Proactive Replanning for Multi-Robot Teams

Brennan Sellner

CMU-RI-TR-09-01

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

The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

January 2009

Thesis Committee:
Reid Simmons, Chair
Sanjiv Singh
Stephen Smith
Tara Estlin (Jet Propulsion Laboratory, California Institute of Technology)

ABSTRACT

Humans are adept at cooperating and coordinating with one another in dynamic, uncertain environments while adapting to unexpected events and delays. Human workers are able to predict the likely outcome of their current tasks, communicate with one another to schedule and reschedule cooperative tasks, and fluidly join and leave teams as the situation warrants.

This thesis investigates how to bring the fluidity, responsiveness, and reliability of human teams to a centrally coordinated, multi-robot team operating in domains requiring tight inter-robot coordination and either the maximization of overall reward or the minimization of makespan. *Proactive replanning* endeavors to create flexible teams, then exploit their adaptability by dynamically adjusting the teams' composition in response to predicted problems and opportunities. We have explored three components of proactive replanning: live duration prediction, mutable teams, and live task modification.

We present algorithms to predict a distribution across the remaining duration of a task, given a measurement of the task's current state and a relatively sparse set of training data. By predicting the duration of tasks throughout their execution, we are able to execute schedules that achieve 45% of the improvement possible with an omniscient planner.

Mutable teams are those that agents may join or leave throughout the cooperative task's execution. Mutable teams leverage the possibilities presented by optional roles: roles in a team that produce salutary effects, but are not strictly required for the task to proceed. By dramatically increasing the options available to the planner, the use of mutable teams allows the creation of significantly shorter schedules with 57% fewer planning iterations.

Live task modification is the adjustment of tasks already underway by the planner, requiring close coordination between planner and executive. It is informed by duration prediction and uses mutable teams to adjust executing tasks in response to predicted problems and opportunities.

We evaluate the effects of the three components of proactive replanning in isolation and in concert within a complex, stochastic, multi-agent simulated domain. We find that proactive replanning significantly reduces the makespan of executed schedules and increases the flexibility of the planner, bringing multi-robot systems one step closer to the fluid teamwork so effortlessly accomplished by humans.

ACKNOWLEDGMENTS

Thanks to my advisor, Reid Simmons, for more gallons of red ink and hours of critical discussion than I care to think about. I'm not dead, so I must be stronger! Thanks to Sanjiv Singh for years of discussions and project meetings. Thanks to Steve Smith for his useful advice, and to Tara Estlin for her support, as well as her willingness to answer questions about all things CASPER.

Thanks to the DiRA/SSP/Trestle/IDSR crew for years of interesting, frustrating, and rewarding multi-robot adventures: Greg Armstrong, Jon Brookshire, Brad Hamner, Fred Heger, Dave Hershberger, Laura Hiatt, Seth Koterba, Nik Melchior, Josue Ramos, and Trey Smith. Syndicate makes sense, trust me!

Thanks to the entire ASPEN/CASPER development team, without whom I would have had to reinvent the wheel, axle, and internal combustion engine. Special thanks to Steve Schaffer and Caroline Chouinard for answering 1,001 questions about the nuts and bolts of ASPEN and CASPER.

Thanks to Aaron Steinfeld, without whom JMP would just be another aerobics exercise and my statistical analysis would be half-baked.

Thanks to Jeff Schneider for fruitful discussions about function approximation and duration prediction.

Thanks to Suzanne Lyons Muth, Jean Harpley, Kristen Schrauder, and Karen Widmaier for better administrative support than anyone could imagine.

Thanks to many friends and colleagues for their support over the years, especially Frank Broz, Justin Carlson, Rachel Gockley, Dani Goldberg, Jonathan Hurst, Mary Koes, Jeremy Kubica, Brad Lisien, Maxim Makatchev, Matt Mason, Marek Michalowski, Anne Mundell, Illah Nourbakhsh, Mayan Roth, Nick Roy, Paul Rybski, Alan Schultz, Sebastian Thrun, Chris Urmson, and Chuck Whittaker.

Finally, thanks to my family for their enduring support (and a mountain of editing), and to Maria and Remi for putting up with me.

DEDICATION

For my brother, Andrew.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Humans are remarkably adept at operating in, and adapting to, dynamic, uncertain environments, setting an extremely high standard for robots operating in the real world. One aspect of human adaptability is our ability to predict the outcome of events and to adjust our plans quickly to accommodate any unexpected changes. When working in teams, humans will provide their teammates with frequent progress updates and estimates about whether they expect to finish their assigned tasks on time. This information flow allows individuals to adjust their schedules to make the best use of their time. For instance, the foreknowledge that a group meeting will be delayed by an hour because the team leader is caught in traffic allows everyone to take on appropriate tasks during the now-free window and reschedule anything that may conflict with the delayed meeting.

Human teams also are extremely flexible: members are able to join or leave teams at will, as their schedules or the tasks allow. A team task is divided as appropriate among the workers, according to their skills and availability. For example, suppose a pair of construction workers are shingling a roof, when one drops his hammer. If no one else is available, no additional shingles can be placed until one of them descends to the ground, retrieves the hammer, and returns to the roof. If instead there is a team nearby mixing cement, one of them may briefly leave the team to throw the hammer back up to the roofing team, significantly shortening the shingling task with minimal impact on the cement team. Human adaptability allows us to replan our work processes, redistribute tasks as needed, and reap the benefits of increased efficiency.

In this thesis, we are motivated by bringing these capabilities of prediction, flexibility, and adaptability to a multi-robot team operating under the control of a central scheduler. Our simulated experimental domains model two related scenarios, in which a set of tasks must be performed to establish and maintain a lunar

outpost. Some tasks require transitioning between discrete locations, while others remain stationary. Some require only a single agent, others require multiple agents, and most may make use of additional agents if they are available, in order to increase the task's speed or reliability. In one scenario, the objective is to maximize reward within a fixed scheduling window, while the goal in the other scenario is to minimize the length of a schedule containing a fixed set of tasks. These characteristics result in scenarios similar to many human tasks, where agents may be added or removed from tasks as conditions warrant.

