, despite using decision trees with boosting (which are typically effective with noisy data), we found that the noisy data were still adversely affecting classification and almost all learning states were being classified as "do nothing." The source of the noise is the lack of direct correlation between a schedule's probabilistic state and

whether a replan will improve it, since it is hard to predict whether a replan will be useful. For example, if something is drastically wrong with the schedule, a replan might not be able to fix it; similarly, if nothing is wrong with the schedule, a replan still might improve scheduled utility. In terms of our classifier, this means that in situations, for example, where the need to replan may seem obvious according to the learning state (*e.g.*, many probability values have changed), replanning is not always useful and so in our training data such learning states were often classified as "do nothing."

We address this by changing the interpretation of the weighted votes for the three classes output by the classifier. Instead of treating them as absolute values, we treat them as relative values and compare them to a baseline classification to see whether the weight of the classifications for "replan and strengthen" or "strengthen only" increase, decrease, or remain the same. If the weight for one of the classifications increases, such as "replan and strengthen," then we determine that replanning and strengthening may be useful in that case. More technically, to get a baseline classification, we classify the baseline learning state where no probabilities have changed (*i.e.*, a learning state of all zeroes). Then, for each class, we calculate the proportion that the current learning state's weighted votes have as compared to the baseline example, and use that as the class's new weighted vote. As a second step to address the noisy data, we randomly choose a class, with each class weighted by their adjusted votes. These steps are shown below.

7.3.3 Probabilistic Plan Management

The overall PPM algorithm is derived from Algorithm 7.1. As before, the agent waits for update messages to be received. When one is, the agent updates all of its local probability profiles, via a call to `updateLocalProfiles`, and generates the learning state based on the new profile probabilities. Then, the meta-level control algorithm is invoked to classify the learning state. The pseudocode for the probabilistic meta-level control algorithm for this domain, which implements

`metaLevelControlAction` from Algorithm 7.1, is:

$rs, so, dn =$ `classify` $(learning\_state)$
$rs_b, so_b, dn_b =$ `classify` $(0,\ 0,\ 0,\ 0,\ 0...)$
$rs' = rs/rs_b, so' = so/so_b, dn' = dn/dn_b$
$rand =$ `random` $\left(0, rs' + so' + dn'\right)$
switch $rand$
  case $rand < rs'$: **return** `replan` $()$, `updateLocalProfiles` $()$, `roundRobinStrengthening` $()$
  case $rs' \leq rand < rs' + so'$: **return** `roundRobinStrengthening` $()$
  case $rs' + so' \leq rand < rs' + so' + dn'$: **return null**
end

The algorithm classifies both the learning state and the baseline learning state of all zeros, and scales the learning state's classification weights as described above. Then, it randomly samples a class from the scaled values. Based on this, it returns the appropriate planning action(s), and the agent performs that action as per Algorithm 7.1. If a replan is performed, probabilities must be updated before probabilistic schedule strengthening can be performed; it is not necessary to update probability profiles after a strengthen, however, as probabilistic schedule strengthening takes care of that. If no replan or strengthening pass is performed, execution continues with the schedule as-is. If there is an inconsistency in the STN due to a method running late, however, the next method is removed from the schedule to make the STN consistent again. At this point, all that is left to do is for the agent to process the changes (`process` in Algorithm 7.1) by sending updates to other agents notifying them of appropriate changes.

7.4 Experiments and Results

We performed experiments to characterize the benefits of probabilistic meta-level control for the single-agent domain, which uses a thresholding meta-level control strategy, as well as for the multi-agent domain, which uses a learning-based meta-level control strategy. Finally, we ran additional experiments characterizing the overall benefit of probabilistic plan management for each domain.

7.4.1 Thresholding Meta-Level Control

We performed several experiments to characterize the effectiveness of the different aspects of meta-level control, as well as PPM overall, for the single-agent domain [Hiatt and Simmons, 2007]. We used as our test problems instances PSP94, PSP100 and PSP107 of the mm-j20 benchmark suite of the resource-relaxed Multi-Mode Resource Constrained Project Scheduling Problem with Minimal and Maximal Time Lags (MRCPSP/max) [Schwindt, 1998]. The problems were augmented as described in Section 3.3.1.

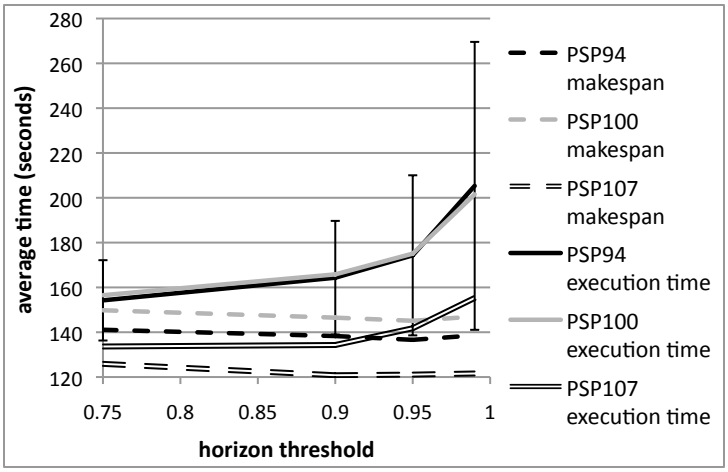
We developed a simple simulator to execute schedules for this domain. Execution starts at time zero, and methods are started at their `est`. To simulate method execution the simulator picks a random duration for the method based on its duration distribution, advances time by that duration, and updates the STN as appropriate. In this simple simulator, time does not advance while replanning is performed; instead, time effectively is paused and resumed when the replan is complete.

Uncertainty Horizon Analysis

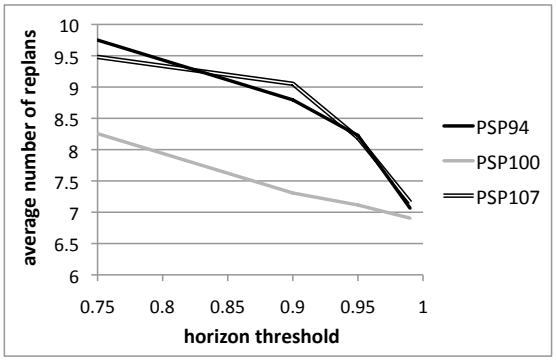
The first experiment tested how the choice of a horizon threshold affects the performance of execution. In this experiment, the agent replanned whenever a method ended at a duration other than its scheduled one, but replanned activities only within the uncertainty horizon. We ran experiments under 5 conditions: the baseline deterministic system, and four conditions testing horizon threshold values of 0.75, 0.9. 0.95 and 0.99. A threshold of 0.75 corresponds to replanning approximately 2 methods in the future, and a threshold of 0.99 replans approximately all 20 methods; in between these two extremes, the number of methods scales roughly exponentially as the threshold linearly increases.

The experiment was run on a 3.4 GHz Pentium D computer with 2 GB RAM. We performed 250 simulation runs for each condition and problem instance combination. Method duration means and standard deviations were randomly re-picked for each simulated run.

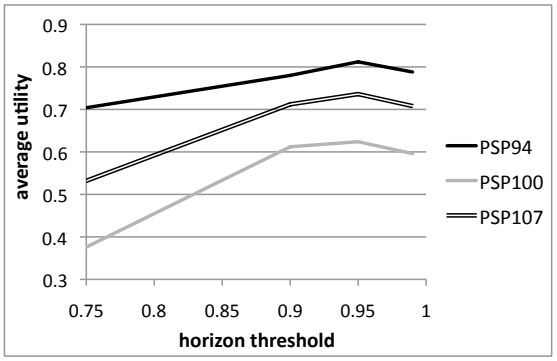
The results are shown in Figure 7.1. Figure 7.1(a) shows both the average makespan of the executed schedule alone and the average makespan plus average time spent managing and replanning, which we refer to as *execution time*. It also has error bars for PSP94 showing the standard deviation of the execution time; for clarity of the graph, we omit error bars for the other two problems. We believe the high variances in execution time, shown by the error bars, are due to the randomness of execution and the randomness of ASPEN, the planner. Figure 7.1(b) shows the average number of replans per simulation run.



(a) Average makespan and execution time using an uncertainty horizon at various thresholds.



(b) Average number of replans using an uncertainty horizon at various thresholds.



(c) Average utility using an uncertainty horizon at various thresholds.

Figure 7.1: Uncertainty horizon results.

Figure 7.1(c) shows average expected utility. Simulations that did not earn utility failed due to the planner timing out during a replan, either because a solution did not exist (*e.g.*, a constraint violation could not be repaired) or a solution was not found in time. We began by allowing ASPEN to take 100,000 repair iterations to replan (Section 3.3.1); this resulted, however, in the planner timing out unintuitively often. Believing this to be due in part to ASPEN's randomness, we therefore switched to a modified random-restart approach [Schoning, 1999], invoking it 5 separate times with an increasing limit of repair iterations - 5000, 5000, 10000, 25000, and finally 50000. This approach significantly decreased the number of time outs. All other statistics reported are compiled from only those runs that completed successfully. Note that the notion of time-outs would be less important when using more methodical, and less randomized, planners.

Figure 7.1(a) shows that, as the horizon threshold increases, the average schedule makespan decreases - an unanticipated benefit. It occurs because with longer horizons, the planner is able to generate better overall plans, since it plans with more complete information. As the horizon threshold increases, however, the average execution time also increases. This is due to replanning more methods and tasks each time a replan is performed, increasing the amount of time each replan takes. This point is highlighted when one considers Figure 7.1(b), which shows that despite the execution time and replanning time increasing as the horizon threshold increases, the number of replans performed *decreases*; therefore, clearly, each replan must take longer as the threshold approaches 1.

The trend in the number of replans in Figure 7.1(b) occurs because, as the threshold decreases, the number of replans increases due to needing to replan extra conflicts that arise by extending the uncertainty horizon. This trend in turn explains Figure 7.1(c), where we believe higher thresholds earn on average higher utility both because they have fewer conflicts between activities inside and outside of the horizon, and because there are fewer replans performed, lessening the number of replans that result in a time-out.

Although as the threshold in Figure 7.1(a) decreases the execution time plateaus, the balance of management time and makespan continues to change, with the schedule span increasing as the time spent managing continues to decrease. This suggests the use of a threshold where the overall execution time is near the lower bound, and the schedule span is as low as possible, such as at the thresholds of 0.75 or 0.9.

A repeated measures ANOVA test showed that the choice of horizon threshold significantly affects schedule makespan, execution time, number of replans, and utility earned, all with a confidence of 5% or better.

Likely Replanning Analysis

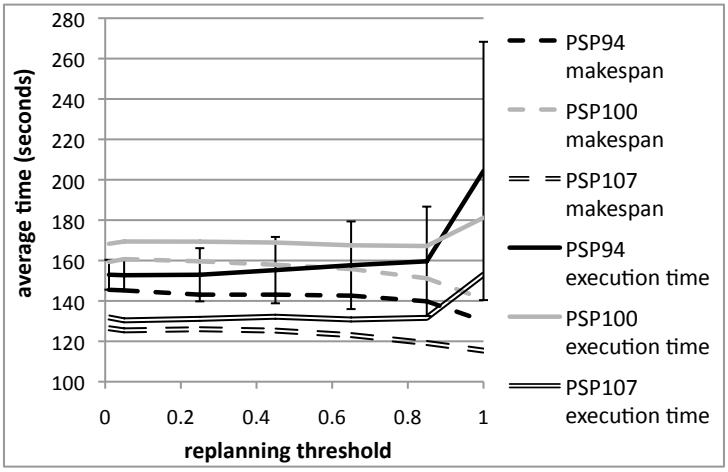
We also tested how the choice of a replanning threshold affects the performance of execution. In this experiment, the agent replanned all activities (*i.e.*, there was no uncertainty horizon), but replanned only when the schedule's expected utility fell below the replanning threshold. We ran experiments under 7 conditions testing replanning threshold values of 0.01, 0.05, 0.25, 0.45, 0.65, 0.85 and 1. Note that a replanning threshold of 1 has the potential to replan even if there are no actual conflicts in the plan, since there may still be a small possibility of a constraint violation.

The experiment was run on a 3.4 GHz Pentium D computer with 2 GB RAM. We performed 250 simulation runs for each condition and problem instance combination. Method duration means and standard deviations were randomly re-picked for each simulated run.

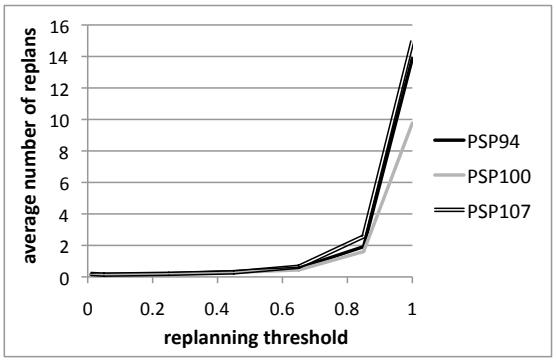
The results are shown in Figure 7.2. Figure 7.2(a) shows both the average makespan of the executed schedule and the execution time, with error bars for PSP94 showing the standard deviation of execution time; we do not show error bars for the other two problems for clarity of the graph. As with the uncertainty horizon results, we believe that the high variances in execution time are due to the randomness of execution and the planner. Figure 7.2(b) shows the average number of replan episodes per simulation run, and Figure 7.2(c) shows the average utility earned. As with the uncertainty horizon results, simulations that did not earn utility failed due to the planner timing out during a replan, either because a solution did not exist (*e.g.*, a constraint violation could not be repaired) or the solution was not found in time. The same modified random-restart approach was utilized, and all other statistics reported are compiled from only those runs that completed successfully.

As Figure 7.2(a) shows, schedule makespan tends to decrease as the replanning threshold increases due to its requirement that schedules have a certain amount of expected utility, which loosely correlates with schedule temporal flexibility. The overall execution time, however, increases, as more replans are performed in response to more schedules' expected utility not meeting the replanning threshold. Figure 7.2(b) explicitly shows this trend of the average number of replans increasing as the replanning threshold increases. The threshold of 1 clearly takes this to an extreme. This threshold, in fact, becomes problematic when one considers Figure 7.2(c). The high expectations incurred by a replanning threshold of 1 result in a higher number of failures, due to more frequent replans which have a chance of not finding a solution in the allotted amount of time. We expect that the average utility would increase if we increased the number of iterations allowed to the planner.

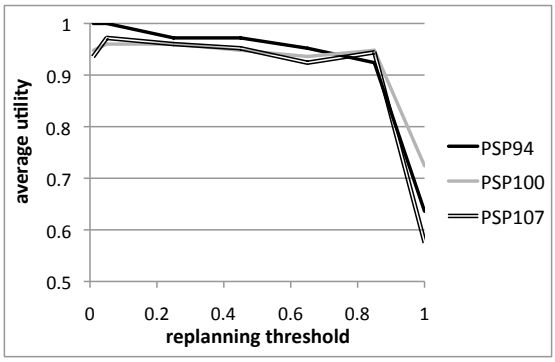
As with the uncertainty horizon, Figure 7.2(a) shows that as the replanning threshold decreases, the execution time plateaus, but the balance of management time and makespan continues to



(a) Average makespan and execution time using likely replanning at various thresholds.



(b) Average number of replan episodes using likely replanning at various thresholds.



(c) Average utility using likely replanning at various thresholds.

Figure 7.2: Likely replanning results.

change, with the schedule span increasing as the time spent continues to decrease. This again suggests the use of a threshold where the overall execution time is near the lower bound, and the schedule span is as low as possible, such as at a threshold of 0.65 or 0.85.

A repeated measures ANOVA showed that the choice of replanning threshold significantly affects schedule makespan, execution time, number of replans and utility, all with a confidence of 5% or better.

Analysis of Probabilistic Plan Management

We ran a third experiment to test overall PPM for this domain, which integrates the probabilistic meta-level control strategies of uncertainty horizon and likely replanning. The experiment compared the probabilistic meta-level control strategy with the baseline management strategy for this domain (Section 7.2.1). In the experiment we also varied the thresholds for the uncertainty horizon and probabilistic flexibility.