Our approach, termed *proactive replanning*, increases the system's efficiency, flexibility, and reliability by predicting deviations from the schedule and providing additional methods to adjust the schedule to deal with unexpected contingencies and opportunities. Current planning and execution systems are limited in their options when execution does not proceed as expected because they often lack the ability to predict problems in the future, cannot schedule agents to participate in only a portion of a task, and are unable to modify a task once it has begun execution.

In this thesis, we explore three aspects of proactive replanning: *(live) duration prediction*, *mutable teams*, and *live task modification*. *Duration prediction* is the prediction of a task's duration, given its state and assigned team. *Live duration prediction* consists of generating such predictions throughout execution. *Mutable teams* are those that allow agents to join or depart a team in mid-task, and include *optional roles*: roles that improve the performance of the team in some fashion, and may be filled if sufficient agents are available, but are not essential to the completion of the task. *Live task modification* is the act of the planner changing an executing task in some fashion, which allows the planner to adjust the schedule to directly compensate for events occurring during execution. In this thesis, we specifically examine modification of the team profile, which consists of adjusting which agents are assigned to the task, and for what spans of time. Taken as a whole, these three components enable the proactive replanning system to foresee problems and opportunities, then react in time to resolve or take advantage of them.

1.1 Thesis Statement

Thesis: *A proactive replanner is able to construct and execute more efficient or value-laden schedules by continuously predicting the duration of executing tasks, leveraging the opportunities provided by mutable teams and optional roles, and modifying the composition of teams as execution progresses.*

We support this statement by presenting algorithms to predict the duration of ongoing tasks; to model and reason about teams whose composition varies throughout execution; and to reason about when to make changes to the team profiles. We

have validated our algorithms through a series of experiments in simulation. With live duration prediction alone, the planner was able to execute schedules with 45% of the additional reward possible if perfect information were available. In isolation, mutable teams enable the generation of schedules 5.65% shorter than possible with immutable teams. When the three aspects of proactive replanning are combined, the complete system is able to generate, optimize, and execute schedules 11.5% shorter than the baseline.

1.2 Document Outline

- Chapter 2 discusses related planning systems, planning and execution architectures, and other work on problems similar to duration prediction.
- Chapter 3 provides a background discussion of the ASPEN planner and CASPER execution system that has been utilized, as well as how the ASPEN/CASPER core has been extended to support proactive replanning. An introduction to Kernel Density Estimation also is provided, which is the function approximation technique upon which our duration prediction algorithms have been built.
- Chapter 4 presents an overview of our approach to proactive replanning, and discusses the three components of proactive replanning and the requirements that proactive replanning places upon the underlying systems.
- Chapter 5 details our approach to duration prediction, both prior to and throughout execution. Duration prediction's applicability is discussed, as well as its integration into the planning and execution system. Duration prediction is experimentally evaluated in isolation.
- Chapter 6 discusses the use of mutable teams and optional roles to increase the flexibility of the planner at both plan- and execution-time. The requirements for mutable teams, as well as how best to represent and design for them, are addressed, as is the extension of our duration prediction algorithms to support mutable teams. The effect of mutable teams on the efficiency of initial schedules is evaluated.
- Chapter 7 explains the use of live task modification to allow the planner to react more effectively to events as execution proceeds. It also discusses the demands placed upon the planner and executive by live task modification in detail, as well as how we implement live task modification within the ASPEN/CASPER framework.

- Chapter 8 details how the different components of proactive replanning are integrated into a complete system, building upon ASPEN and CASPER. Examples of our proactive replanning-specific heuristics are discussed, and our complete heuristic suite is summarized. Extensive experiments are presented that evaluate the different combinations of live duration prediction, mutable teams, and live task modification.

- Chapter 9 presents our conclusions, future work, and summarizes the thesis.

The following sections summarize the main results of Chapters 5 - 8.

1.3 Live Duration Prediction (Chapter 5)

The ability to predict the expected duration of tasks both prior to, and throughout, execution lies at the core of proactive replanning. By providing advance warning of the effects of execution-time events, duration prediction allows the planner to react while there is still time to address problems effectively. *Duration prediction* is the prediction of a distribution across possible durations for a task, given the task's initial state and assigned set of agents. *Live duration prediction* is doing so given the task's current state as it executes. By updating its schedule as events occur during execution, the planner is better able to adapt the schedule to the realities of execution. By predicting a distribution, rather than a scalar value, the planner is able to engage in several new strategies (unattainable if we simply compute a scalar duration), such as multi-metric optimization, reasoning about deadlines, the efficient use of mutable teams, and prediction jitter compensation. In contrast, existing planning and execution systems dispatch tasks from the planner to the executive, at which point control is passed entirely to the executive. Minimal updates are provided to the planner during execution, often consisting simply of completion or failure notices when tasks have ceased executing. This paucity of data is insufficient to support a proactive replanning approach.

We used a form of Kernel Density Estimation (Silverman, 1986) to develop an approach to duration prediction that enables the estimation of duration distributions given relatively sparse training data. Kernel Density Estimation is a non-parametric method related to histograms that is used to estimate an arbitrary distribution from training data without making *a priori* assumptions about the form of the underlying distribution. This allows our approach to be used in scenarios where it is not feasible to collect exhaustive training data, while being able to incorporate additional data as it becomes available. We have evaluated the effect of duration prediction in a simulated multi-agent lunar outpost construction scenario, where the objective is to maximize reward within a fixed planning horizon. With sufficient training data,

the planner was able to achieve 45% of the improvement possible if it were omniscient. We also examined the effects of varying the amount of training data, and found that live duration prediction produced significantly improved results with even very sparse data (4 training runs).

1.4 Mutable Teams (Chapter 6)

Mutable teams are teams that agents may join, or leave, during the execution of a task, allowing the planner to schedule agents to participate during only a portion of the task's execution. Such teams often include *optional roles*, which need not be filled for the task to proceed, but provide tangible benefits if sufficient agents are available, such as a higher rate of progress or reduced likelihood of failure. These contrast with *required roles*, which must be filled throughout the task's performance for it to proceed, although a required role need not be filled in its entirety by a single agent. We refer to the set of assigned agents, and their scheduled arrival and departure times, as a *team profile*. To the best of our knowledge, existing scheduling systems exclusively address immutable teams, with a fixed number of agents assigned to a task throughout its execution.