As with the two previous experiments, this experiment was run on a 3.4 GHz Pentium D computer with 2 GB RAM. We performed 250 simulation runs for each condition and problem instance combination. Method duration means and standard deviations were randomly re-picked for each simulated run.

To summarize the results, Figure 7.3 shows the average execution time across all three problems using various combinations of thresholds of the uncertainty horizon and likely replanning. We also show the average utility in Figure 7.4; note the axes are reversed in the diagram for legibility. The label no-uh refers no uncertainty horizon being used (and so only using likely replanning), and the label no-lr refers to not utilizing likely replanning (and so only using the uncertainty horizon). The bar with both no-uh and no-lr, then, is baseline deterministic plan management. The bars are also labeled with their numerical values.

The analyses of the two individual components hold when we consider the simulations that incorporate both likely replanning and the uncertainty horizon. Specifically, the average utility is highest at higher horizon thresholds and lower replanning thresholds, as Figure 7.4 shows. Figure 7.3 also shows that as the thresholds decrease, the average execution time in general decreases both because it replans fewer tasks during each replan episode and because fewer replans are made. These figures also show that using the uncertainty horizon and likely replanning together results in a lower execution time than each technique results in separately, while retaining the high average utility earned when using likely replanning alone. The graphs indicate also that after a certain point the analysis is not extremely parameter sensitive, and the execution time plateaus at a global minimum.

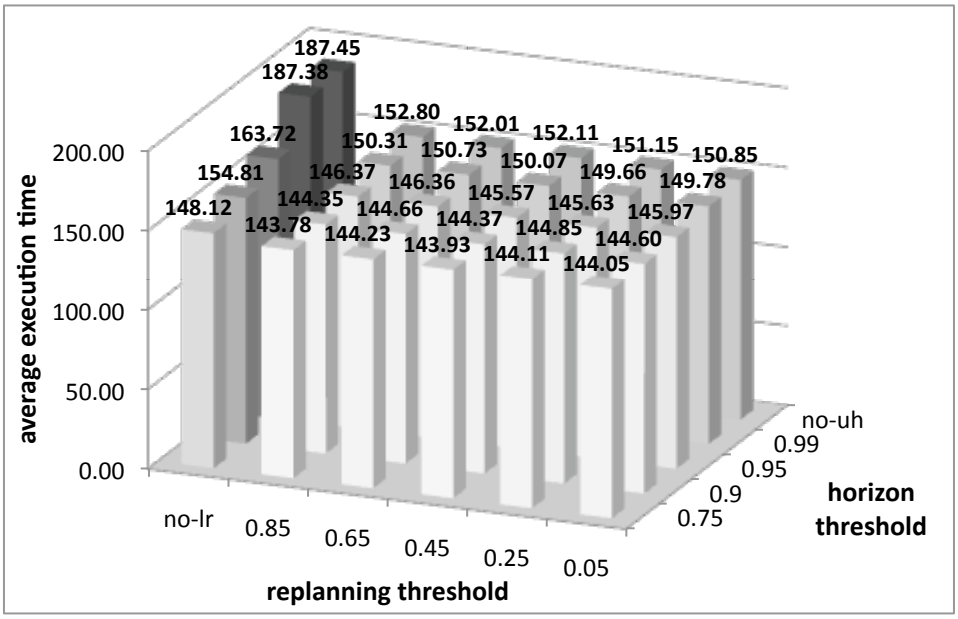


Figure 7.3: Average execution time across all three problems using PPM with both the uncertainty horizon and likely replanning at various thresholds.

Overall, these results show that the benefit of PPM is clear, and achieves a significantly better balance of time spent replanning during execution versus schedule makespan. Using a horizon threshold of 0.9 and a replanning threshold of 0.85, PPM decreases the schedule execution time from an average of 187s to an average of 144s - a 23% reduction, while increasing average utility by 30%.

7.4.2 Learning Probabilistic Meta-Level Control

We performed several experiments to characterize the effectiveness of the different aspects of meta-level control, as well as PPM overall, for the multi-agent domain. To perform the experiments, we use a multi-agent, distributed execution environment. Agents' planners and executors run on separate threads, which allows deliberation to be separated from execution. The executor executes the current schedule by activating the next pending activity to execute as soon as all of its causal and temporal constraints are satisfied. A simulator also runs separately and receives method start commands from the agent executor and provides back execution results as they become known. All agents, and the simulator, run and communicate asynchronously but are grounded by a current

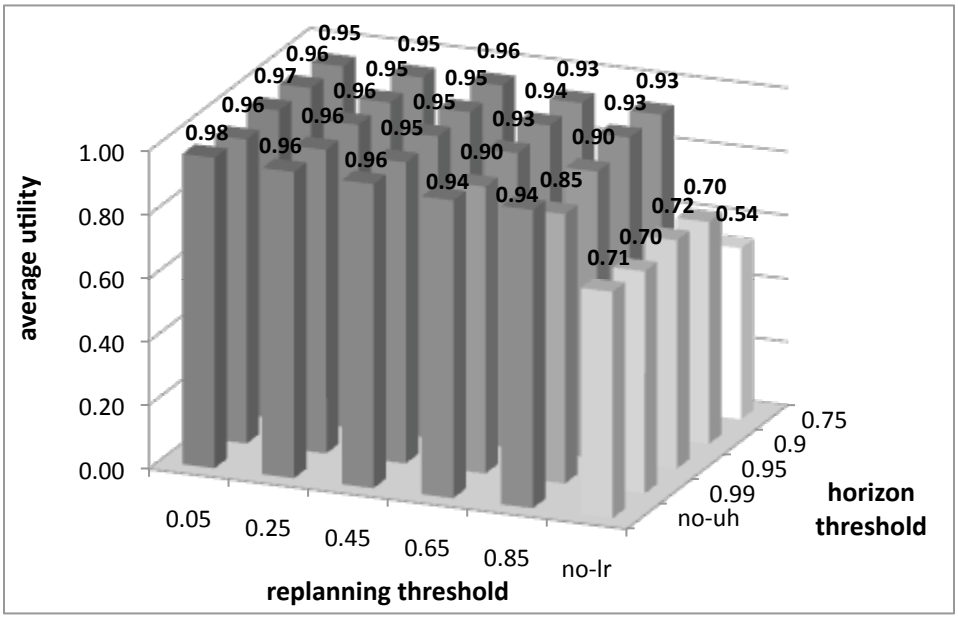


Figure 7.4: Average utility across all three problems using PPM with both the uncertainty horizon and likely replanning at various thresholds.

understanding of time. Time, *e.g.*, method durations and deadlines, is measured in an abstract measure of "ticks"; for our purposes, one tick equals 5 seconds.

Test Suite

We created a test suite specifically for this series of experiments in order to have problems that were similar to each other structurally, for the benefit of the learning algorithm. The test suite was generated by an in-house problem generator. All problems had 10 agents. The structures of the problems were randomly created according to a variable template. Each problem had, underneath the *sum* taskgroup, two subproblems with a *sum_and uaf*. The subproblems had between 3 and 6 tasks underneath them, and all deadlines were imposed at this level. We call these tasks "windows" for clarity. The windows had either *sum_and* or *sum uafs*, and there were enabling constraints between many of them. Each window had 4 tasks underneath it. All tasks at this level were *max uafs* and had four child methods each. Two of these four had a utility and expected duration of 10; the other two had a utility and expected duration of 5. The methods were also typically assigned between two different agents. In this test suite, methods have deterministic utility, which promotes consistency in the utility of the results. The duration distributions were 3-point distributions: the

highest was 1.3 times the expected duration; the middle was the expected duration; and the lowest was 0.7 times the expected duration. Forty percent of methods had a 20% likelihood of a failure outcome. There were no disablers, facilitators or hinderers.

Several parameters were varied throughout problem generation to get our overall test suite:

- Subproblem overlap - how much the execution intervals of the subproblems overlapped.
- Window overlap - how much the execution intervals of windows overlapped.
- Deadline tightness - how tight the deadlines were at the window level.

Each problem in our test suite was generated with a different set of values for these three parameters. The generator then constructed problems randomly, using the parameters as a guide to determine deadlines, etc., according to the structure above. Table 7.1 shows the main features of each of the problems, sorted by the number of methods and average number of method outcomes. In general, the problems have between 128 and 192 methods in the activity network, with roughly 4 execution outcomes possible for each method. The table also shows the number of max-leaf tasks, tasks overall, enables NLEs, and ticks for each problem. The "ticks" field specifies the maximum length of execution for each problem; recall that method durations, deadlines, etc. for this domain are represented in terms of ticks, and that for our purposes, one tick equals 5 seconds.

Analysis of Probabilistic Meta-Level Control

We performed two experiments in which we try to characterize the benefits gained by using our probabilistic meta-level control strategy. For our first experiment, we tested probabilistic meta-level control alone, without schedule strengthening. We ran trials under three different control algorithms:

1. control-baseline - replanning according to the baseline, deterministic plan management (described in Section 7.3.1).
2. control-always - replanning in response to every update.
3. control-likely - replanning according to the probabilistic meta-level control strategy.

The control-always condition was included to provide an approximate upper bound for how much replanning could be supported during execution; we are most interested in the difference between the control-baseline and control-likely conditions to measure the effectiveness of probabilistic meta-level control.

Table 7.1: Characteristics of the 20 generated test suite problems.

	methods	**avg # method outcomes**	**max-leafs**	**tasks**	**enable NLEs**	**ticks**
1	128	3.91	32	43	2	125
2	128	4.01	32	43	2	164
3	128	4.01	32	43	2	232
4	128	4.10	32	43	2	210
5	128	4.17	32	43	2	131
6	128	4.20	32	43	2	183
7	128	4.20	32	43	2	79
8	128	4.24	32	43	2	76
9	128	4.27	32	43	2	120
10	144	4.33	36	48	3	93
11	144	4.33	36	48	3	233
12	160	4.03	40	53	4	101
13	160	4.11	40	53	4	80
14	160	4.14	40	53	4	164
15	160	4.14	40	53	4	233
16	160	4.16	40	53	4	212
17	160	4.22	40	53	4	221
18	176	4.28	44	58	5	150
19	176	4.50	44	58	5	169
20	192	4.08	48	63	6	232

Table 7.2: Average amount of effort spent managing plans under various probabilistic meta-level control strategies.

	PADS time	replanning time	num replans	num update messages
control-baseline	0	5.9	40.8	114.2
control-always	0	17.4	117.8	117.8
control-likely	1.7	1.3	10.2	149.0

Agent threads were distributed among 10 3.6 GHz Intel computers, each with 2 GB RAM. For each problem and condition combination, we ran 10 simulations, each with their own stochastically chosen durations and outcomes. To ensure fairness across conditions, these simulations were purposefully seeded such that the same set of durations and outcomes were executed by each of the conditions of the experiment.

Table 7.2 shows how much effort was spent replanning for each of the conditions. It specifies the average time spent on PADS by each agent updating profiles (which is, of course, non-zero only for the control-likely case). It also shows the average number of replans performed by each agent, as well as the number of update messages received; recall that agents decide whether to replan in response to update messages (Section 7.1.3). The table shows that the control-always condition performs almost three times as many replans as does the control-baseline condition, which replans in response to 36% of update messages received. The control-likely condition, on the other hand, performs roughly a quarter of the number of replans as the control-baseline condition, and replans in response to 7% of update messages received, reducing the plan management time by 49%. Note how PADS causes more messages to be passed during execution, as probability profiles tend to change more frequently than their deterministic counterparts. Despite the overhead that PADS incurs, the overall effect of the probabilistic meta-level control algorithm is a significant decrease in computational effort.

Figure 7.5 shows the average utility earned by each condition. The utility is expressed as the fraction of the omniscient utility earned during execution. We define the omniscient utility for a given simulation to be the utility produced by a deterministic, centralized version of the planner on omniscient versions of the problems, where the actual method durations and outcomes generated by the simulator are given as inputs. As the planner is heuristic, the fraction of omniscient utility earned can be greater than one. Problem numbers correspond to the numbers in Table 7.1, and the problems are sorted by how well the control-baseline condition performs; this roughly correlates

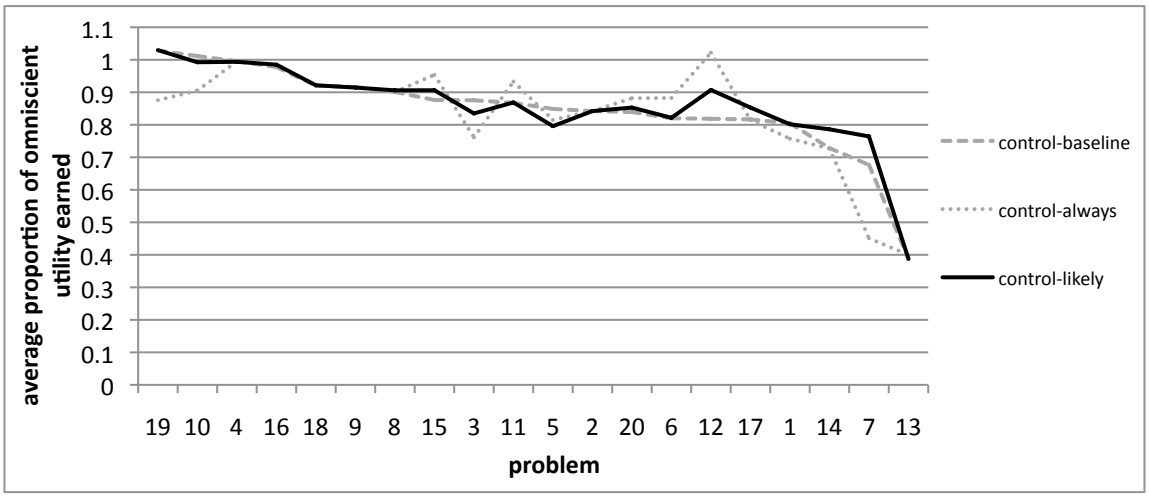


Figure 7.5: Average proportion of omniscient utility earned for various probabilistic meta-level control strategies.

to sorting by increasing problem difficulty.

An important feature of this graph is the difference between control-baseline and control-always. We had expected control-always to always outperform the control-baseline condition. Although the two conditions are not significantly different, their behavior disproves our conjecture: control-always does outperform control-baseline on some of the problems, but there are several in which it performs worse than the control-baseline condition. This is, in part, due to control-always replanning at inopportune times and missing the opportunity to respond to an important update (such as an important, remote method failing), as the system is unresponsive to updates during replans. On average, in fact, control-always earns slightly less utility than the control-baseline condition, although not significantly so.

In contrast, the control-likely condition slightly outperforms the control-baseline condition with respect to utility, although again without a significant difference. These results establish the control-likely condition as the best option. By using probabilistic meta-level control, deliberation during execution is reduced by 49%, while still maintaining high-utility schedules.

Our second experiment tested probabilistic meta-level control in conjunction with probabilistic schedule strengthening. For each problem in the generated test suite, we ran simulations executing the problem under a variety of control algorithms:

1. PPM-always - replanning and strengthening in response to every update.

2. PPM-change - replanning and strengthening whenever anything in the local schedule's deterministic state changes, or when a remote max-leaf task's deterministic state changes. Otherwise, strengthening (without replanning) when anything in the probabilistic state changes.

3. PPM-baseline - replanning and strengthening whenever the baseline, deterministic plan management system chooses to replan.

4. PPM-likely - replanning and strengthening, or only strengthening, according to the probabilistic meta-level control strategy.

5. PPM-interval - replanning and strengthening, or only strengthening, at fixed intervals, with roughly the same frequency as PPM-likely.

6. PPM-random - replanning and strengthening, or only strengthening, at random times, with roughly the same frequency as PPM-likely. In this condition, a replan and strengthen were programmed to occur automatically if a replan or strengthen had not happened in 2.5 times the expected interval between replans.

The PPM-always and PPM-change conditions were included to provide approximate upper bounds for how much replanning and strengthening could be supported during execution. We conjectured that PPM-likely would have a considerable drop in computational effort as compared to these conditions, while meeting or exceeding their average utility; after seeing the above results for control-always, we expected PPM-always and PPM-change to be very adversely affected by their large computational demands, which far exceed those of control-always.

PPM-interval and PPM-random, on the other hand, were included to test our claim that PPM-likely intelligently limits replanning by correctly identifying *when* to replan. Each condition replanned and strengthened, or only strengthened, with roughly the same frequency as PPM-likely, but not necessarily at the same times. The frequency was determined with respect to the number of updates received. For each simulation performed by PPM-likely, we calculated the ratio of replans and strengthens performed to the number of updates received. Separate ratios were calculated for replanning and strengthening together, and for strengthening only. Then, for each problem and each agent, we conservatively took the maximum of these ratios. We then passed the frequencies to the appropriate agents during execution so they could use them in their control algorithm. The design decision to base the frequency of replanning on updates instead of time was made with the belief that this would prove to be the stronger strategy, as replans and strengthens would be more likely to be performed at highly dynamic times.

Agent threads were distributed among 10 3.6 GHz Intel computers, each with 2 GB RAM. For each problem and condition combination, we ran 10 simulations, each with their own stochasti-

Table 7.3: Average amount of effort spent managing plans under various probabilistic meta-level control strategies using strengthening.

	PADS time	strengthen time	replan time	num only strengthens	num replans and strengthens
PPM-always	2.3	79.1	31.9	0	140.3
PPM-change	2.1	55.1	22.1	4.0	66.1
PPM-baseline	2.1	38.8	13.7	0	64.9
PPM-likely	1.9	20.5	3.2	10.4	9.4
PPM-interval	2.2	25.7	4.8	13.9	13.4
PPM-random	2.2	26.5	4.6	13.8	14.0

cally chosen durations and outcomes. To ensure fairness across conditions, these simulations were purposefully seeded such that the same set of durations and outcomes were executed by each of the conditions of the experiments.

Table 7.3 shows how much effort was spent during plan management for each of the conditions. The table shows the average time spent by PADS updating profiles, strengthening schedules, and replanning by each agent. It also shows the average number of schedule strengthenings, and replans and strengthenings, each agent performed; note that the column showing the number of schedule strengthenings specifies the number of times the classifier returned "strengthen only," and does not include strengthening following a replan.

The results are what we expected. Clearly, PPM-always provides an upper bound on computation time, spending a total of 113.3 seconds managing execution. The average time spent on each replan for this condition, however, is much lower that the average for all the other conditions; this is because it replans and strengthens even when nothing in the deterministic or probabilistic state has changed, and such replans and strengthens tend to be faster than replans and strengthens which actually modify the schedule. PPM-change is next in terms of computation time (79.3 seconds), followed by PPM-baseline (54.6 seconds). PPM-likely drastically reduces planning effort compared to these conditions, with a total management time of 25.6 seconds. Note that PPM-interval and PPM-random end up with slightly greater numbers of replans and strengthens than PPM-likely. This is due to their conservative implementation, which uses the maximum of the replanning and strengthening ratios from PPM-likely, described above.

Figure 7.6 shows the average utility earned by each condition. The utility is expressed as the fraction of the omniscient utility earned during execution. We define the omniscient utility for a

given simulation to be the utility produced by a deterministic, centralized version of the planner on omniscient versions of the problems, where the actual method durations and outcomes generated by the simulator are given as inputs. As the planner is heuristic, the fraction of omniscient utility earned can be greater than one. Problem numbers correspond to the numbers in Table 7.1, and the problems are sorted by how well the PPM-always condition did.

The conditions PPM-always, PPM-change, PPM-baseline and PPM-likely are very close for all of the problems, with PPM-likely ending up earning only a little more utility, on average. Surprisingly, and in contrast to the control-always condition (Figure 7.5), PPM-always and PPM-change did not seem to be as adversely affected by their high computational load. Looking into this further, we discovered that even though these conditions do suffer from lack of responsiveness, utility in general does not suffer very much because their schedules are strengthened, protecting them against execution outcomes that would otherwise have led to failure in the face of their unresponsiveness. This highlights one of the benefits of probabilistic schedule strengthening: that it protects against agents not being responsive. Using both replanning and probabilistic schedule strengthening together truly does allow agents to take advantage of opportunities that may arise during execution while also protecting their schedules against unexpected or undesirable outcomes.

Along these lines, however, we do expect that if we lessened the amount of seconds that pass per tick (currently, that value is 5), PPM-always, and perhaps even PPM-baseline, would also be adversely affected by their high computational loads and probabilistic schedule strengthening would not be able to sufficiently protect them from the resulting utility loss. We expect, therefore, that in such a situation probabilistic meta-level control would become a necessity for executing schedules with high utility, instead of only reducing the amount of computation.

Using a repeated measures ANOVA, no combination of PPM-always, PPM-change, PPM-baseline or PPM-likely is statistically significantly different. Each of these conditions, however, is significantly better than both PPM-interval and PPM-random with $p < 0.001$; similarly, PPM-interval and PPM-random are significantly different from each other with the less confident p value of $p < 0.09$, with PPM-interval being the worse condition. Notice that PPM-likely is significantly better than PPM-interval and PPM-random despite replanning and strengthening fewer times, furthering our claim of PPM-likely's dominance over these strategies.

These results confirm that PPM-likely does, in fact, intelligently limit replanning and strengthening, eliminating likely wasteful and unnecessary planning effort during execution. The results of PPM-interval and PPM-random, in addition, confirm our claim that PPM-likely intelligently chooses not only how much to replan, but also *when* to replan: although they performed slightly more replans and strengthens on average than PPM-likely, on a number of problems their performance was much worse due to their unfocused manner of replanning and strengthening. Overall,

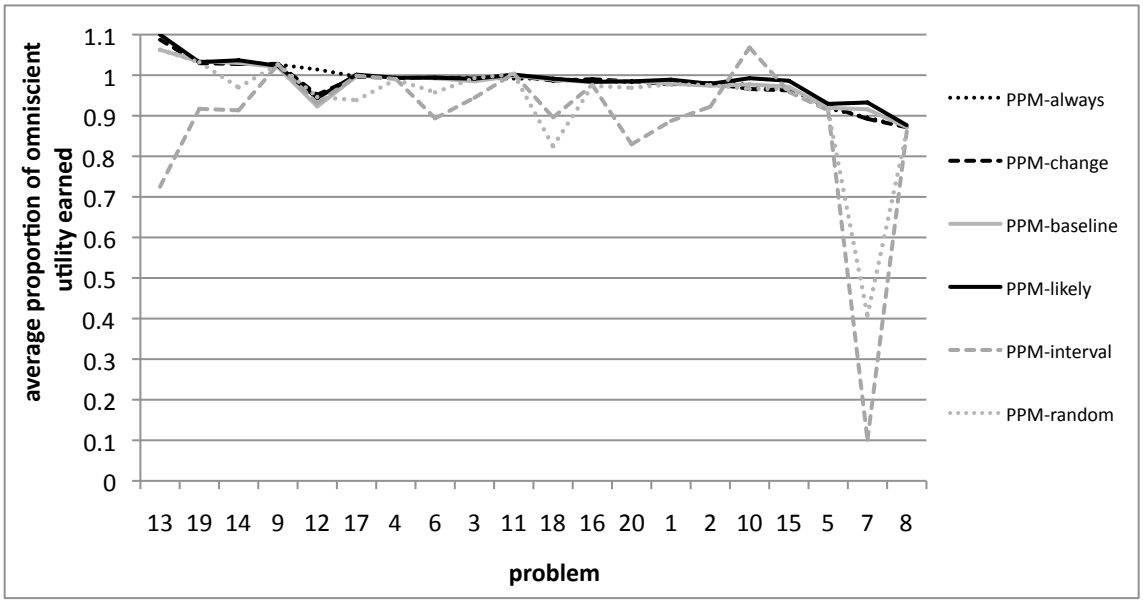


Figure 7.6: Average proportion of omniscient utility earned for various probabilistic meta-level control strategies.

the benefit of probabilistic meta-level control is clear. When PPM-likely is used, the amount of time spent replanning and schedule strengthening is decreased by 79% as compared to PPM-always, and 55% as compared to PPM-baseline, with no significant difference in utility earned.

Analysis of Probabilistic Plan Management

To highlight the overall benefit of PPM, we ran an experiment comparing PPM-likely with four other conditions:

1. control-baseline - replanning whenever the baseline, deterministic system would have.
2. control-likely - replanning according to the probabilistic meta-level control strategy.
3. ds-baseline - replanning and deterministically strengthening whenever the baseline, deterministic system would replan.
4. PPM-baseline - replanning and (probabilistically) strengthening whenever the baseline, deterministic plan management system chooses to replan.

The first, second and fourth conditions are straightforward, and are repeated from the above experiments. The condition ds-baseline is similar to control-baseline, with the additional step of following every replan with a less informed, deterministic version of schedule strengthening [Gallagher, 2009]. Deterministic strengthening makes two types of modifications to the schedule in order to protect it from activity failure:

- Schedule a back-up method for methods under a max-leaf that either have a non-zero failure likelihood or whose `est` plus their maximum duration is greater than their deadline. The back-up method must have an average utility of at least half the average utility of the original method.
- Swap in a shorter-duration sibling for methods under a max-leaf whose `est` plus their maximum duration is greater than the `lst` of the subsequently scheduled method (on the same local schedule). The sibling method's average utility must be at least half the average utility of the original method.

As with the previous experiment, agent threads were distributed among 10 3.6 GHz Intel computers, each with 2 GB RAM. For each problem and condition combination, we ran 10 simulations, each with their own stochastically chosen durations and outcomes. To ensure fairness across conditions, these simulations were purposefully seeded such that the same set of durations and outcomes were executed by each of the conditions of the experiments.

Table 7.4 shows the average amount of time spent managing plans under the various strategies. It also shows the average number of schedule strengthenings, and replans and strengthenings, each agent performed on average; note that the column showing number of schedule strengthens specifies the number of times the classifier returned "strengthen only," and does not include strengthening following a replan. For the deterministic conditions that do not strengthen, the replans and strengthens column denotes the average number of replans performed.

In these results, PPM-likely performed far fewer replans/strengthens than the deterministic baseline condition (19.8 versus 40.8 for control-baseline); however, the time spent managing plans was much higher. This is because of a higher computational cost to probabilistic schedule strengthening than to replanning for the problems of this test suite. In the previous test suite, agents spent an average of 47.7 seconds replanning in the control condition; for this test suite, however, agents in the control condition spent an average of 6.1 seconds replanning. This large difference is due to the simplified activity network structure of the problems we utilized for this test suite (*i.e.*, no disablers or soft NLEs, etc.). We anticipate that the savings PPM-likely provides would be much more significant for more complicated problems. In contrast, in more complicated problem structures, schedule strengthening time would not increase by as much, as its complexity is more dependent

Table 7.4: Average amount of effort spent managing plans under various probabilistic plan management approaches.

	PADS time	strengthen time	replan time	num only strengthens	num replans (and strengthens)
control-baseline	0	0	5.9	0	40.8
control-likely	1.7	0	1.3	0	10.2
ds-baseline	0	2.8	5.9	0	51.1
PPM-baseline	2.1	38.8	13.7	0	64.9
PPM-likely	1.9	20.5	3.2	10.4	9.4

on the number of methods than it is on the complexity of the activity network. Still, even in this domain, PADS-likely incurs an average overhead of only 17.8 seconds over control-baseline – a minute overhead when compared to an average execution time of 802 seconds.

Figure 7.7 shows the average utility earned by each problem. The utility is expressed as the fraction of the omniscient utility earned during execution. We define the omniscient utility for a given simulation to be the utility produced by a deterministic, centralized version of the planner on omniscient versions of the problems, where the actual method durations and outcomes generated by the simulator are given as inputs. As the planner is heuristic, the fraction of omniscient utility earned can be greater than one. Problem numbers correspond to the numbers in Table 7.1, and the problems are sorted by how well the control condition did; this roughly correlates to sorting the problems by increasing problem difficulty.

The deterministic strengthening condition, ds-baseline, performs on average better than the control-baseline condition. Ultimately, though, it fails to match PPM-baseline or PPM-likely in utility, because its limited analysis does not identify all cases where failures might occur and because it does not determine whether its modifications ultimately help the schedule.

PPM-likely clearly outperforms, on average, all of the deterministic conditions. It does underperform ds-baseline for problem 8; the utility difference, however, was less than 5 utility points. Figure 7.8 explicitly shows this, specifying the absolute average utility earned by each condition. On average, PPM-likely earned 13.6% more utility than the control-baseline condition, and 8.1% more than the ds-baseline condition, and is the clear winner.

Using a repeated measures ANOVA, there is no statistically significant difference between PPM-likely and PPM-baseline; these conditions, however, are significantly different from each of

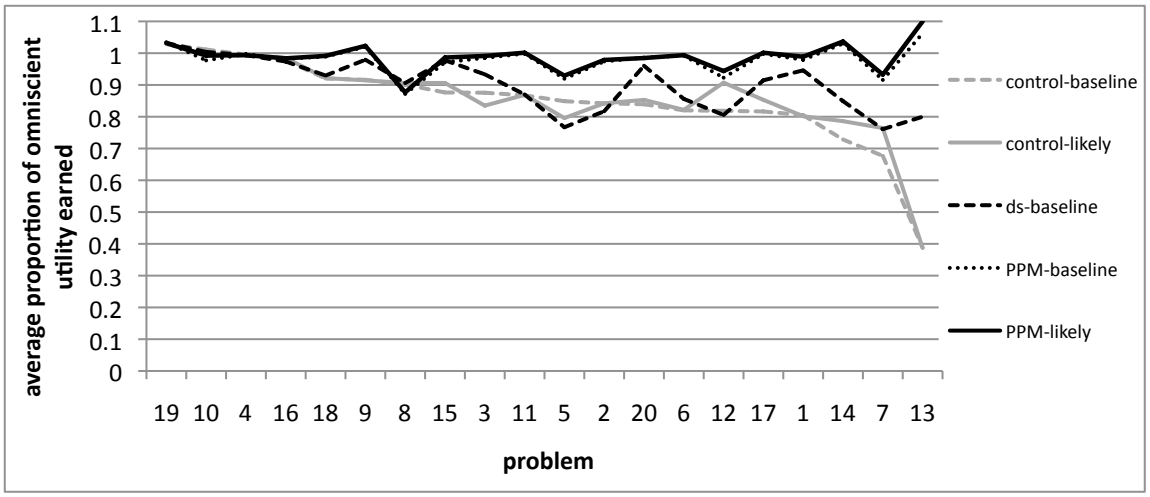


Figure 7.7: Average proportion of omniscient utility earned for various plan management approaches.

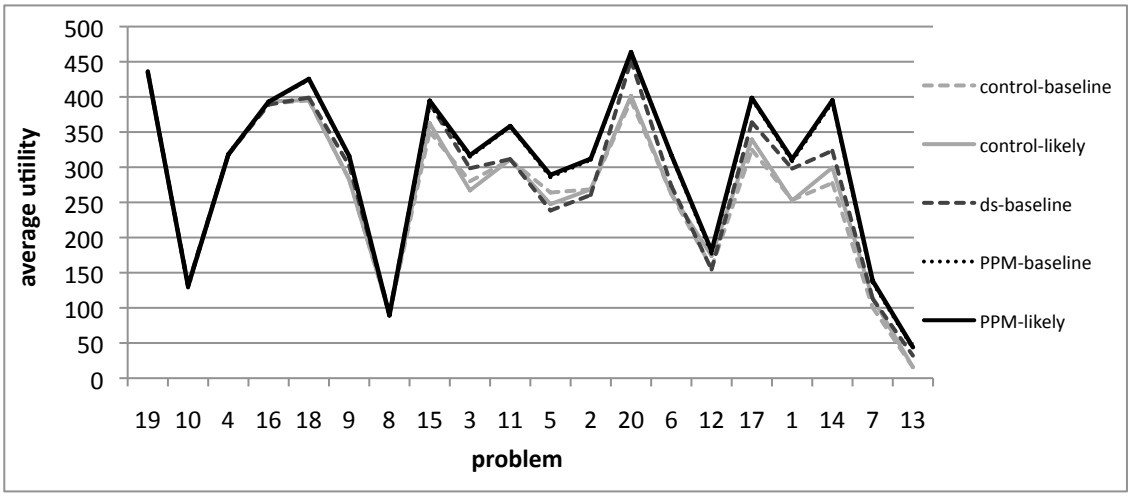


Figure 7.8: Average utility earned for various plan management approaches.