The concept of optional roles alone grants the planner significant additional flexibility during both the formation of the initial schedule and its execution. By providing flexibility in the number of agents that may be assigned to a task, optional roles allow the planner to trade resources against execution time as the situation warrants. Mutable teams enhance the utility of optional roles by allowing agents to be added to a team mid-task as they become available. This allows the planner to increase task efficiency, even if there are insufficient agents available to fill the optional role(s) in their entirety. In addition, mutable teams and optional roles provide the planner with the framework necessary to adjust the allocation of agents as execution-time events warrant: if two teams are executing in parallel, with one running ahead and the other behind, the planner may transfer agents to the slower task, balancing the schedule and reducing the impact of the slow team on the makespan. Performing such an operation requires both mutable teams and live task modification.

Chapter 6 discusses the concepts of mutable teams and optional roles in detail, the requirements placed upon the planner and executive in order to support mutable teams, and several possible representations of mutable teams within the planner. The design of tasks to support optional roles and mutable teams is addressed, as well as the extension of our duration prediction algorithms to estimate task duration for mutable teams, given training data from immutable teams. Experimental results are presented, comparing the efficiency of schedules generated

using mutable or immutable teams. The scenario explored here is a variant on the lunar outpost construction theme, with many tasks, optional roles, and a goal of minimizing the schedule's makespan. The introduction of mutable teams yields schedules shorter by a statistically significant amount (5.65%), with no overall computational penalty. The additional options provided by mutable teams result in 57.5% fewer plan repairs, although each repair is more computationally expensive, due to the complexity of predicting the duration of mutable teams.

1.5 Live Task Modification (Chapter 7)

Live task modification is the act of changing some aspect of an executing task, generally in an attempt to take advantage of an opportunity or alleviate a problem detected during execution. In this thesis, we address the adjustment of the team profile of an executing task by the planner. This formulation of live task modification is predicated upon the presence of mutable teams, although the general concept may be applied to any adjustable parameter of a task, such as its goal, path, or speed. By modifying an executing team's profile, the planner is able to adjust the arrival and departure time of agents, add additional agents to the team, or remove existing assignments. This expands the planner's set of options when resolving conflicts involving executing tasks or optimizing the near-future portion of the schedule. For instance, if a team is running behind, additional agents may be transferred into open optional roles to bring the team back on the original schedule. Similarly, overachieving teams may donate agents to other teams or start future tasks earlier.

In contrast, existing planning and execution systems do not allow the planner to affect tasks once they have begun execution. While this provides a clean break between the authority of the planner and executive, it relegates the planner to a passive role in addressing the effects of execution-time events: it may adjust only the remainder of the schedule, rather than actively attempting to solve the problem at hand.

Chapter 7 discusses the applications of live task modification, the demands it places upon the planner and executive, and our implementation of live task modification within the ASPEN/CASPER framework. We do not independently evaluate live task modification, due to the dependence of our formulation on mutable teams. Instead, we examine its effects in the context of a complete proactive replanning system. The results of these experiments are reported in Chapter 8.

1.6 Proactive Replanning (Chapter 8)

In Chapter 8, we address how duration prediction, mutable teams, and live task modification fit together within the ASPEN/CASPER framework to form a complete proactive replanning system. We discuss their integration into ASPEN/-CASPER, as well as how we extended the ASPEN/CASPER core to more fully support proactive replanning.

ASPEN is a heuristic-based iterative repair planner/scheduler that makes use of a set of repair and optimization methods. Each method consists of a fixed series of steps that may resolve a conflict or optimize some portion of the schedule. For instance, repair methods include moving a task, deleting a task, or transferring an agent between tasks. Each method, in turn, has a set of *choice points*, where decisions are made such as upon which tasks to operate or where to place a task. Domain-specific heuristics are applied at each of these choice points, allowing ASPEN's repair and optimization behavior to be customized to the task at hand. In this chapter, we revisit our additional repair and optimization methods, first introduced in Chapter 3, and survey our suite of heuristics. We discuss in detail the *Transfer* method, in which the planner decides whether, and how, to transfer agents between tasks.

This chapter also presents the metric simulator (TaskSim) developed for this thesis. TaskSim models tasks using an expanded form of Augmented Task Networks (Woods, 1970), allowing their execution to be stochastic and dependent upon an evolving state. The integration of TaskSim with the CASPER execution system is addressed, as well as how the simulator's world model varies in a significant manner from that of the planner.

Extensive experiments also are presented that evaluate the effects of utilizing different combinations of live duration prediction, mutable teams, and live task modification. In these experiments, we utilize the same lunar construction scenario as was used to evaluate mutable teams: there are many multi-agent tasks with optional roles, and an overall goal of minimizing the executed schedule's makespan. We find that mutable teams reduce the makespan by 5%, while adding live task modification yields an additional 6.5%, but only if live duration prediction is also enabled. With all three components active, the proactive replanning system is able to construct, optimize, and execute schedules 11.5% shorter than is otherwise possible.

Finally, the results are presented of experiments exploring the applicability of proactive replanning to a range of domains. We find that as the number of agents and the impact of individual failures increase, proactive replanning provides a correspondingly increased reduction in makespan.

1.7 Summary

This thesis will present three components of a proactive replanning system: (live) duration prediction, mutable teams, and live task modification. We have evaluated duration prediction and mutable teams in isolation, showing their positive effect on the efficiency of multi-agent schedules in two lunar-inspired construction scenarios. In addition, we have performed an extensive experiment evaluating the effects of the three components of proactive replanning in isolation, and in various combinations, providing insight into the utility of each. Overall, we have validated the hypothesis that a proactive replanning system is able to build and execute complex, multi-agent schedules more efficiently and robustly than existing planning-execution systems.

Chapter 2

Related Work

Prior work related to proactive replanning and our approach to it falls into three categories: planning, scheduling, and execution systems; task duration prediction algorithms; and work on representing mutable teams, or modifying them during execution. We will discuss how existing systems address the concepts of proactive replanning, then survey a variety of algorithms related to the prediction of task duration or similar properties, and conclude with a discussion of work related to mutable teams and live task modification.

2.1 Planning, Scheduling, and Execution Systems

There are two major components to consider when evaluating the architecture of a proactive replanning system: the planner/scheduler itself and how it interacts with the lower-level elements of the system. We will discuss existing planners, then cover architectures into which they have been integrated.