Table 7.5: Average size of communication log file.

	avg log size (MB)
control-baseline	0.29
control-always	0.31
control-likely	0.71
PPM-always	0.92
PPM-change	0.90
PPM-baseline	0.96
PPM-likely	0.79

the other conditions with $p < 0.001$. The control-baseline and control-likely conditions are not significantly different. The ds-baseline condition is significantly different from each of the other conditions with $p < 0.001$.

In distributed scenarios, PPM increases the amount of communication between agents, as agents must communicate more information about each activity. Communicating such information also increases the frequency of communication, as probability profiles change more frequently than deterministic information (as was suggested by Table 7.2). Additionally, communication happens more often because schedules are changing more frequently due to the added step of schedule strengthening. Table 7.5 shows the average amount of communication that occurred for each agent, expressed as the average file size (in megabytes) of a text log file that recorded all outgoing messages across agents.

The table suggests that the main source of communication increase is communicating the probability profiles, not the added step of schedule strengthening. We determine this by comparing the large difference in the amount of communication of control-baseline versus PPM-baseline (0.29MB versus 0.96MB), with the small difference between the amount of communication for control-likely versus PPM-likely (0.71MB versus 0.79MB). The difference between the PPM-likely and PPM-always conditions also shows that probabilistic meta-level control is able to decrease communication somewhat by limiting the frequency of replanning and probabilistic schedule strengthening. In these experiments, the level of communication was within reasonable range for our available bandwidth. If not, communication could be explicitly limited as described in Sections 5.3.5 and 8.1.1.

Overall, these results highlight the benefit of probabilistic plan management. When probabilistic meta-level control is used alone, it can greatly decrease the computational costs of plan

management for this domain. PPM, including both probabilistic schedule strengthening and probabilistic meta-level control, is capable of significantly increasing the utility earned at a modest increase in computational effort for this domain, and achieves its goal effectively managing plans during execution in a scalable way.

Chapter 8

Future Work and Conclusions

8.1 Future Work

Our current system of probabilistic plan management has been shown to be effective in the management of the execution of plans for uncertain, temporal domains. There are other ways to use the probabilistic information provided by PADS, however, that remain to be explored. We discuss below some main areas in which we believe it would be beneficial to extend PPM in the future.

8.1.1 Meta-Level Control Extensions

Although we do not explicitly describe it as such, probabilistic schedule strengthening is an anytime algorithm where the benefit of the initial effort is high and tapers off as the algorithm continues due to the prioritization of strategies and activities by expected utility loss. In this thesis, we do not limit its computation time, and let it run until completion; we found this to be effective for our domains. If it is necessary to limit schedule strengthening computation, we previously suggested imposing a fixed threshold on the po values of activities considered (Section 6.2.1). A more dynamic approach would be to utilize a meta-level control technique (*e.g.*, [Russell and Wefald, 1991, Hansen and Zilberstein, 2001b]) to adaptively decide whether further strengthening was worth the effort.

As implemented, our meta-level control strategies ignore the cost of computation, and focus instead on its possible benefit. This works well to lessen computation, and is sufficient for domains, such as the multi-agent domain we present, where deliberation and execution occur at the same time. It would be interesting to see, however, whether our approach could be combined with

one such as in [Raja and Lesser, 2007, Musliner et al., 2003] to fully integrate both domain and computation probabilistic models into a meta-level control strategy. This would involve using our probabilistic state information as part of an MDP model that chooses deliberative actions based on both the state of the current schedule and performance profiles of deliberation.

We also did not explicitly consider the cost of communication, as we assume sufficient bandwidth in our multi-agent domain. Although we propose a simple way to limit communication, by only communicating changes with a minimum change in probability profile, in situations with a firm cap on bandwidth more thorough deliberation may be useful. One example strategy would be to communicate only probability changes that are likely to cause a planning action in another agent; this would involve a learning process similar to how we learn when an agent should replan based on the updates it receives. An alternate strategy would be to reason about which information is most important to pass on at a given time, and which can be reserved for later.

Similarly, reasoning could be performed to decide what level of coordination between agents is useful, and when it would be useful [Rubinstein et al., 2010, Raja et al., 2010]. For example, if one agent is unable to improve its expected execution outcome via probabilistic schedule strengthening, in oversubscribed settings it may be useful to coordinate with other agents to change which portions of the task tree are scheduled, with the goal of finding tasks to execute that have a better expected outcome. This coordinated probabilistic schedule strengthening could be achieved by agents communicating with each other how they expect their schedules to be impacted by hypothetical changes, with the group as a whole committing to the change which results in the highest overall expected utility.

8.1.2 Further Integration with Planning

An interesting area of exploration would be to integrate PPM with the deterministic planner being used, instead of layering PPM on top of the plans the planner produces. Although many deterministic planners have been augmented to handle decision-theoretic planning (*e.g.*, [Blythe, 1999, Kushmerick et al., 1995, Blythe, 1998]), most have done so at the expense of tractability. In accordance with the philosophy of this approach, however, PADS could be integrated with the planning process at a lower level, improving the search without fully integrating probability information into the planner. A simple example of this in the single-agent domain using PADS to determine which child of a ONE task should be scheduled to maximize expected utility. PADS could also help to guide the search at key points by using the expected utility loss and expected utility gain heuristics to identify activities that are profitable to include in the schedule. For example, in the multi-agent domain, when choosing between two children of an OR *max* task, the eug heuristic could be used

to suggest which would have the highest positive impact on taskgroup utility.

8.1.3 Demonstration on Other Domains / Planners

One extension of this work that we would like to see is applying it to other planners in order to demonstrate its principles in another setting. For example, we would like to use our approach in conjunction with FF-Replan [Yoon et al., 2007] in order see what improvement could be garnered. In the problems FF-Replan considers, uncertainty is in the effects of actions. In its determinization of these problems, FF-Replan uses an "all-outcomes" technique that expands a probabilistic action such that there is one deterministic action available to the planner for each possible probabilistic effect of the original action. Planning is then done based on these deterministic actions. One possible way FF-Replan could benefit from our approach is to analyze the current deterministic plan, find a likely point of failure, and then try switching to alternate deterministic actions to try to bolster the probability of success.

8.1.4 Human-Robot Teams

Although, as is, PPM could be used to manage the execution of plans for teams of both human and robotic agents, it could also be extended to specifically tailor to those types of situations. First, in many situations it would be appropriate to include humans' preferences into the plan objective, complicating the analysis. For example, humans may like their job more if they are frequently assigned tasks that they enjoy performing. Other modifications to the objective function include discounting utility earned later in the plan or discounting utility once a certain amount has been earned. This would reflect, for example, a human getting tired and assigning a higher cost to execution as time goes on. These more complex plan objective functions would make schedule strengthening more difficult since it is harder to predict what types of actions will bolster low-utility portions of the plan.

Additionally, including humans in the loop introduces the possibility of having plan stability as a component of the objective function [van der Krogt and de Weerdt, 2005]. Ideally, in human-robot scenarios, plans should remain relatively stable throughout execution in order to facilitate continued human understanding of the plan as well as agent predictability. Instead of using meta-level control to limit only the frequency of replanning, which implicitly encourages plan stability, it could be used to explicitly encourage plan stability and limit the number of times (and the degree to which) the plans change overall. It could also be used to choose the best times to replan; an example of this is discouraging replanning a human's schedule at times when the human is likely

mentally overloaded, and favoring replanning at a time when the human has the capacity to fully process and understand the resulting changes.

8.1.5 Non-Cooperative Multi-Agent Scenarios

Our multi-agent work assumes that agents are cooperative and work together to maximize joint plan utility. Breaking that assumption entails rethinking much of the PPM strategies. The distributed PADS algorithm and basic communication protocol, for example, would need to change as agents may not have incentive to share all of their probability profile information. Instead, agents may share information or query other agents for information if they feel that it would raise their expected utility [Gmytrasiewicz and Durfee, 2001]. Negotiation would also be a possibility for an information sharing protocol. Ideally, agents would track not only the expected benefit from receiving information, but also whether sharing information would have an expected cost; such metrics could be used to decide whether an information trade was worth while.

Similar ideas apply to probabilistic schedule strengthening. Non-cooperative, self-interested agents have little incentive, for example, to schedule a back-up activity for other agents. Again, negotiation could be one way of achieving coordination, based on agents' projected expected gain in utility.

8.2 Conclusions

This thesis has described an approach to planning and execution for uncertain domains called *Probabilistic Plan Management* (PPM). The thesis claims that explicitly analyzing the probabilistic state of deterministic plans built for uncertain domains provides important information that can be leveraged in a scalable way to effectively manage plan execution, providing some of the benefits of non-deterministic planning while avoiding much of its computational overhead. PPM relies on a probabilistic analysis of deterministic schedules to inform probabilistic schedule strengthening and to act as the basis of a probabilistic meta-level control strategy. We support this thesis by developing algorithms for each of the three components of the approach, in the contexts of a single-agent domain with temporal constraints and activity durational uncertainty and a multi-agent domain with a rich set of activity constraints and sources of uncertainty.

We developed PADS, a general way to probabilistically analyze deterministic schedules. PADS assumes that plan execution has a specific objective that can be expressed in terms of utility and that the domain has a known uncertainty model. With this information, PADS analyzes deterministic schedules to calculate their expected utility. It explicitly considers the uncertainty model of the do-

main and propagates probability information throughout the schedule. A novelty of our algorithm is how quickly PADS can be calculated and its results utilized to make plan improvements, due to analytically calculating the uncertainty of a problem in the context of a given schedule. For large, 10-25 agent problems, schedule expected utility can be globally calculated in an average of 0.58 seconds.

We also developed a way to strengthen deterministic schedules based on the probabilistic analysis, making them more robust to the uncertainty of execution. This step is best done in conjunction with replanning during execution to ensure that robust, high-utility schedules are always being executed. Probabilistic schedule strengthening is also best done when the structure of the domain can be exploited to identify classes of strengthening actions that may be helpful in different situations. Through experimentation, we found strengthened schedules to earn 36% more utility, on average, than their initial deterministic counterparts, when both were executed with no replanning allowed. We also found that even in cases where it increases computation time, probabilistic schedule strengthening makes up for the lack of agent responsiveness by ensuring that agents' schedules are strengthened and robust to unexpected outcomes that may occur while an agent is tied up replanning or strengthening.

We investigated the use of a PADS-driven probabilistic meta-level control strategy, which effectively controls planning actions during execution by performing them only when they would likely be beneficial. It identifies cases when there are likely opportunities to exploit or failures to avoid, and replans only at those times. Probabilistic meta-level control reduces unnecessary replanning, and improves agent responsiveness during execution. Experiments in the multi-agent domain showed that this strategy can reduce plan management effort by an average of 49% over approaches driven by deterministic heuristics, with no significant change in utility earned.

As part of this thesis, we performed experiments to test the efficacy of our overall PPM approach. For a single-agent domain, PPM afforded a 23% reduction in schedule execution time (where execution time is defined as the schedule makespan plus the time spent replanning during execution). For a multi-agent, distributed domain, PPM using probabilistic schedule strengthening in conjunction with replanning and probabilistic meta-level control resulted in 14% more utility earned on average, with only a small added overhead.

Overall, probabilistic plan management is able to take advantage of the strengths of nondeterministic planning while, in large part, avoiding its computational burden. It is able to significantly improve the expected outcome of execution while also reasoning about the probabilistic state of the schedule to reduce the computational overhead of plan management. We conclude that probabilistic plan management, including both probabilistic schedule strengthening and probabilistic meta-level control, achieves its goal of being able to leverage the probabilistic information

provided by PADS to effectively manage plan execution in a scalable way.

Chapter 9

Bibliography

George Alexander, Anita Raja, and David J. Musliner. Controlling deliberation in a markov decision process-based agent. In *Seventh International Joint Conference on Autonomous Agents and Multi-Agent Systems AAMAS '08*, 2008. 2.3

George Alexander, Anita Raja, and David Musliner. Controlling deliberation in coordinators. In Michael T. Cox and Anita Raja, editors, *Metareasoning: Thinking about thinking*. MIT Press, 2010. 2.3

N. Fazil Ayan, Ugur Kuter, Fusun Yaman, and Robert P. Goldman. HOTRiDE: Hierarchical ordered task replanning in dynamic environments. In *Proceedings of the 3rd Workshop on Planning and Plan Execution for Real-World Systems (held in conjunction with ICAPS 2007)*, September 2007. 2.2.2

Donald R. Barr and E. Todd Sherrill. Mean and variance of truncated normal distributions. *The American Statistician*, 53(4):357–361, November 1999. 3.3.1, 4.1

J. C. Beck and N. Wilson. Proactive algorithms for job shop scheduling with probabilistic durations. *Journal of Artificial Intelligence Research*, 28:183–232, 2007. 1, 2.1.3

Piergiorgio Bertoli, Alessandro Cimatti, Marco Roverie, and Paolo Traverso. Planning in nondeterministic domains under partial observability via symbolic model checking. In *Proceedings of IJCAI '01*, 2001. 2.1.2

Lars Blackmore, Askar Bektassov, Masahiro Ono, and Brian C. Williams. Robust, optimal predictive control of jump markov linear systems using particles. *Hybrid Systems: Computation and Control*, 2007. 2.1.3

Avrim Blum and John Langford. Probabilistic planning in the graphplan framework. In *Proceedings of the Fifth European Conference on Planning (ECP '99)*, 1999. 2.1.2

Jim Blythe. Decision-theoretic planning. *AI Magazine*, 20(2):37–54, 1999. 8.1.2

Jim Blythe. *Planning under Uncertainty in Dynamic Domains*. PhD thesis, Carnegie Mellon University, 1998. 2.1.2, 8.1.2

Jim Blythe and Manuela Veloso. Analogical replan for efficient conditional planning. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence (AAAI '97)*, Menlo Park, California, 1997. 2.1.2

Mark Boddy and Thomas Dean. Decision-theoretic deliberation scheduling for problem solving in time-constrained environments. *Artificial Intelligence*, 67(2):245–286, 1994. 2.3

Mark Boddy, Bryan Horling, John Phelps, Robert Goldman, Regis Vincent, C. Long, and Bob Kohout. C_TAEMS language specification v. 1.06. October 2005. 3.3.2

Justin A. Boyan and Michael L. Littman. Exact solutions to time-dependent mdps. In *Advances in Neural Information Processing Systems (NIPS)*, 2000. 2.1.1

L. Breiman. Bagging predictors. *Machine Learning*, 24:123–140, 1996. 3.4

Daniel Bryce, Mausam, and Sungwook Yoon. Workshop on a reality check for planning and scheduling under uncertainty (held in conjunction with ICAPS '08). 2008. URL `http://www.ai.sri.com/~bryce/ICAPS08-workshop.html`. 1

Olivier Buffet and Douglas Aberdeen. The factored policy-gradient planner. *Artificial Intelligence*, 173(5-6):722–747, 2009. 2.1.1

Amedeo Cesta and Angelo Oddi. Gaining efficiency and flexibility in the simple temporal problem. In *Proceedings of the Third International Workshop on Temporal Representation and Reasoning*, 1996. 3.3.2