2.1.1 Planners / Schedulers

The demands of proactive replanning on the planner are significant, including support for durative actions, temporal constraints, exogenous events, multiple agents (or at least metric resources), and the ability to quickly replan or repair a plan in response to feedback from the executive. Many existing planners are unsuited to such domains.

Classical planning approaches such as those using the STRIPS formulation (Fikes and Nilsson, 1971) are of limited use due to their modeling of actions as instantaneous transitions between states. Since the core of our approach to proactive replanning is to monitor task progress and modify teams during execution,

the planner must model actions as having duration and being interruptible. Some classical planning approaches have incorporated durative actions. For example, the PDDL2.1 Level3 language (Fox and Long, 2002) extends the classical PDDL language to allow the specification of durative actions, preconditions and effects at the start and end of durative actions, and invariant conditions. However, PDDL2.1 maintains the representation of an action as a transition between two states, and time points within the transition cannot be specified. This makes any expression of duration prediction or live task modification impossible, and renders planners such as TLplan (Bacchus and Ady, 2001), TGP (Smith and Weld, 1999), O-Plan (Currie and Tate, 1991), and SHOP2 (Nau et al., 2001) unable to take advantage of the benefits of proactive replanning.

Conformant and conditional planners also are not applicable to domains amenable to proactive replanning. Conformant planners such as CGP, (Smith and Weld, 1998), CMBP (Cimatti and Roveri, 2000), and C-PLAN (Ferraris and Giunchiglia, 2000) produce plans containing only actions that will lead to the goal regardless of the current state. In the complex, durative domains that proactive replanning is most suited for, this is far too restrictive, as such actions almost never exist. Conditional planners such as SGP (Weld et al., 1998), MBP (Bertoli et al., 2001), and GPT (Bonet and Geffner, 2000) attempt to build plans with branches based on the results of sensing actions. Again, the complexity of the domains we are considering becomes an insurmountable issue, as the set of all possible state traces becomes much too large to enumerate.

The design-to-time (Garvey and Lesser, 1995) approach to real-time scheduling endeavors to dynamically build the best schedules possible within a constrained amount of time, preferring those that maximize some measure of quality, while minimizing duration. It operates upon a hierarchical representation of tasks, referred to as TAEMS, where any given task may have a variety of alternatives that have differing qualities and duration. Uncertain durations and qualities are represented as (very) discrete distributions. Execution is periodically stopped and the progress being made by active tasks is evaluated. Alternative tasks may be swapped in at these discrete, pre-selected monitoring points. While design-to-time shares many characteristics with proactive replanning, it is difficult to represent and reason about mutable teams within the TAEMS framework. The potential application of C-TAEMS (an extension of TAEMS) to proactive replanning is evaluated in Section 6.3.3. In addition, the nature of design-to-time's monitoring points is incompatible with proactive replanning's continuous approach to duration prediction and execution: in proactive replanning, we make every effort to ensure that execution proceeds uninterrupted. Finally, when adjusting the scheduling in response to execution-time feedback, a proactive replanner reasons about modifying a task, while design-to-time instead replaces a poorly performing task with a different

method, which does not share any state with its predecessor.

Design-to-criteria (Wagner and Lesser, 1999) is based upon design-to-time, expanding it to include a variety of hard and soft constraints relating to time, resources, task interactions, and inter-agent commitments. The schedules generated by design-to-criteria may contain uncertainty and known potential failure points, with the expectation that rescheduling will occur during execution. Scheduling and rescheduling is guided by a set of criteria, expressing preferences for the overall duration, cost, and/or quality of the final plan, allowing high-level manipulation of the planner's strategy. While design-to-criteria is much more flexible than design-to-time, it shares the same shortcomings with respect to the problems proactive replanning addresses.

Work on schedule revision (also known as reactive scheduling), such as the OPIS (Smith (1988), Smith et al. (1990)) and GERRY (Zweben et al., 1992) systems, focused on repeatedly selecting and applying heuristic-based methods to repair conflicts or optimize a schedule. These approaches are conceptually similar to our approach, in that they are designed to incorporate information from the partial execution of a schedule (such as task completions and machine breakdowns), and revise it minimally, rather than rescheduling from scratch. OPIS is also able to modify executing tasks to a limited extent. For instance, if an agent breaks down, OPIS will divide any active tasks to allow the remainder of the task to be scheduled on a new agent. The primary differences between this work and ours are a matter of the scope of schedule revisions and duration models. Schedule revision generally occurs at task boundaries, while we continually update our duration predictions and proactively modify the schedule in response to changes in a task's ongoing execution. In addition, these schedule revision systems make use of deterministic estimates of duration, while we model a fine-grained discrete distribution, increasing the power and complexity of the reasoning that we are able to perform about task durations.

Alternative approaches describe the world as a set of state variables, each as a function of time. Each variable has an associated timeline, which encodes the variable's past states and predicted future states, given the current plan. A task is an interval on one or more of these timelines, within which the values of the associated variables change. This formulation is much more amenable to proactive replanning, as this representation of tasks lends itself to mid-task analysis. A number of planners use such formulations; we will discuss three: ASPEN (Chien et al., 2000b), EUROPA (Frank and Jónsson, 2003) (the successor to HSTS (Smith (1993), Muscettola (1994)), and IxTeT (Laborie and Ghallab, 1995). All of these planners are capable of handling a range of resource types (e.g. reusable or consumable) through the use of underlying constraint networks and constraint satisfaction techniques.

Given a set of goals, a plan (initially empty), the current state, and the predicted variable timelines, ASPEN performs iterative plan repair to resolve conflicts in the predicted schedule. Repair and optimization steps may be interleaved as the state of the system is updated[1], using a most-commitment strategy (all variables are grounded as early as possible). This strategy makes the evaluation of metrics and projections of resource usage much simpler, but reduces the plan's flexibility. ASPEN is a good fit to proactive replanning, although it lacks explicit multi-agent support, and heuristics aware of the structure of mutable teams and tasks with optional roles had to be developed. We have built our implementation of proactive replanning using ASPEN as a base.

EUROPA is a constraint-based interval planner that casts the planning problem as a dynamic constraint satisfaction problem. It also makes use of timelines, is functionally similar to ASPEN. However, it makes use of flexible time, rather than fixed times: a task in EUROPA is constrained to occur within an interval, rather than scheduled to run at a specific time. This flexibility may provide greater leverage for duration prediction, as the predicted duration distributions may be incorporated more directly into the plan. However, a flexible time approach complicates the use of mutable teams, especially when ensuring that a required role is filled throughout a task.

IxTeT also plans by using timelines, each of which consists of a sequence of temporal assertions which can represent either the persistence of a value over an interval or an instantaneous change of value. IxTeT is based on a partial-order causal link (POCL) planning process with constraint-satisfaction techniques and generates order-constrained plans with unbound variables. These plans are more flexible at execution time than the fully grounded plans produced by ASPEN, but make it significantly more difficult to evaluate metrics and predict conflicts. While IxTeT can perform plan repair through search in the partial plan space, the importance to proactive replanning of metric evaluation and conflict prediction make IxTeT less suited to proactive replanning than ASPEN.

2.1.2 Architectures: Integrating Planning and Execution

Architectures serve to tie together the different elements of an autonomous system into a cohesive whole, and define how these components interact. Such architectures commonly have been divided into two or three primary components: a functional layer that interacts with hardware and executes low-level commands; and a decisional layer that determines what commands should be executed (Alami et al. (1998), Volpe et al. (2000)). The decisional layer often is split into a high-

[1]The mix of repair and optimization steps is determined by the enclosing architecture.

level deliberative planner and a mid-level execution layer responsible for overseeing the functional layer (Bonasso et al. (1997), Gat (1992)). Proactive replanning requires a close, high-frequency connection between the planning and executive layers, ensuring that the planner is kept up to date on execution progress, may continuously replan, and may affect currently executing tasks. Existing architectures "lock" currently executing tasks to prevent the planner from modifying them; this is the primary architecture-related impediment to the implementation of proactive replanning.

Due to the need to rapidly re-evaluate team composition during task execution, the "Sense-Plan-Act" approach exemplified by the PLANEX system used on Shakey (Fikes et al., 1972) is not applicable. Similarly, batch planning such as that used in the Remote Agent experiment (Bernard et al., 2000) (Muscettola et al., 1998) is unable to take advantage of the dynamic nature of proactive replanning domains.

"Reactive planning" techniques, in which the executive is endowed with limited planning capabilities and (if present) the planner is used similarly to batch-planning, are not applicable to proactive replanning, as they do not predict state values sufficiently far into the future. Since one of proactive replanning's benefits is the ability to prevent predicted problems, the long-term projection of state values is a crucial component. Examples of reactive planners include those that simply choose among available tasks and perform some failure recovery, such as RAP (Firby, 1994), ESL (Gat, 1997), and TDL (Simmons and Apfelbaum, 1998) and transformational planners such as XFRM (Beetz and McDermott, 1994). While such techniques would be useful in an executive as part of a proactive replanning system, a longer-term scheduling approach also is required.

Three-layer architectures such as 3T (Bonasso et al., 1997) and ATLANTIS (Gat, 1992) may be adaptable to proactive replanning, but the ties between their planning and executive layers generally are not tight enough to support continuous replanning and live task modification.

A number of architectures, such as IPEM (Ambros-Ingerson and Steel, 1988), ROGUE (Haigh and Veloso, 1998), and SIPE (Wilkins, 1988) have implemented continuous planning, in which planning and execution are seamlessly interleaved. However, IPEM and ROGUE are purely classical planning approaches, while SIPE does not explicitly represent time, rendering them not applicable to proactive replanning.

LAAS (Alami et al., 1998)[2] and CLARAty (Nesnas et al., 2003) (Volpe et al., 2000) both encapsulate planning/scheduling and execution into a single layer. In the case of LAAS, the two are tightly intertwined, while in CLARAty the deci-

[2]IxTeT is the planning component of LAAS, which is named after the lab that developed it.

sional layer still contains distinct planning and execution objects. At the moment, two instances of CLARAty's decisional layer are available: CASPER (Chien et al., 2000a) and CLEaR (Estlin et al., 2001). CASPER uses the ASPEN (Chien et al., 2000b) planner and a simple executive, while CLEaR adds a TDL-based (Simmons and Apfelbaum, 1998) executive to CASPER. Both LAAS and CLARAty appear well-suited to proactive replanning, due to the relatively tight coordination possible between their respective planning and execution components. However, as currently implemented, neither is able to modify a currently executing task, since active tasks are locked to avoid conflicts between planner and executive. In fact, Chien et al. (2000a) states that ensuring the planner does not modify executing tasks is an explicit problem that must be solved in order to perform interleaved plan execution and repair. In order to realize proactive replanning, and specifically live task modification, this restriction on the modification of executing tasks must be relaxed to allow the planner to dynamically change the allocation of agents. Due to concerns about how well IxTeT (LAAS's planner) could be adapted to proactive replanning, we selected CASPER as a foundation for our work.

The final relevant architecture is IDEA (Dias et al., 2003) (Muscettola et al., 2002), which advocates the use of planning as the core of each level of abstraction, from mission planning to reactive execution. Rather than using a deliberative planner and a specialized executive, IDEA instead uses a constraint-based planner for both planning and execution: at the planning level, a long horizon is used to construct longer-term schedules, while the executive ensures that the short-term schedule remains valid. As currently implemented, the two planning horizons are disjoint: the long-term planner may not affect tasks that have begun execution. To support proactive replanning, this segregation would need to be relaxed: the long-term planner must be able to modify executing tasks. IDEA's inter-module communications appear sufficient for the short-term planner to keep the long-term planner abreast of developments. IDEA makes use of the EUROPA planner as its internal planning module.

2.2 Duration Prediction

To our knowledge, no existing planning/execution systems dynamically predict the remaining duration of a task. However, the problem may be cast as the approximation of a function (the duration distribution), given a set of training data. Much applicable research has been performed on various aspects of function approximation. We are interested in predicting a distribution across a continuous metric (remaining duration) given a collection of continuous and discrete state inputs (the training data and current task state), under the Markovian assumption. There are

two elements to this problem: (1) predicting the duration distribution at a specific point in the state space, and (2) generalizing this to allow predictions across the entire space with relatively sparse training data.