S. Chien, G. Rabideau, R. Knight, R. Sherwood, B. Engelhardt, D. Mutz, T. Estlin, B. Smith, F. Fisher, T. Barrett, G. Stebbins, and D. Tran. Aspen - automated planning and scheduling for space mission operations. In *Space Ops*, Toulouse, June 2000. URL `http://citeseer.ist.psu.edu/chien00aspen.html`. 2.2.2, 2.3, 3.3.1

Alessandro Cimatti and Marco Roveri. Conformant planning via symbolic model checking. *Journal of Artificial Intelligence Research*, 13:305–338, 2000. URL `citeseer.ist.psu.edu/cimatti00conformant.html`. 2.1.3

Michael T. Cox. Metacognition in computation: A selected research review. *Artificial Intelligence*, 169(2):104–141, 2005. 2.3

Richard Dearden, Nicolas Meuleau, Sailesh Ramakrishnan, David E. Smith, and Rich Washington. Incremental contingency planning. In *Procceedings of ICAPS Workshop on Planning under Uncertainty*, 2003. URL `citeseer.ist.psu.edu/dearden03incremental.html`. 2.1.2

Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61–95, May 1991. 2.2.1, 3.2.2

Keith Decker. TAEMS: A framework for environment centered analysis & design of coordination mechanisms. In G. O'Hare and N. Jennings, editors, *Foundations of Distributed Artificial Intelligence*, chapter 16, pages 429–448. Wiley Inter-Science, 1996. 3.3.2

Denise Draper, Steve Hanks, and Daniel Weld. Probabilistic planning with information gathering and contingent execution. In *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems (AIPS)*, Menlo Park, California, 1994. 2.1.2

Mark Drummond, John Bresina, and Keith Swanson. Just-in-case scheduling. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, 1994. URL `citeseer.ist.psu.edu/drummond94justcase.html`. 2.1.2

Kutluhan Erol, James Hendler, and Dana S. Nau. Htn planning: Complexity and expressivity. In *Proceedings of AAAI-94*, 1994. 3.2.1

Tara Estlin, Rich Volpe, Issa Nesnas, Darren Mutz, Forest Fisher, Barbara Engelhardt, and Steve Chien. Decision-making in a robotic architecture for autonomy. In *Proceedings of iSAIRAS 2001*, Montreal, CA, June 2001. 2.3

Zhengzhu Feng, Richard Dearden, Nicolas Meuleau, and Richard Washington. Dynamic programming for structured continuous markov decision problems. In *Proceedings of the Twentieth Conference on Uncertainty in Artificial Intelligence (UAI '04)*, 2004. 2.1.1

Janae Foss, Nulifer Onder, and David E. Smith. Preventing unrecoverable failures through precautionary planning. In *Workshop on Moving Planning and Scheduling Systems into the Real World (held in conjunction with ICAPS '07)*, 2007. 2.1.2

Maria Fox, Richard Howey, and Derek Long. Exploration of the robustness of plans. In *Proceedings of AAAI-06*, 2006. 2.1.3

Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139, 1997. 3.4

Christian Fritz and Shiela McIlraith. Computing robust plans in continuous domains. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS)*, 2009. 2.1.3

Anthony Gallagher. *Embracing Conflicts: Exploiting Inconsistencies in Distributed Schedules using Simple Temporal Network Representation*. PhD thesis, Carnegie Mellon University, 2009. 1.2, 1.2, 2.2.1, 4.3, 7.4.2

Anthony Gallagher, Terry L. Zimmerman, and Stephen F. Smith. Incremental scheduling to maximize quality in a dynamic environment. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS '06)*, 2006. 2.2.2

Alfonso Gerevini and Ivan Serina. Fast plan adaptation through planning graphs: Local and systematic search techniques. In *Proceedings of the International Conference on Artificial Intelligence Planning Systems (AIPS '00)*, 2000. 2.2.2

Piotr J. Gmytrasiewicz and Edmund H. Durfee. Rational communication in multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 4(3):233–273, 2001. 8.1.5

Robert P. Goldman and Mark S. Boddy. Expressive planning and explicit knowledge. In *AIPS*, 1996. 2.1.3

Charles M. Grinstead and J. Laurie Snell. *Introduction to Probability*. American Mathematical Society, 1997. 3.3.2

Eric A. Hansen and Shlomo Zilberstein. LAO*: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence*, 129(1-2):35–62, 2001a. URL `http://rbr.cs.umass.edu/shlomo/papers/HZaij01b.html`. 2.3

Eric A. Hansen and Shlomo Zilberstein. Monitoring and control of anytime algorithms: A dynamic programming approach. *Artificial Intelligence*, 126(1-2):139–157, 2001b. URL `http://rbr.cs.umass.edu/shlomo/papers/HZaij01a.html`. 2.3, 8.1.1

Laura M. Hiatt and Reid Simmons. Torwards probabilistic plan management. In *Proceedings of the 3rd Workshop on Planning and Plan Execution for Real-World Systems (held in conjunction with ICAPS 2007)*, Providence, RI, September 2007. URL `http://www.cs.drexel.edu/~pmodi/papers/sultanik-ijcai07.pdf`. 5.2, 7.2, 7.4.1

Laura M. Hiatt, Terry L. Zimmerman, Stephen F. Smith, and Reid Simmons. Reasoning about executional uncertainty to strengthen schedules. In *Proceedings of the Workshop on A Reality Check for Planning and Scheduling Under Uncertainty (held in conjunction with ICAPS-08)*, 2008. 2, 5.3, 6.1, 6.4.1

Laura M. Hiatt, Terry L. Zimmerman, Stephen F. Smith, and Reid Simmons. Strengthening schedules through uncertainty analysis. In *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence (IJCAI-09)*, 2009. 2, 5.3, 6.1, 6.4.2

Jorg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Hournal of Artificial Intelligence Research*, 14:263:302, 2001. 2.2

Ronald A. Howard. *Dynamic Programming and Markov Processes*. MIT Press, 1960. 2.1, 2.1.1

Ronald A. Howard. *Dynamic Probabilistic Systems: Semi-Markov and Decision Processes*. John Wiley & Sons, New York, New York, 1971. 2.1.1

Sergio Jiménez, Andrew Coles, and Amanda Smith. Planning in probabilistic domains using a deterministic numeric planner. In *Proceedings of the 25th Workshop of the UK Planning and Scheduling Special Interest Group (PlanSIG 2006)*, December 2006. 2.2.1

Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101, 1998. 2.1.1

Nicholas Kushmerick, Steve Hanks, and Daniel Weld. An algorithm for probabilistic planning. *Artificial Intelligence*, 76(1-2):239–286, 1995. 2.1.3, 8.1.2

P. Laborie and M. Ghallab. Planning with sharable resource constraints. In *Proceedings of IJCAI-05*, Montreal, Canada, 1995. URL `http://dli.iiit.ac.in/ijcai/IJCAI-95-VOL2/PDF/081.pdf`. 2.2.1

Hoong Chuin Lau, Thomas Ou, and Fei Xiao. Robust local search and its application to generating robust local schedules. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS '07)*, September 2007. 2.2.1

Thomas Léauté and Brian C. Williams. Coordinating agile systems through the model-based execution of temporal plans. In *Proceedings of the Twentieth National Conference on Artif icial Intelligence (AAAI '05)*, 2005. 2.3

Lihong Li and Michael L. Littman. Lazy approximation for solving continuous finite-horizon MDPs. In *Proceedings of the Twentieth American National Conference on Artificial Intelligence (AAAI '05)*, 2005. 2.1.1

Iain Little and Sylvie Thiébaux. Concurrent probabilistic planning in the graphplan framework. In *Proceedings of ICAPS '06*, 2006. 2.1.2

Iain Little, Douglas Aberdeen, and Sylvie Thiébaux. Prottle: A probabilistic temporal planner. In *Proceedings of the Twentieth American National Conference on Artificial Intelligence (AAAI '05)*, Pittsburgh, PA, July 2005. 2.1.2

A. K. Machworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977. 3.3.1

Rajiv T. Maheswaran and Pedro Szekely. Criticality metrics for distributed plan and schedule management. In *Proceedings of the 18th International Conference on Automated Planning and Scheduling (ICAPS)*, 2008. 2.2.2

Janusz Marecki, Sven Koenig, and Milind Tambe. A fast analytical algorithm for solving markov decision processes with real-valued resources. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2007. 2.1.1

Mausam and Daniel S. Weld. Probabilisitic temporal planning with uncertain durations. In *Proceedings of AAAI-06*, Boston, MA, July 2006. 2.1.1

Mausam and Daniel S. Weld. Concurrent probabilistic temporal planning. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS '05)*, 2005. 2.1.1

K. N. McKay, T. E. Morton, P. Ramnath, and J. Wang. Aversion dynamics' scheduling when the system changes. *Journal of Scheduling*, 3(2), 2000. 1

Nicolas Meuleau and David E. Smith. Optimal limited contingency planning. In *Workshop on Planning under Uncertainty (held in conjunction with ICAPS '03)*, 2003. 2.1.2

Paul Morris, Nicola Muscettola, and Thierry Vidal. Dynamic control of plans with temporal uncertainty. In *Proceedings of IJCAI*, pages 494–502, 2001. 1, 2.2.1

David J. Musliner, E, and Kang G. Shin. CIRCA: A cooperative intelligent real-time control architecture. *IEEE Transactions on Systems, Man, and Cybernetics*, 23(6), 1993. 2.1.3

David J. Musliner, Robert P. Goldman, and Kurt D. Krebsbach. Deliberation scheduling strategies for adaptive mission planning in real-time environments. In *Third International Workshop on Self Adaptive Software*, 2003. 2.3, 8.1.1

David J. Musliner, Edmund H. Durfee, Jianhui Wu, Dmitri A. Dolgov, Robert P. Goldman, and Mark S. Boddy. Coordinated plan management using multiagent MDPs. In *Working Notes of the AAAI Spring Symposium on Distributed Plan and Schedule Management*, March 2006. 2.1.1

David J. Musliner, Jim Carciofini, Edmund H. Durfee, Jianhui Wu, Robert P. Goldman, and Mark S. Boddy. 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-07)*, September 2007. 2.1.1

Nulifer Onder and Martha Pollack. Conditional, probabilistic planning: A unifying algorithm and effective search control mechanisms. In *Proceedings of the 16th National Conference on Artificial Intelligence*, 1999. 2.1.2

Nulifer Onder, Garrett C. Whelan, and Li Li. Engineering a conformant probabilistic planner. *Journal of Artificial Intelligence Research*, 25:1–15, 2006. 2.1.3

Mark Alan Peot. *Decision-Theoretic Planning*. PhD thesis, Department of Engineering-Economic Systems, Stanford University, 1998. 2.1.2

Nicola Policella, Amedeo Cesta, Angelo Oddi, and Stephen F. Smith. From precedence constraint posting to partial order schedules. A CSP approach to robust scheduling. *AI Communications*, 20(3):163–180, 2007. 1, 2.2.1

Nicola Policella, Amedeo Cesta, Angelo Oddi, and Stephen F. Smith. Solve-and-robustify. Synthesizing partial order schedules by chaining. *Journal of Scheduling*, 12(3):299–314, June 2009. 3.2.2

Martha E. Pollack and John F. Horty. There's more to life than making plans: Plan management in dynamic, multi-agent environments. *AI Magazine*, 20(4):71–84, 1999. 2.2.2

J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, 1993. 3.4

J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986. 3.4

Anita Raja and Victor Lesser. A framework for meta-level control in multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 15(2):147–196, 2007. 1.3, 2.3, 8.1.1

Anita Raja, George Alexander, Victor Lesser, and Mike Krainin. Coordinating agents' meta-level control. In Michael T. Cox and Anita Raja, editors, *Metareasoning: Thinking about thinking*. MIT Press, 2010. 8.1.1

Zachary Rubinstein, Stephen F. Smith, and Terry Zimmerman. The role of metareasoning in achieving effective multi-agent coordination. In Michael T. Cox and Anita Raja, editors, *Metareasoning: Thinking about thinking*. MIT Press, 2010. 8.1.1

Stuart Russell and Eric Wefald. Principles of metareasoning. *Artificial Intelligence*, 49, 1991. 2.3, 8.1.1

Uwe Schoning. A probabilistic algorithm for k-SAT and constraint satisfaction problems. In *Proceedings of the 40th IEEE Symposium on Foundations of Computer Science*, 1999. 7.4.1

Christoph Schwindt. Generation of resource-constrained project scheduling problems subject to temporal constraints. Technical Report WIOR-543, Institut fur Wirtschaftstheorie und Operations Research, Universit at Karlsruhe, November 1998. 3.3.1, 5.4.1, 7.4.1

Brennan Sellner and Reid Simmons. Towards proactive replanning for multi-robot teams. In *Proceedings of the 5th International Workshop on Planning and Scheduling in Space 2006*, Baltimore, MD, October 2006. 2.2.1, 3

Daniel E. Smith and Daniel S. Weld. Conformant graphplan. In *Proceedings of AAAI-98*, Madison, WI, 1998. 2.1.3

Stephen F. Smith, Anthony Gallagher, Terry Zimmerman, Laura Barbulescu, and Zachary Rubinstein. Multi-agent management of joint schedules. In *Proceedings of the 2006 AAAI Spring Symposium on Distributed Plan and Schedule Management*, March 2006. 2.2.1

Stephen F. Smith, Anthony Gallagher, Terry Zimmerman, Laura Barbulescu, and Zachary Rubinstein. Distributed management of flexible times schedules. In *Proceedings of AAMAS-07*, May 2007. 2.2.2, 3.2.2, 3.3.2, 5.3, 7.3.1

Austin Tate, Brian Drabble, and Richard Kirby. O-Plan 2: an open architecture for command, planning and control. In M. Fox. and M. Zweben, editors, *Knowledge Based Scheduling*. Morgan Kaufmann, 1994. URL `citeseer.ist.psu.edu/tate94oplan.html`. 2.2.2

Roman van der Krogt and Mathijs de Weerdt. Plan repair as an extension of planning. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS '05)*, Monterey, California, 2005. 2.2.2, 8.1.4

Vincent Vidal and Héctor Geffner. Branching and pruning: An optimal temporal POCL planner based on constraint programming. *Artificial Intelligence*, 170(3):298–335, 2006. 2.2.1

Thomas Wagner, Anita Raja, and Victor Lesser. Modeling uncertainty and its implications to sophisticated control in TAEMS agents. *Autonomous Agents and Multi-Agent Systems*, 13(3): 235–292, 2006. 2.1.2

Ian Warfield, Chad Hogg, Stephen Lee-Urban, and Héctor Muñoz-Avila. Adaptation of hierarchicial task network plans. In *FLAIRS '07*, May 2007. 2.2.2

Daniel S. Weld, Corin R. Anderson, and David E. Smith. Extending graphplan to handle uncertainty and sensing actions. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-98)*, 1998. 2.1.2

David E. Wilkins and Marie desJardins. A call for knowledge-based planning. *AI Magazine*, 22: 99–115, 2000. 3.2.1

Ping Xuan and Victor Lesser. Incorporating uncertainty in agent commitments. In *Proceedings of ATAL-99*, 1999. 2.1.2

Sungwook Yoon, Alan Fern, and Robert Givan. FF-Replan: A baseline for probabilistic planning. In *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS '07)*, Providence, RI, September 2007. 2.2, 8.1.3

Håkan L. S. Younes and David J. Musliner. Probabilistic plan verification through acceptance sampling. In Froduald Kabanza and Sylvie Thiébaux, editors, *Proceedings of the AIPS-02 Workshop on Planning via Model Checking*, pages 81–88, Toulouse, France, 2002. URL `http://www.tempastic.org/papers/aipsws2002.pdf`. 2.1.3

Håkan L. S. Younes and Reid G. Simmons. Solving generalized semi-markov decision processes using continuous phase-type distributions. In *Proceedings of AAAI-04*, San Jose, CA, 2004a. URL `http://www.tempastic.org/papers/aaai2004.pdf`. 2.1.1