The first portion of the problem has been well-studied by the function approximation community. Parametric distributions, such as the gamma and normal, can be fit to arbitrary data using approaches such as maximum likelihood estimation (Kay, 1993). However, parametric distributions make assumptions about the underlying distribution that may not hold, especially when predicting the duration of tasks executed in dynamic, uncertain environments that may have multi-modal underlying distributions.

Nonparametric approaches such as thin-plate splines (Bookstein, 1989) and piecewise linear regression are able to fit arbitrary functions, and in general are sufficient for the first portion of the problem. However, they break down when generalizing across larger numbers of dimensions.

Approaches such as multivariate adaptive regression splines (Friedman, 1991), locally weighted projection regression (Vijayakumar and Schaal, 2000), and neural networks are capable of approximating functions from high-dimensional input spaces. While all could be utilized for duration prediction, when we applied them to our domain, fitting times tended to be long and over-fitting often occurred.

Belker et al. (2003) use model trees (Karalic, 1992) (Quinlan, 1992) to predict the expected duration of a task, given previous observations. This approach provides a piecewise linear regression model of the duration function, and outputs a single estimate of duration. It is unclear how this approach could be extended to predict distributions across task duration.

Meuleau et al. (2004) use kd-trees to condense plateaus in a value function to achieve a more compact representation. This is not directly applicable to our problem, as remaining duration often scales smoothly with the task state variables, rather than forming plateaus, but presents one possible approach to generalizing a large data set.

We have previously (Sellner et al., 2005) modeled the expected duration of a task that may be attempted multiple times by either a human operator or an autonomous system. This work focused on the allocation of the task to either human or autonomous control, and resulted in an estimate of task duration for the two possible assignments of the next attempt at the task. Although the work could be used to integrate humans into a duration prediction algorithm, it is not directly applicable.

We have selected a modified form of kernel density estimation (KDE) (Silverman, 1986) as our prediction method. KDE is a nonparametric method that is able to estimate an arbitrary distribution from training data without making assumptions about the structure of the underlying distribution. It places "kernel" distributions

around each training point, then combines the kernels to form an estimate of the underlying distribution. This allows KDE to model arbitrary distributions, including multi-modal distributions that cannot be captured by most parametric approaches. In addition, KDE is able to form a (very approximate) estimate with minimal training data. KDE is not subject to over-fitting, as it operates directly from the data, and is able to interpolate to a degree between data points. KDE is discussed in detail in Sections 3.3 and 5.4.

2.3 Mutable Teams and Live Task Modification

To the best of our knowledge, neither mutable teams, nor live task modification, have been implemented as, or even considered for, a component of a planning/execution system. At most, existing planners are able to abort currently-executing tasks or specify one of a set of possible teams, with no mid-task changes to the team profile. However, a variety of research topics touch upon different aspects of mutable teams and live task modification.

2.3.1 Multi-robot task allocation

The multi-robot task allocation problem (MRTA) is concerned with finding the optimal assignment of robots to a set of tasks. While mutable teams and live task modification bear some similarity to the problem, they are in fact complementary. Live task modification is concerned with adding or removing agents to, or from, executing multi-agent tasks in response to observations, while MRTA is generally considered to be a static optimization problem (Gerkey and Mataric, 2003). There has been some work on "dynamic" MRTA (Strens and Windelinckx, 2005), but the authors define a dynamic MRTA problem as one in which new tasks arrive at random times, and must be accommodated by the scheduler. Their work still makes the implicit assertion that any task that has begun execution is immune to modification by the planner/scheduler. Mataric et al. (2003) discuss MRTA incorporating the possibility of agents taking an opportunistic strategy in which they may release their current task in favor of a different one. However, all tasks are performed by a single agent, precluding live task modification as formulated here, as there are no teams to modify.

2.3.2 Swarms

Many swarm, or emergent, approaches to multi-agent systems (Correll and Martinoli, 2004) (Mataric, 1992) can be considered to be conducting live task modification with mutable teams, as agents may be freely added to, or removed, from the

team without dramatically affecting performance. However, we are interested in problems in which teams of agents perform highly coordinated tasks, rather than the loosely coupled foraging, inspection, or mapping tasks common to swarming algorithms. Reasoning within a swarm is performed locally on each agent, with little to no explicit coordination between agents, and no central control. This works well for loosely coupled tasks, but we are examining highly coordinated domains where agents must closely coordinate their actions (e.g. multiple robots carrying a large object), and must be able to respect temporal constraints between tasks. The decentralized, minimally coordinated approach taken by swarms does not translate well into our domains of interest.

2.3.3 Role Exchange

The MOVER architecture (Jennings and Kirkwood-Watts, 1998) provides the ability to procedurally add, remove, or substitute agents on a team. However, the conditions for such team changes must be prespecified in a policy for each task. There is no planner using this capability to improve the overall performance of the scenario; instead, teams are reactively modified based on local information and policy. This is a limited form of mutable teams, although its lack of planner integration falls short of proactive replanning.

The work of Stone and Veloso (1998) supports the periodic exchange of roles between members of a team, but only addresses domains in which there is a single team. In addition, there is no tight inter-agent coordination in their domain (robotic soccer). As a result, agents may freely move between roles with no need for explicit addition or removal tasks. A significant body of work exists that addresses the dynamic assignment of roles within a robotic soccer team (e.g. Emery et al. (2002) and Tambe et al. (1999)), but is subject to the same domain characteristics as Stone and Veloso's work. Live task modification addresses the dynamic (re)allocation of agents between many teams in order to efficiently perform a scenario, rather than reshuffling agents within a team. This type of reshuffling can be considered to be a constrained form of mutable teams, but does not explore the challenges of coordinating the transfer of agents between teams. This work also does not represent the concept of optional roles: instead, agents are being exchanged between a fixed set of required roles.

2.4 Summary

There are three broad areas of work related to proactive replanning: existing planners and planning/execution architectures, duration prediction algorithms, and ap-

proaches to mutable teams and live task modification. The ASPEN and EUROPA planners appear to be the best suited to proactive replanning applications, due to their support for durative actions, temporal constraints, exogenous events, metric resources, and continuous replanning. The CLARAty/CASPER and IDEA architectures (which incorporate ASPEN and EUROPA, respectively) are both good fits for proactive replanning, as they provide the potential for the tight planner-executive ties necessary for successful proactive replanning. Work in the area of function approximation is best suited to duration prediction, although no prior work addresses the precise problem we are solving. A variety of approaches similar to mutable teams and live task modification exist, but no planning/execution systems incorporate the dynamic shifting of agents between teams in the process of executing tasks, nor do existing planners represent optional roles or the commitment of agents to only a portion of a task.