Håkan L. S. Younes and Reid G. Simmons. VHPOP: Versatile heuristic partial order planner. *Journal of Artificial Intelligence Research*, 20:405–430, 2003. 2.2.1

Håkan L. S. Younes and Reid G. Simmons. Policy generation for continuous-time stochastic domains with concurrency. In *Proceedings of ICAPS '04*, 2004b. URL `citeseer.ist.psu.edu/younes04policy.html`. 2.1.2

Håkan L. S. Younes, David J. Musliner, and Reid G. Simmons. A framework for planning in continuous-time stochastic domains. In *ICAPS*, pages 195–204, 2003. 2.1.1

Ji Zhu, Saharon Rosset, Hui Zou, and Travor Hastie. Multi-class adaboost. Technical Report 430, University of Michigan, 2005. http://www-stat.stanford.edu/ hastie/Papers/samme.pdf. 3.4

Appendix A

Sample Problems

A.1 Sample MRCPSP/max problem in ASPEN syntax

```
activity a001 {
  decompositions = a001_0 or a001_1;
};

activity a001_0 {
  decompositions = a001_0_by_a012_0 or a001_0_by_a012_1;
};

activity a001_0_by_a012_0 {
  duration = 10;
  constraints = ((ends_before start of a012_0 by [1, 1000000]));
};

activity a001_0_by_a012_1 {
  duration = 10;
  constraints = ((ends_before start of a012_1 by [3, 1000000]));
};

activity a001_1 {
  decompositions = a001_1_by_a012_0 or a001_1_by_a012_1;
};

activity a001_1_by_a012_0 {
  duration = 4;
  constraints = ((ends_before start of a012_0 by [1, 1000000]));
};

activity a001_1_by_a012_1 {
  duration = 4;
  constraints = ((ends_before start of a012_1 by [6, 1000000]));
};

activity a002 {
  decompositions =  a002_0 or a002_1;
};

activity a002_0 {
  decompositions = a002_0_by_a015_0_a020_0 or a002_0_by_a015_0_a020_1 or
                   a002_0_by_a015_1_a020_0 or a002_0_by_a015_1_a020_1;
};

activity a002_0_by_a015_0_a020_0 {
  duration = 7;
  constraints = ((ends_before start of a015_0 by [-19, 1000000]) and
                 (ends_before start of a020_0 by [10, 1000000]));
};

activity a002_0_by_a015_0_a020_1 {
  duration = 7;
  constraints = ((ends_before start of a015_0 by [-34, 1000000]) and
                 (ends_before start of a020_1 by [1, 1000000]));
};

activity a002_0_by_a015_1_a020_0 {
```

```
  duration = 7;
  constraints = ((ends_before start of a015_1 by [19, 1000000]) and
                 (ends_before start of a020_0 by [10, 1000000]));
};

activity a002_0_by_a015_1_a020_1 {
  duration = 7;
  constraints = ((ends_before start of a015_1 by [19, 1000000]) and
                 (ends_before start of a020_1 by [1, 1000000]));
};

activity a002_1 {
  decompositions = a002_1_by_a015_0_a020_0 or a002_1_by_a015_0_a020_1 or
                   a002_1_by_a015_1_a020_0 or a002_1_by_a015_1_a020_1;
};

activity a002_1_by_a015_0_a020_0 {
  duration = 2;
  constraints = ((ends_before start of a015_0 by [2, 1000000]) and
                 (ends_before start of a020_0 by [1, 1000000]));
};

activity a002_1_by_a015_0_a020_1 {
  duration = 2;
  constraints = ((ends_before start of a015_0 by [2, 1000000]) and
                 (ends_before start of a020_1 by [4, 1000000]));
};

activity a002_1_by_a015_1_a020_0 {
  duration = 2;
  constraints = ((ends_before start of a015_1 by [6, 1000000]) and
                 (ends_before start of a020_0 by [1, 1000000]));
};

activity a002_1_by_a015_1_a020_1 {
  duration = 2;
  constraints = ((ends_before start of a015_1 by [6, 1000000]) and
                 (ends_before start of a020_1 by [4, 1000000]));
};

activity a003 {
  decompositions =  a003_0 or a003_1;
};

activity a003_0 {
  decompositions = a003_0_by_a010_0_a013_0_a020_0 or a003_0_by_a010_0_a013_1_a020_0 or
                   a003_0_by_a010_1_a013_0_a020_0 or a003_0_by_a010_1_a013_1_a020_0 or
                   a003_0_by_a010_0_a013_0_a020_1 or a003_0_by_a010_0_a013_1_a020_1 or
                   a003_0_by_a010_1_a013_0_a020_1 or a003_0_by_a010_1_a013_1_a020_1;
};

activity a003_0_by_a010_0_a013_0_a020_0 {
  duration = 4;
  constraints = ((ends_before start of a010_0 by [6, 1000000]) and
                 (ends_before start of a013_0 by [-1000000, 16]) and
                 (ends_before start of a020_0 by [2, 1000000]));
};
```

```
activity a003_0_by_a010_0_a013_1_a020_0 {
  duration = 4;
  constraints = ((ends_before start of a010_0 by [6, 1000000]) and
                 (ends_before start of a013_1 by [-1000000, 18]) and
                 (ends_before start of a020_0 by [2, 1000000]));
};

activity a003_0_by_a010_1_a013_0_a020_0 {
  duration = 4;
  constraints = ((ends_before start of a010_1 by [8, 1000000]) and
                 (ends_before start of a013_0 by [-1000000, 16]) and
                 (ends_before start of a020_0 by [2, 1000000]));
};

activity a003_0_by_a010_1_a013_1_a020_0 {
  duration = 4;
  constraints = ((ends_before start of a010_1 by [8, 1000000]) and
                 (ends_before start of a013_1 by [-1000000, 18]) and
                 (ends_before start of a020_0 by [2, 1000000]));
};

activity a003_0_by_a010_0_a013_0_a020_1 {
  duration = 4;
  constraints = ((ends_before start of a010_0 by [6, 1000000]) and
                 (ends_before start of a013_0 by [-1000000, 16]) and
                 (ends_before start of a020_1 by [8, 1000000]));
};

activity a003_0_by_a010_0_a013_1_a020_1 {
  duration = 4;
  constraints = ((ends_before start of a010_0 by [6, 1000000]) and
                 (ends_before start of a013_1 by [-1000000, 18]) and
                 (ends_before start of a020_1 by [8, 1000000]));
};

activity a003_0_by_a010_1_a013_0_a020_1 {
  duration = 4;
  constraints = ((ends_before start of a010_1 by [8, 1000000]) and
                 (ends_before start of a013_0 by [-1000000, 16]) and
                 (ends_before start of a020_1 by [8, 1000000]));
};

activity a003_0_by_a010_1_a013_1_a020_1 {
  duration = 4;
  constraints = ((ends_before start of a010_1 by [8, 1000000]) and
                 (ends_before start of a013_1 by [-1000000, 18]) and
                 (ends_before start of a020_1 by [8, 1000000]));
};

activity a003_1 {
  decompositions = a003_1_by_a010_0_a013_0_a020_0 or a003_1_by_a010_0_a013_1_a020_0 or
                   a003_1_by_a010_1_a013_0_a020_0 or a003_1_by_a010_1_a013_1_a020_0 or
                   a003_1_by_a010_0_a013_0_a020_1 or a003_1_by_a010_0_a013_1_a020_1 or
                   a003_1_by_a010_1_a013_0_a020_1 or a003_1_by_a010_1_a013_1_a020_1;
};
```

```
activity a003_1_by_a010_0_a013_0_a020_0 {
  duration = 5;
  constraints = ((ends_before start of a010_0 by [-8, 1000000]) and
                 (ends_before start of a013_0 by [-1000000, 12]) and
                 (ends_before start of a020_0 by [-17, 1000000]));
};

activity a003_1_by_a010_0_a013_1_a020_0 {
  duration = 5;
  constraints = ((ends_before start of a010_0 by [-15, 1000000]) and
                 (ends_before start of a013_1 by [-1000000, 30]) and
                 (ends_before start of a020_0 by [-32, 1000000]));
};

activity a003_1_by_a010_1_a013_0_a020_0 {
  duration = 5;
  constraints = ((ends_before start of a010_1 by [3, 1000000]) and
                 (ends_before start of a013_0 by [-1000000, 12]) and
                 (ends_before start of a020_0 by [-47, 1000000]));
};

activity a003_1_by_a010_1_a013_1_a020_0 {
  duration = 5;
  constraints = ((ends_before start of a010_1 by [3, 1000000]) and
                 (ends_before start of a013_1 by [-1000000, 30]) and
                 (ends_before start of a020_0 by [-62, 1000000]));
};

activity a003_1_by_a010_0_a013_0_a020_1 {
  duration = 5;
  constraints = ((ends_before start of a010_0 by [-22, 1000000]) and
                 (ends_before start of a013_0 by [-1000000, 12]) and
                 (ends_before start of a020_1 by [13, 1000000]));
};

activity a003_1_by_a010_0_a013_1_a020_1 {
  duration = 5;
  constraints = ((ends_before start of a010_0 by [-29, 1000000]) and
                 (ends_before start of a013_1 by [-1000000, 30]) and
                 (ends_before start of a020_1 by [13, 1000000]));
};

activity a003_1_by_a010_1_a013_0_a020_1 {
  duration = 5;
  constraints = ((ends_before start of a010_1 by [3, 1000000]) and
                 (ends_before start of a013_0 by [-1000000, 12]) and
                 (ends_before start of a020_1 by [13, 1000000]));
};

activity a003_1_by_a010_1_a013_1_a020_1 {
  duration = 5;
  constraints = ((ends_before start of a010_1 by [3, 1000000]) and
                 (ends_before start of a013_1 by [-1000000, 30]) and
                 (ends_before start of a020_1 by [13, 1000000]));
};

activity a004 {
```

```
  decompositions = a004_0 or a004_1;
};

activity a004_0 {
  decompositions = a004_0_by_a006_0_a012_0 or a004_0_by_a006_1_a012_0 or
         a004_0_by_a006_0_a012_1 or a004_0_by_a006_1_a012_1;
};

activity a004_0_by_a006_0_a012_0 {
  duration = 9;
  constraints = ((ends_before start of a006_0 by [20, 1000000]) and
                 (ends_before start of a012_0 by [25, 1000000]));
};

activity a004_0_by_a006_1_a012_0 {
  duration = 9;
  constraints = ((ends_before start of a006_1 by [22, 1000000]) and
                 (ends_before start of a012_0 by [25, 1000000]));
};

activity a004_0_by_a006_0_a012_1 {
  duration = 9;
  constraints = ((ends_before start of a006_0 by [20, 1000000]) and
                 (ends_before start of a012_1 by [18, 1000000]));
};

activity a004_0_by_a006_1_a012_1 {
  duration = 9;
  constraints = ((ends_before start of a006_1 by [22, 1000000]) and
                 (ends_before start of a012_1 by [18, 1000000]));
};

activity a004_1 {
  decompositions = a004_1_by_a006_0_a012_0 or a004_1_by_a006_1_a012_0 or
                   a004_1_by_a006_0_a012_1 or a004_1_by_a006_1_a012_1;
};

activity a004_1_by_a006_0_a012_0 {
  duration = 7;
  constraints = ((ends_before start of a006_0 by [18, 1000000]) and
                 (ends_before start of a012_0 by [-18, 1000000]));
};

activity a004_1_by_a006_1_a012_0 {
  duration = 7;
  constraints = ((ends_before start of a006_1 by [6, 1000000]) and
                 (ends_before start of a012_0 by [-31, 1000000]));
};

activity a004_1_by_a006_0_a012_1 {
  duration = 7;
  constraints = ((ends_before start of a006_0 by [18, 1000000]) and
                 (ends_before start of a012_1 by [20, 1000000]));
};

activity a004_1_by_a006_1_a012_1 {
  duration = 7;
```

```
  constraints = ((ends_before start of a006_1 by [6, 1000000]) and
                 (ends_before start of a012_1 by [20, 1000000]));
};

activity a005 {
  decompositions = a005_0;
};

activity a005_0 {
  decompositions = a005_0_by_a014_0 or a005_0_by_a014_1;
};

activity a005_0_by_a014_0 {
  duration = 1;
  constraints = ((ends_before start of a014_0 by [2, 8]));
};

activity a005_0_by_a014_1 {
  duration = 1;
  constraints = ((ends_before start of a014_1 by [1, 7]));
};

activity a006 {
  decompositions =  a006_0 or a006_1;
};

activity a006_0 {
  decompositions = a006_0_by_a017_0 or a006_0_by_a017_1;
};

activity a006_0_by_a017_0 {
  duration = 9;
  constraints = ((ends_before start of a017_0 by [27, 1000000]));
};

activity a006_0_by_a017_1 {
  duration = 9;
  constraints = ((ends_before start of a017_1 by [15, 1000000]));
};

activity a006_1 {
  decompositions = a006_1_by_a017_0 or a006_1_by_a017_1;
};

activity a006_1_by_a017_0 {
  duration = 4;
  constraints = ((ends_before start of a017_0 by [4, 1000000]));
};

activity a006_1_by_a017_1 {
  duration = 4;
  constraints = ((ends_before start of a017_1 by [10, 1000000]));
};

activity a007 {
  decompositions = a007_0 or a007_1 with (duration <- duration);
};
```

```
activity a007_0 {
  decompositions = a007_0_by_a010_0_a013_0 or a007_0_by_a010_0_a013_1 or
                   a007_0_by_a010_1_a013_0 or a007_0_by_a010_1_a013_1;
};

activity a007_0_by_a010_0_a013_0 {
  duration = 3;
  constraints = ((ends_before start of a010_0 by [-1000000, -1]) and
                 (ends_before start of a013_0 by [5, 42]));
};

activity a007_0_by_a010_0_a013_1 {
  duration = 3;
  constraints = ((ends_before start of a010_0 by [-1000000, -1]) and
                 (ends_before start of a013_1 by [8, 31]));
};

activity a007_0_by_a010_1_a013_0 {
  duration = 3;