Chapter 3

Background

As with most worthy endeavors, proactive replanning would not have been possible had we not been able to build upon the work of others. This chapter discusses the core technologies upon which we have designed our approach to proactive replanning. While we have selected a specific planning / execution system upon which to base this implementation, note that proactive replanning is applicable to a broad range of planners.

We have made use of three primary technologies: ASPEN, CASPER, and kernel density estimators. ASPEN (Chien et al., 2000b) is an iterative-repair planner developed at the Jet Propulsion Laboratory that we have extended to support the various tenets of proactive replanning. Its design renders it amenable to proactive replanning, as well as allowing its straightforward extension. We discuss ASPEN's structure here, as well as the extensions we have made to its core to support proactive replanning. CASPER (Chien et al., 2000a) is the planning/execution system developed by the Jet Propulsion Laboratory that makes use of ASPEN as its planning element. CASPER provides several viable executives: we have made use of the relatively simple, single-threaded option, but a more flexible TDL-based (Simmons and Apfelbaum, 1998) executive is also available. Finally, kernel density estimation (Silverman, 1986) is at the core of our approach to duration prediction. It has been applied to a wide range of domains, and is an extremely flexible method for extrapolating distributions from a data set.

3.1 ASPEN Planner

ASPEN (Automated Scheduling/Planning ENvironment, (Chien et al., 2000b)) is a modular planning/scheduling system designed to support a wide range of scheduling domains, with a focus on spacecraft operations and the autonomy necessary

there. As with other iterative-repair planners, ASPEN generates an initial (usually flawed) plan, then iteratively repairs detected conflicts until the schedule is valid. ASPEN provides a core set of capabilities that the domain expert may extend in a variety of fashions to better represent the domain in question. These extensions primarily take the form of heuristics, which are used throughout ASPEN's plan repair and optimization processes. In addition, we have extended the core of ASPEN to provide capabilities necessary to operate efficiently in our experimental domains.

3.1.1 ASPEN Core

The core of ASPEN consists of five main components: the schedule database, the temporal constraint network, a set of timelines, the parameter constraint network, and a set of repair and optimization methods.

Schedule Database

ASPEN's schedule database is the central repository of information about all tasks[1] currently on the schedule or under consideration. A task, the basic unit of data operated on by ASPEN, represents the actions that must be taken to accomplish some goal, and has an associated start time, end time, and duration. In addition, a task may require a certain set (or amount) of resources, contain a set of internal and external parameter dependencies, and be temporally constrained relative to other tasks. Tasks also may be explicitly hierarchical, with one or more possible decompositions into sets of subtasks. This decomposition facility allows schedules to be reasoned about at a variety of levels of abstraction, given an appropriate domain structure. The schedule database tracks and maintains the relationships within a hierarchical task tree. The database serves as the central hub through which other ASPEN modules interact, and facilitates rapid operations upon tasks or groups of tasks.

Temporal Constraint Network

A Temporal Constraint Network (TCN) is a graph-based structure used to represent temporal constraints between points in different tasks. In ASPEN, constraints may be applied to either the start or end time of a task to impose temporal constraints such as ordering, minimum spacing, or maximum spacing. Nodes in the TCN are the end points of tasks, with each edge representing a constraint between a pair of

[1] The ASPEN literature refers to the central data structure of the system as an *activity*; we use the term *task* interchangeably, and will refer to activities as tasks throughout this document for consistency.

points. All constraints in the TCN must be satisfied simultaneously for a schedule to be valid: that is, the TCN represents the conjunction of all constraints between active tasks in the schedule database. The TCN may be queried for the set of currently violated constraints, and provides specialized propagation mechanisms to propagate rapidly changes throughout the network.

In this work, the TCN is used to enforce ordering constraints on sets of tasks that must be performed serially: for instance, the components for a communications tower must be transported to the construction site before they are assembled. The TCN is also utilized in our representation of mutable teams to ensure that the tasks assigning agents to a team are contained within the bounds of the cooperative task.

Timelines

ASPEN provides two general classes of timelines: resource and state. Both represent the value of a given variable as it changes across the planning horizon, and allow tasks to place reservations upon them. Resource timelines represent a metric resource, such as battery power, available memory, or the number of available agents. The domain specification may place upper or lower limits on the amount of the resource that remains, with a timeline being defined as in conflict if at any point the specified limits are violated. A schedule is invalid if at any point a timeline is in conflict, and must be repaired until resource usage again falls within the specified bounds.

Reservations upon a resource timeline result in a step change in the amount of resource at the beginning of the task. This is a simplifying approximation; the actual consumption of a resource such as battery power more likely will be approximately linear across the task's duration. Resource timelines may be specified as either depletable or nondepletable. Reservations on the former type result in a permanent change in the amount of resources; depletable timelines are appropriate for modeling a resource that may be described as being consumed. In contrast, a reservation on a nondepletable timeline affects the available amount of the resource only for the duration of the task, with the used amount returning once the task completes. Nondepletable resource timelines are useful for moderating contention for a limited set of items, such as the use of a particular piece of equipment.

Atomic timelines are a specialization of nondepletable resource timelines: they have a maximum capacity of 1, a minimum capacity of 0, and tasks may reserve exactly one unit. These are used to ensure that only a single task may access a given agent or piece of equipment at any given time.

In this work, we make use of nondepletable timelines to model the number of agents assigned to cooperative tasks, ensuring that the minimum set of agents is al-

ways provided. We also provide a single atomic *lock* timeline per agent. Each task utilizing an agent must place a reservation on the agent's lock timeline, ensuring that an agent is not committed to two or more simultaneous tasks.

In addition to resource timelines, ASPEN supports state timelines. State timelines are used to represent a set of discrete states and the valid transitions between them. The domain designer specifies the legal states and transitions for a given timeline. For instance, a camera may have three states: *cold*, *warming*, and *ready*, if it must be warmed prior to use. Legal transitions in this case may be $cold \rightarrow warming$, $warming \rightarrow ready$, and $ready \rightarrow cold$.