  constraints = ((ends_before start of a010_1 by [-1000000, -29]) and
                 (ends_before start of a013_0 by [5, 42]));
};

activity a007_0_by_a010_1_a013_1 {
  duration = 3;
  constraints = ((ends_before start of a010_1 by [-1000000, -29]) and
                 (ends_before start of a013_1 by [8, 31]));
};

activity a007_1 {
  decompositions = a007_1_by_a010_0_a013_0 or a007_1_by_a010_0_a013_1 or
         a007_1_by_a010_1_a013_0 or a007_1_by_a010_1_a013_1;
};

activity a007_1_by_a010_0_a013_0 {
  duration = 3;
  constraints = ((ends_before start of a010_0 by [-1000000, -1]) and
                 (ends_before start of a013_0 by [1, 18]));
};

activity a007_1_by_a010_0_a013_1 {
  duration = 3;
  constraints = ((ends_before start of a010_0 by [-1000000, -1]) and
                 (ends_before start of a013_1 by [6, 20]));
};

activity a007_1_by_a010_1_a013_0 {
  duration = 3;
  constraints = ((ends_before start of a010_1 by [-1000000, 7]) and
                 (ends_before start of a013_0 by [1, 18]));
};

activity a007_1_by_a010_1_a013_1 {
  duration = 3;
  constraints = ((ends_before start of a010_1 by [-1000000, 7]) and
                 (ends_before start of a013_1 by [6, 20]));
```

```
};

activity a008 {
  decompositions = a008_0 or a008_1;
};

activity a008_0 {
  decompositions = a008_0_by_a009_0_a016_0 or a008_0_by_a009_0_a016_1 or
                   a008_0_by_a009_1_a016_0 or a008_0_by_a009_1_a016_1;
};

activity a008_0_by_a009_0_a016_0 {
  duration = 10;
  constraints = ((ends_before start of a009_0 by [-1000000, -29]) and
                 (ends_before start of a016_0 by [18, 1000000]));
};

activity a008_0_by_a009_0_a016_1 {
  duration = 10;
  constraints = ((ends_before start of a009_0 by [-1000000, -29]) and
                 (ends_before start of a016_1 by [17, 1000000]));
};

activity a008_0_by_a009_1_a016_0 {
  duration = 10;
  constraints = ((ends_before start of a009_1 by [-1000000, -4]) and
                 (ends_before start of a016_0 by [18, 1000000]));
};

activity a008_0_by_a009_1_a016_1 {
  duration = 10;
  constraints = ((ends_before start of a009_1 by [-1000000, -4]) and
                 (ends_before start of a016_1 by [17, 1000000]));
};

activity a008_1 {
  decompositions = a008_1_by_a009_0_a016_0 or a008_1_by_a009_0_a016_1_none or
         a008_1_by_a009_1_a016_0_none or a008_1_by_a009_1_a016_1_none;
};

activity a008_1_by_a009_0_a016_0 {
  duration = 9;
  constraints = ((ends_before start of a009_0 by [-1000000, -10]) and
                 (ends_before start of a016_0 by [7, 1000000]));
};

activity a008_1_by_a009_0_a016_1 {
  duration = 9;
  constraints = ((ends_before start of a009_0 by [-1000000, -10]) and
                 (ends_before start of a016_1 by [11, 1000000]));
};

activity a008_1_by_a009_1_a016_0 {
  duration = 9;
  constraints = ((ends_before start of a009_1 by [-1000000, -1]) and
                 (ends_before start of a016_0 by [7, 1000000]));
};
```

```
activity a008_1_by_a009_1_a016_1 {
  duration = 9;
  constraints = ((ends_before start of a009_1 by [-1000000, -1]) and
                 (ends_before start of a016_1 by [11, 1000000]));
};

activity a009 {
  decompositions = a009_0 or a009_1;
};

activity a009_0 {
  decompositions = a009_0_by_a012_0_a013_0_a016_0 or a009_0_by_a012_0_a013_0_a016_1 or
                   a009_0_by_a012_0_a013_1_a016_0 or a009_0_by_a012_0_a013_1_a016_1 or
                   a009_0_by_a012_1_a013_0_a016_0 or a009_0_by_a012_1_a013_0_a016_1 or
                   a009_0_by_a012_1_a013_1_a016_0 or a009_0_by_a012_1_a013_1_a016_1;
};

activity a009_0_by_a012_0_a013_0_a016_0 {
  duration = 10;
  constraints = ((ends_before start of a012_0 by [-1000000, -4]) and
                 (ends_before start of a013_0 by [-1000000, -21]) and
                 (ends_before start of a016_0 by [-1000000, 43]));
};

activity a009_0_by_a012_0_a013_0_a016_1 {
  duration = 10;
  constraints = ((ends_before start of a012_0 by [-1000000, -4]) and
                 (ends_before start of a013_0 by [-1000000, -21]) and
                 (ends_before start of a016_1 by [-1000000, 39]));
};

activity a009_0_by_a012_0_a013_1_a016_0 {
  duration = 10;
  constraints = ((ends_before start of a012_0 by [-1000000, -4]) and
                 (ends_before start of a013_1 by [-1000000, -22]) and
                 (ends_before start of a016_0 by [-1000000, 43]));
};

activity a009_0_by_a012_0_a013_1_a016_1 {
  duration = 10;
  constraints = ((ends_before start of a012_0 by [-1000000, -4]) and
                 (ends_before start of a013_1 by [-1000000, -22]) and
                 (ends_before start of a016_1 by [-1000000, 39]));
};

activity a009_0_by_a012_1_a013_0_a016_0 {
  duration = 10;
  constraints = ((ends_before start of a012_1 by [-1000000, -17]) and
                 (ends_before start of a013_0 by [-1000000, -21]) and
                 (ends_before start of a016_0 by [-1000000, 43]));
};

activity a009_0_by_a012_1_a013_0_a016_1 {
  duration = 10;
  constraints = ((ends_before start of a012_1 by [-1000000, -17]) and
                 (ends_before start of a013_0 by [-1000000, -21]) and
```

```
                (ends_before start of a016_1 by [-1000000, 39]));
};

activity a009_0_by_a012_1_a013_1_a016_0 {
  duration = 10;
  constraints = ((ends_before start of a012_1 by [-1000000, -17]) and
                 (ends_before start of a013_1 by [-1000000, -22]) and
                 (ends_before start of a016_0 by [-1000000, 43]));
};

activity a009_0_by_a012_1_a013_1_a016_1 {
  duration = 10;
  constraints = ((ends_before start of a012_1 by [-1000000, -17]) and
                 (ends_before start of a013_1 by [-1000000, -22]) and
                 (ends_before start of a016_1 by [-1000000, 39]));
};

activity a009_1 {
  decompositions = a009_1_by_a012_0_a013_0_a016_0 or a009_1_by_a012_0_a013_0_a016_1 or
                   a009_1_by_a012_0_a013_1_a016_0 or a009_1_by_a012_0_a013_1_a016_1 or
                   a009_1_by_a012_1_a013_0_a016_0 or a009_1_by_a012_1_a013_0_a016_1 or
                   a009_1_by_a012_1_a013_1_a016_0 or a009_1_by_a012_1_a013_1_a016_1;
};

activity a009_1_by_a012_0_a013_0_a016_0 {
  duration = 4;
  constraints = ((ends_before start of a012_0 by [-1000000, -1]) and
                 (ends_before start of a013_0 by [-1000000, -3]) and
                 (ends_before start of a016_0 by [-1000000, 47]));
};

activity a009_1_by_a012_0_a013_0_a016_1 {
  duration = 4;
  constraints = ((ends_before start of a012_0 by [-1000000, -1]) and
                 (ends_before start of a013_0 by [-1000000, -3]) and
                 (ends_before start of a016_1 by [-1000000, 18]));
};

activity a009_1_by_a012_0_a013_1_a016_0 {
  duration = 4;
  constraints = ((ends_before start of a012_0 by [-1000000, -1]) and
                 (ends_before start of a013_1 by [-1000000, -6]) and
                 (ends_before start of a016_0 by [-1000000, 47]));
};

activity a009_1_by_a012_0_a013_1_a016_1 {
  duration = 4;
  constraints = ((ends_before start of a012_0 by [-1000000, -1]) and
                 (ends_before start of a013_1 by [-1000000, -6]) and
                 (ends_before start of a016_1 by [-1000000, 18]));
};

activity a009_1_by_a012_1_a013_0_a016_0 {
  duration = 4;
  constraints = ((ends_before start of a012_1 by [-1000000, -16]) and
                 (ends_before start of a013_0 by [-1000000, -3]) and
                 (ends_before start of a016_0 by [-1000000, 47]));
```

```
};

activity a009_1_by_a012_1_a013_0_a016_1 {
  duration = 4;
  constraints = ((ends_before start of a012_1 by [-1000000, -16]) and
                 (ends_before start of a013_0 by [-1000000, -3]) and
                 (ends_before start of a016_1 by [-1000000, 18]));
};

activity a009_1_by_a012_1_a013_1_a016_0 {
  duration = 4;
  constraints = ((ends_before start of a012_1 by [-1000000, -16]) and
                 (ends_before start of a013_1 by [-1000000, -6]) and
                 (ends_before start of a016_0 by [-1000000, 47]));
};

activity a009_1_by_a012_1_a013_1_a016_1 {
  duration = 4;
  constraints = ((ends_before start of a012_1 by [-1000000, -16]) and
                 (ends_before start of a013_1 by [-1000000, -6]) and
                 (ends_before start of a016_1 by [-1000000, 18]));
};

activity a010 {
  decompositions = a010_0 or a010_1;
};

activity a010_0 {
  duration = 2;
};

activity a010_1 {
  duration = 10;
};

activity a011 {
  decompositions = a011_0 or a011_1;
};

activity a011_0 {
  decompositions = a011_0_by_a012_0_a018_0 or a011_0_by_a012_0_a018_1 or
                   a011_0_by_a012_1_a018_0 or a011_0_by_a012_1_a018_1;
};

activity a011_0_by_a012_0_a018_0 {
  duration = 4;
  constraints = ((ends_before start of a012_0 by [-1000000, -1]) and
                 (ends_before start of a018_0 by [9, 1000000]));
};

activity a011_0_by_a012_0_a018_1 {
  duration = 4;
  constraints = ((ends_before start of a012_0 by [-1000000, -1]) and
                 (ends_before start of a018_1 by [6, 1000000]));
};

activity a011_0_by_a012_1_a018_0 {
```

```
  duration = 4;
  constraints = ((ends_before start of a012_1 by [-1000000, -7]) and
                 (ends_before start of a018_0 by [9, 1000000]));
};

activity a011_0_by_a012_1_a018_1 {
  duration = 4;
  constraints = ((ends_before start of a012_1 by [-1000000, -7]) and
                 (ends_before start of a018_1 by [6, 1000000]));
};


activity a011_1 {
  decompositions = a011_1_by_a012_0_a018_0 or a011_1_by_a012_0_a018_1 or
                   a011_1_by_a012_1_a018_0 or a011_1_by_a012_1_a018_1;
};

activity a011_1_by_a012_0_a018_0 {
  duration = 10;
  constraints = ((ends_before start of a012_0 by [-1000000, 3]) and
                 (ends_before start of a018_0 by [-24, 1000000]));
};

activity a011_1_by_a012_0_a018_1 {
  duration = 10;
  constraints = ((ends_before start of a012_0 by [-1000000, 3]) and
                 (ends_before start of a018_1 by [25, 1000000]));
};

activity a011_1_by_a012_1_a018_0 {
  duration = 10;
  constraints = ((ends_before start of a012_1 by [-1000000, -7]) and
                 (ends_before start of a018_0 by [-39, 1000000]));
};

activity a011_1_by_a012_1_a018_1 {
  duration = 10;
  constraints = ((ends_before start of a012_1 by [-1000000, -7]) and
                 (ends_before start of a018_1 by [25, 1000000]));
};

activity a012 {
  decompositions = a012_0 or a012_1;
};

activity a012_0 {
  decompositions = a012_0_by_a019_0 or a012_0_by_a019_1;
};

activity a012_0_by_a019_0 {
  duration = 6;
  constraints = ((ends_before start of a019_0 by [-1000000, 16]));
};

activity a012_0_by_a019_1 {
  duration = 6;
  constraints = ((ends_before start of a019_1 by [-1000000, 17]));
```

```
};

activity a012_1 {
  decompositions = a012_1_by_a019_0 or a012_1_by_a019_1;
};

activity a012_1_by_a019_0 {
  duration = 6;
  constraints = ((ends_before start of a019_0 by [-1000000, 27]));
};

activity a012_1_by_a019_1 {
  duration = 6;
  constraints = ((ends_before start of a019_1 by [-1000000, 25]));
};

activity a013 {
  decompositions = a013_0 or a013_1;
};

activity a013_0 {
  duration = 7;
};

activity a013_1 {
  duration = 10;
};

activity a014 {
  decompositions = a014_0 or a014_1;
};

activity a014_0 {
  duration = 4;
};

activity a014_1 {
  duration = 4;
};

activity a015 {
  decompositions = a015_0 or a015_1;
};

activity a015_0 {
  duration = 8;
};

activity a015_1 {
  duration = 1;
};

activity a016 {
  decompositions = a016_0 or a016_1;
};

activity a016_0 {
```

```
  duration = 7;
};

activity a016_1 {
  duration = 2;
};

activity a017 {
  decompositions = a017_0 or a017_1;
};

activity a017_0 {
  duration = 9;
};

activity a017_1 {
  duration = 10;
};

activity a018 {
  decompositions = a018_0 or a018_1;
};

activity a018_0 {
  duration = 5;
};

activity a018_1 {
  duration = 2;
};

activity a019 {
  decompositions = a019_0 or a019_1;
};

activity a019_0 {
  duration = 5;
};

activity a019_1 {
  duration = 6;
};

activity a020 {
  decompositions = a020_0 or a020_1;
};

activity a020_0 {
  duration = 10;
};

activity a020_1 {
  duration = 8;
};
```

A.2 Sample C_TAEMS Problem

```
;;Scenario time bounds
(spec_boh 20)
(spec_eoh 115)

;;Agents
(spec_agent (label Miek))
(spec_agent (label Marc))
(spec_agent (label Christie))
(spec_agent (label Hector))
(spec_agent (label Moses))
(spec_agent (label Jesper))
(spec_agent (label Blayne))
(spec_agent (label Amy))
(spec_agent (label Les))
(spec_agent (label Kevan))
(spec_agent (label Starbuck))
(spec_agent (label Tandy))
(spec_agent (label Betty))
(spec_agent (label Ole))
(spec_agent (label Liyuan))
(spec_agent (label Naim))
(spec_agent (label Galen))
(spec_agent (label Sandy))
(spec_agent (label Kyu))
(spec_agent (label Cris))

;;Activities
(spec_task_group (label scenario-1)
   (subtasks problem-1 problem-2)
   (qaf q_sum))

(spec_task (label problem-1)
   (subtasks guttural-winroot songbird-winroot hunk-winroot)
   (qaf q_sum_and))

(spec_task (label guttural-winroot)
   (earliest_start_time 30) (deadline 61)
   (subtasks averment exudate healthcare)
   (qaf q_sum_and))

(spec_task (label averment)
   (subtasks solve solve-ff1 solve-rd1)
   (qaf q_max))

(spec_method (label solve) (agent Betty)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 11.75 1.0)
         (duration_distribution 9 0.25 4 0.25 7 0.50))))

(spec_method (label solve-ff1) (agent Betty)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 10.00 1.0)
```

```
         (duration_distribution 4 0.25 6 0.50 7 0.25))))

(spec_method (label solve-rd1) (agent Moses)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 7.75 1.0)
         (duration_distribution 9 0.25 4 0.25 7 0.50))))

(spec_task (label exudate)
   (subtasks snarf snarf-ff1 snarf-rd1)
   (qaf q_max))

(spec_method (label snarf) (agent Moses)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 11.75 1.0)
         (duration_distribution 11 0.50 14 0.25 7 0.25))))

(spec_method (label snarf-ff1) (agent Moses)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 7.75 1.0)
         (duration_distribution 4 0.25 5 0.50 6 0.25))))

(spec_method (label snarf-rd1) (agent Hector)
   (outcomes
      (default (density 0.800000)
         (quality_distribution 10.00 1.0)
         (duration_distribution 3 0.25 5 0.50 6 0.25))
      (failure (density 0.200000)
         (quality_distribution 0.00 1.0)
         (duration_distribution 3 0.25 5 0.50 6 0.25))))

(spec_task (label healthcare)
   (subtasks mousetrap mousetrap-rd1)
   (qaf q_max))

(spec_method (label mousetrap) (agent Hector)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 11.75 1.0)
         (duration_distribution 9 0.50 11 0.25 6 0.25))))

(spec_method (label mousetrap-rd1) (agent Jesper)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 7.75 1.0)
         (duration_distribution 4 0.25 6 0.50 7 0.25))))

(spec_task (label songbird-winroot)
   (earliest_start_time 45) (deadline 55)
   (subtasks owngoal tandoori)
   (qaf q_sum))

(spec_task (label owngoal)
   (subtasks scorn scorn-ff1 scorn-rd1)
   (qaf q_max))
```

```
(spec_method (label scorn) (agent Les)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 10.75 1.0)
         (duration_distribution 9 0.25 4 0.25 7 0.50))))

(spec_method (label scorn-ff1) (agent Les)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 10.00 1.0)
         (duration_distribution 8 0.25 5 0.25 7 0.50))))

(spec_method (label scorn-rd1) (agent Cris)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 7.75 1.0)
         (duration_distribution 8 0.50 10 0.25 5 0.25))))

(spec_task (label tandoori)
   (subtasks unstop unstop-rd1)
   (qaf q_max))

(spec_method (label unstop) (agent Marc)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 11.75 1.0)
         (duration_distribution 9 0.50 11 0.25 6 0.25))))

(spec_method (label unstop-rd1) (agent Les)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 10.00 1.0)
         (duration_distribution 8 0.50 10 0.25 5 0.25))))

(spec_task (label hunk-winroot)
   (earliest_start_time 50) (deadline 74)
   (subtasks barmecide glacis cherry)
   (qaf q_sum_and))

(spec_task (label barmecide)
   (subtasks gutter gutter-ff1 gutter-rd1)
   (qaf q_max))

(spec_method (label gutter) (agent Kevan)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 10.75 1.0)
         (duration_distribution 3 0.25 5 0.50 6 0.25))))

(spec_method (label gutter-ff1) (agent Kevan)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 10.00 1.0)
         (duration_distribution 4 0.25 5 0.50 6 0.25))))

(spec_method (label gutter-rd1) (agent Liyuan)
```

```
   (outcomes
      (default (density 1.000000)
         (quality_distribution 8.75 1.0)
         (duration_distribution 9 0.50 11 0.25 6 0.25))))

(spec_task (label glacis)
   (subtasks purchase purchase-ff1 purchase-rd1)
   (qaf q_max))

(spec_method (label purchase) (agent Liyuan)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 11.75 1.0)
         (duration_distribution 4 0.25 6 0.50 7 0.25))))

(spec_method (label purchase-ff1) (agent Liyuan)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 10.00 1.0)
         (duration_distribution 4 0.25 6 0.50 7 0.25))))

(spec_method (label purchase-rd1) (agent Kevan)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 7.75 1.0)
         (duration_distribution 9 0.50 11 0.25 6 0.25))))

(spec_task (label cherry)
   (subtasks misreport misreport-ff1)
   (qaf q_max))

(spec_method (label misreport) (agent Kevan)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 10.75 1.0)
         (duration_distribution 10 0.50 13 0.25 7 0.25))))

(spec_method (label misreport-ff1) (agent Kevan)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 7.75 1.0)
         (duration_distribution 9 0.50 10 0.25 7 0.25))))

(spec_task
   (label problem-2)
   (subtasks elevon-winroot pingo-winroot extent-winroot)
   (qaf q_sum_and))

(spec_task (label elevon-winroot)
   (earliest_start_time 72) (deadline 84)
   (subtasks racerunner washup reticle)
   (qaf q_sum))

(spec_task  (label racerunner)
   (subtasks fissure fissure-ff1 fissure-rd1)
   (qaf q_max))
```

```
(spec_method (label fissure) (agent Ole)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 10.00 1.0)
         (duration_distribution 3 0.25 5 0.50 6 0.25))))

(spec_method (label fissure-ff1) (agent Ole)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 7.75 1.0)
         (duration_distribution 3 0.25 4 0.75))))

(spec_method  (label fissure-rd1) (agent Naim)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 10.00 1.0)
         (duration_distribution 4 0.25 6 0.50 7 0.25))))

(spec_task (label washup)
   (subtasks tie tie-rd1)
   (qaf q_max))

(spec_method (label tie) (agent Kyu)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 10.75 1.0)
         (duration_distribution 10 0.50 13 0.25 7 0.25))))

(spec_method (label tie-rd1) (agent Tandy)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 10.00 1.0)
         (duration_distribution 3 0.25 5 0.50 6 0.25))))

(spec_task (label reticle)
   (subtasks stem stem-ff1 stem-rd1)
   (qaf q_max))

(spec_method (label stem) (agent Tandy)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 10.00 1.0)
         (duration_distribution 9 0.25 4 0.25 7 0.50))))

(spec_method (label stem-ff1) (agent Tandy)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 10.00 1.0)
         (duration_distribution 4 0.25 5 0.50 6 0.25))))

(spec_method (label stem-rd1) (agent Betty)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 8.75 1.0)
         (duration_distribution 9 0.25 4 0.25 7 0.50))))

(spec_task (label pingo-winroot)
```

```
   (earliest_start_time 78) (deadline 90)
   (subtasks bromicacid aquanaut mandinka)
   (qaf q_sum))

(spec_task (label bromicacid)
   (subtasks randomize randomize-ff1)
   (qaf q_max))

(spec_method (label randomize) (agent Naim)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 11.75 1.0)
         (duration_distribution 10 0.50 13 0.25 7 0.25))))

(spec_method (label randomize-ff1) (agent Naim)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 7.75 1.0)
         (duration_distribution 8 0.50 9 0.25 6 0.25))))

(spec_task  (label aquanaut)
   (subtasks enervate enervate-ff1)
   (qaf q_max))

(spec_method (label enervate) (agent Ole)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 11.75 1.0)
         (duration_distribution 9 0.25 4 0.25 7 0.50))))

(spec_method (label enervate-ff1) (agent Ole)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 7.75 1.0)
         (duration_distribution 4 0.25 6 0.50 7 0.25))))

(spec_task (label mandinka)
   (subtasks entrust entrust-ff1)
   (qaf q_max))

(spec_method (label entrust) (agent Kyu)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 11.75 1.0)
         (duration_distribution 8 0.50 10 0.25 5 0.25))))

(spec_method (label entrust-ff1) (agent Kyu)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 8.75 1.0)
         (duration_distribution 4 0.25 5 0.50 6 0.25))))

(spec_task (label extent-winroot)
   (earliest_start_time 84) (deadline 110)
   (subtasks squashbug-syncpoint civilwar-syncpoint cellarage-syncpoint)
   (qaf q_sum_and))
```

```
(spec_task (label squashbug-syncpoint)
   (subtasks cloudscape salesman plainsman angina)
   (qaf q_sync_sum))

(spec_task (label cloudscape)
   (subtasks orient orient-rd1)
   (qaf q_max))

(spec_method (label orient) (agent Miek)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 10.75 1.0)
         (duration_distribution 4 0.25 6 0.50 7 0.25))))

(spec_method (label orient-rd1) (agent Cris)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 7.75 1.0)
         (duration_distribution 8 0.50 10 0.25 5 0.25))))

(spec_task (label salesman)
   (subtasks reface reface-ff1)
   (qaf q_max))

(spec_method (label reface) (agent Blayne)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 10.00 1.0)
         (duration_distribution 11 0.50 14 0.25 7 0.25))))

(spec_method (label reface-ff1) (agent Blayne)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 10.00 1.0)
         (duration_distribution 9 0.50 10 0.25 7 0.25))))

(spec_task (label plainsman)
   (subtasks diarize diarize-rd1)
   (qaf q_max))

(spec_method (label diarize) (agent Amy)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 11.75 1.0)
         (duration_distribution 9 0.50 11 0.25 6 0.25))))

(spec_method (label diarize-rd1) (agent Blayne)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 10.00 1.0)
         (duration_distribution 9 0.25 4 0.25 7 0.50))))

(spec_task (label angina)
   (subtasks search search-ff1)
   (qaf q_max))

(spec_method (label search) (agent Cris)
```

```
   (outcomes
      (default (density 1.000000)
         (quality_distribution 10.75 1.0)
         (duration_distribution 8 0.50 10 0.25 5 0.25))))

(spec_method (label search-ff1) (agent Cris)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 10.00 1.0)
         (duration_distribution 8 0.25 5 0.25 7 0.50))))

(spec_task (label civilwar-syncpoint)
   (subtasks gyrocopter shoreline putoption denverboot)
   (qaf q_sum))

(spec_task (label gyrocopter)
   (subtasks intoxicate intoxicate-ff1 intoxicate-rd1)
   (qaf q_max))

(spec_method (label intoxicate) (agent Les)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 11.75 1.0)
         (duration_distribution 9 0.50 11 0.25 6 0.25))))

(spec_method (label intoxicate-ff1) (agent Les)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 10.00 1.0)
         (duration_distribution 3 0.25 4 0.75))))

(spec_method (label intoxicate-rd1) (agent Cris)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 7.75 1.0)
         (duration_distribution 8 0.50 10 0.25 5 0.25))))

(spec_task (label shoreline)
   (subtasks enwreathe enwreathe-rd1)
   (qaf q_max))

(spec_method (label enwreathe) (agent Cris)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 10.75 1.0)
         (duration_distribution 10 0.50 13 0.25 7 0.25))))

(spec_method (label enwreathe-rd1) (agent Miek)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 7.75 1.0)
         (duration_distribution 4 0.25 6 0.50 7 0.25))))

(spec_task (label putoption)
   (subtasks configure configure-ff1 configure-rd1)
   (qaf q_max))
```

```
(spec_method (label configure) (agent Marc)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 11.75 1.0)
         (duration_distribution 4 0.25 6 0.50 7 0.25))))

(spec_method (label configure-ff1) (agent Marc)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 7.75 1.0)
         (duration_distribution 4 0.25 5 0.50 6 0.25))))

(spec_method (label configure-rd1) (agent Les)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 10.00 1.0)
         (duration_distribution 3 0.25 5 0.50 6 0.25))))

(spec_task (label denverboot)
   (subtasks skitter skitter-ff1)
   (qaf q_max))

(spec_method (label skitter) (agent Starbuck)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 10.00 1.0)
         (duration_distribution 9 0.25 4 0.25 7 0.50))))

(spec_method (label skitter-ff1) (agent Starbuck)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 10.00 1.0)
         (duration_distribution 3 0.25 4 0.75))))

(spec_task (label cellarage-syncpoint)
   (subtasks bonfire mojarra skimmilk flapjack)
   (qaf q_sum))

(spec_task (label bonfire)
   (subtasks preprogram preprogram-ff1 preprogram-rd1)
   (qaf q_max))

(spec_method (label preprogram) (agent Galen)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 11.75 1.0)
         (duration_distribution 3 0.25 5 0.50 6 0.25))))

(spec_method (label preprogram-ff1) (agent Galen)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 8.75 1.0)
         (duration_distribution 9 0.50 10 0.25 7 0.25))))

(spec_method (label preprogram-rd1) (agent Starbuck)
   (outcomes
      (default (density 1.000000)
```

```
         (quality_distribution 10.00 1.0)
         (duration_distribution 4 0.25 6 0.50 7 0.25))))

(spec_task (label mojarra)
   (subtasks barf barf-ff1)
   (qaf q_max))

(spec_method (label barf) (agent Starbuck)
   (outcomes (default
         (density 1.000000)
         (quality_distribution 10.75 1.0)
         (duration_distribution 3 0.25 5 0.50 6 0.25))))

(spec_method (label barf-ff1) (agent Starbuck)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 7.75 1.0)
         (duration_distribution 8 0.25 5 0.25 7 0.50))))

(spec_task (label skimmilk)
   (subtasks conceive conceive-ff1 conceive-rd1)
   (qaf q_max))

(spec_method (label conceive) (agent Marc)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 10.00 1.0)
         (duration_distribution 11 0.50 14 0.25 7 0.25))))

(spec_method (label conceive-ff1) (agent Marc)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 8.75 1.0)
         (duration_distribution 4 0.25 6 0.50 7 0.25))))

(spec_method (label conceive-rd1) (agent Les)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 8.75 1.0)
         (duration_distribution 4 0.25 6 0.50 7 0.25))))

(spec_task (label flapjack)
   (subtasks import import-rd1)
   (qaf q_max))

(spec_method  (label import) (agent Les)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 10.00 1.0)
         (duration_distribution 3 0.25 5 0.50 6 0.25))))

(spec_method (label import-rd1) (agent Marc)
   (outcomes
      (default (density 1.000000)
         (quality_distribution 10.00 1.0)
         (duration_distribution 4 0.25 6 0.50 7 0.25))))
```

```
;;NLEs
(spec_hinders (label salesman_to_fissure-ff1)
   (from salesman) (to fissure-ff1) (delay 1)
   (quality_power 0.50) (duration_power 0.50))

(spec_enables (label guttural-winroot_to_tandoori)
   (from guttural-winroot) (to tandoori) (delay 1))

(spec_enables (label gutter-rd1_to_famous)
   (from gutter-rd1) (to famous) (delay 1))

(spec_enables (label scorn-rd1_to_gutter-rd1)
   (from scorn-rd1) (to gutter-rd1) (delay 1))

(spec_disables (label racerunner_to_randomize)
   (from racerunner) (to randomize) (delay 1))

(spec_facilitates (label barmecide_to_exudate)
   (from barmecide) (to exudate) (delay 1)
   (quality_power 0.50) (duration_power 0.50))

(spec_facilitates (label enwreathe-rd1_to_elevon-winroot)
   (from enwreathe-rd1) (to elevon-winroot) (delay 1)
   (quality_power 0.50) (duration_power 0.50))

(spec_enables (label elevon-winroot_to_entrust)
   (from elevon-winroot) (to entrust) (delay 1))

(spec_enables (label tandoori_to_barmecide)
   (from tandoori) (to barmecide) (delay 1))

(spec_facilitates (label guttural-winroot_to_purchase-ff1)
   (from guttural-winroot) (to purchase-ff1) (delay 1)
   (quality_power 0.50) (duration_power 0.50))

(spec_hinders (label enervate_to_diarize)
   (from enervate) (to diarize) (delay 1)
   (quality_power 0.50) (duration_power 0.50))

(spec_enables (label stem-rd1_to_enervate)
   (from stem-rd1) (to enervate) (delay 1))

;;Initial Schedule
(spec_schedule
 (schedule_elements
  (mousetrap (start_time 30) (spec_attributes (performer "Hector") (duration 9)))
  (snarf (start_time 30) (spec_attributes (performer "Moses") (duration 11)))
  (solve (start_time 30) (spec_attributes (performer "Betty") (duration 7)))
  (unstop (start_time 45) (spec_attributes (performer "Marc") (duration 9)))
  (scorn (start_time 45) (spec_attributes (performer "Les") (duration 7)))
  (scorn-rd1 (start_time 45) (spec_attributes (performer "Cris") (duration 8)))
  (misreport (start_time 50) (spec_attributes (performer "Kevan") (duration 10)))
  (gutter-rd1 (start_time 55) (spec_attributes (performer "Liyuan") (duration 9)))
  (gutter (start_time 60) (spec_attributes (performer "Kevan") (duration 5)))
  (purchase (start_time 64) (spec_attributes (performer "Liyuan") (duration 6)))
  (misreport-ff1 (start_time 65) (spec_attributes (performer "Kevan") (duration 9)))
  (stem-ff1 (start_time 72) (spec_attributes (performer "Tandy") (duration 5)))
```

```
 (stem-rd1 (start_time 72) (spec_attributes (performer "Betty") (duration 7)))
 (fissure-rd1 (start_time 72) (spec_attributes (performer "Naim") (duration 6)))
 (tie (start_time 72) (spec_attributes (performer "Kyu") (duration 10)))
 (randomize (start_time 78) (spec_attributes (performer "Naim") (duration 10)))
 (enervate (start_time 80) (spec_attributes (performer "Ole") (duration 7)))
 (entrust (start_time 82) (spec_attributes (performer "Kyu") (duration 8)))
 (enwreathe-rd1 (start_time 84) (spec_attributes (performer "Miek") (duration 6)))
 (import-rd1 (start_time 84) (spec_attributes (performer "Marc") (duration 6)))
 (intoxicate (start_time 84) (spec_attributes (performer "Les") (duration 9)))
 (barf (start_time 84) (spec_attributes (performer "Starbuck") (duration 5)))
 (preprogram (start_time 84) (spec_attributes (performer "Galen") (duration 5)))
 (preprogram-rd1 (start_time 89) (spec_attributes (performer "Starbuck") (duration 6)))
 (orient (start_time 90) (spec_attributes (performer "Miek") (duration 6)))
 (conceive (start_time 90) (spec_attributes (performer "Marc") (duration 11)))
 (reface-ff1 (start_time 90) (spec_attributes (performer "Blayne") (duration 9)))
 (diarize (start_time 90) (spec_attributes (performer "Amy") (duration 6)))
 (search (start_time 90) (spec_attributes (performer "Cris") (duration 8)))
 (skitter-ff1 (start_time 95) (spec_attributes (performer "Starbuck") (duration 4)))
 (configure (start_time 101) (spec_attributes (performer "Marc") (duration 6))))
(spec_attributes (Quality 421.35208)))
```