Tasks may place a reservation for a specific state on the timeline, which will result in a conflict if the timeline is not in the appropriate state during the reservation. Tasks also may change the state of a timeline, either at the beginning or end of the task. Any attempt to change the timeline through an illegal transition results in a conflict that must be resolved before the schedule may be considered to be valid.

We use state timelines to represent the gross positions of agents by specifying discrete sites of interest, such as (*Lander*, *Habitat*, and *Communications*), as well as two intermediate states (*Moving* and *Stranded*). An agent is *moving* if it is participating in a task that is travelling between two of the sites, and becomes *stranded* if it leaves a team prior to arriving at the destination site. The only way to transition out of the *stranded* state is via a *Move* task. This representation gives the planner a reasonable representation of position, without the complexity of a metric model, and enforces the need for motion tasks to reposition agents as needed.

Parameter Constraint Network

The Parameter Constraint Network (PCN)[2] is used to encode arbitrary intra- and inter-task constraints on task parameters. Every task has a set of system parameters (e.g. start time, end time, and duration) and user-defined parameters (e.g. resource usage, required state, or any other value). The PCN allows constraints to be established from a set of parameters to one other parameter, either within a task or between two separate tasks. A dependency between parameters p_1 and p_2 is defined as a function from one to the other: $p_1 = f(p_2)$. f is an arbitrary function whose input and output types match those of p_2 and p_1, respectively. Note that the inverse dependency is not guaranteed to hold, unless the user explicitly specifies it: that is, it is possible that $p_2 \neq f^{-1}(p_1)$. ASPEN includes a set of useful functions, such as `sum`, `floor`, and the various inequalities. The user also may write additional custom functions for use within the PCN.

[2] In some ASPEN publications, such as Fukunaga et al. (1997), the PCN is referred to as the Parameter Dependency Network (PDN).

When a value is updated, the PCN marks all dependencies for which the updated value is an input as being unsatisfied. When the PCN's satisfaction routine is subsequently invoked, it automatically propagates values and invokes functions to satisfy all affected dependencies. This propagation is delayed for efficiency reasons, allowing many value updates to occur before the cost of a network propagation is incurred. We utilize the PCN for duration prediction by encoding a task's state in a set of parameters that are updated as information arrives from the executive. The state parameters are used as the inputs to a custom function that invokes our prediction algorithm, and outputs the mean of the predicted duration distribution.

Note that the TCN may be considered to be a specialization of the PCN that operates only on the start and end time parameters. ASPEN supports the creation of such targeted derivatives of the PCN whenever the improved efficiency is worth the development costs.

Repair and Optimization Methods

ASPEN includes a variety of repair and optimization methods, which are hard-coded routines that modify the schedule in an attempt to repair a conflict or optimize a metric. Each method contains a series of *choice points* where a decision must be made, such as which conflict to address, which task to move, or which method of repair to apply. Each choice point invokes one of a set of user-specified heuristics to make the selection, selecting a heuristic stochastically, according to user-defined weights. A suite of generic heuristics is included with ASPEN, but the user will normally write ones customized to the domain. The ease with which domain-specific knowledge may be inserted throughout the decision process greatly eases the customization of ASPEN to a given domain. Our heuristics are discussed in detail in Section 8.2.

3.1.2 Approach to Planning, Repair, and Optimization

While ASPEN may be used as either a constructive or an iterative repair planner, we utilize it exclusively in the iterative mode. When repairing or optimizing the schedule, ASPEN selects either a conflict to repair or a metric to optimize, invokes one iteration of an appropriate method, and repeats.

Planning and Repair

Plan repair in ASPEN revolves around the concept of a *conflict*. A conflict is defined as an inconsistency in the schedule. ASPEN uses a wide variety of conflict types to represent the various problems that may occur, such as:

Algorithm 3.1 ASPEN's basic repair algorithm.

```
1:  int numRepairs = 0;
2:  int minConflicts = ∞;
3:  while numConflicts() > 0
            ∧ numRepairs < maxRepairs
            ∧ timeElapsed < maxTime do
4:      Select a conflict (Choice Point)
5:      Select an applicable repair method (Choice Point)
6:      Apply the method (Multiple Choice Points)
7:      if numConflicts() < minConflicts then
8:          Store current schedule as s;
9:          minConflicts = numConficts();
10:     end if
11:     numRepairs++;
12: end while
13: if numConflicts() > 0 then
14:     Reload schedule s.
15: end if
```

- An invalid transition on a state timeline,
- A missing task,
- An unsatisfied dependency in the PCN,
- A violated constraint in the TCN, or
- An oversubscribed resource

Schedule repair consists of the iterative selection of a conflict and the application of a repair method; ASPEN's high-level repair strategy is detailed in Algorithm 3.1. Repair continues until the schedule is free of conflicts, or one of the user-specified limits is reached (e.g. number of repair iterations or time consumed). The user can increase the responsiveness of the planner by specifying such limits, at the cost of potentially not repairing all conflicts. If a repair cycle completes without resolving all conflicts, ASPEN reloads the interim schedule that contained the fewest conflicts (line 14 of Algorithm 3.1).

User-specified heuristics are used to make a variety of decisions at choice points throughout the repair process, such as those at lines 4, 5, and 6 in Algorithm 3.1. Each heuristic is applicable to one or more types of choice points (see

Algorithm 3.2 ASPEN's *Add* repair method

1: Select a task to add that may affect the chosen conflict (**Choice Point**)
2: Create the task
3: Lift the task off of all timelines
4: Ground all task parameters
5: Choose a preliminary duration (**Choice Point**)
6: Set the task's duration
7: Choose intervals into which the task may be placed (**Choice Point**)
8: Choose the start time, from the selected intervals (**Choice Point**)
9: Set the task's start time
10: Choose and set a final duration (**Choice Point**)
11: Place the activity on the timelines

Table 3.2 for a comprehensive list of choice point types). For instance, in line 4 of Algorithm 3.1, a heuristic is invoked to select the next conflict to repair. The user specifies a heuristic that attempts to order the conflicts in a way conducive to their resolution. Note that most of ASPEN's heuristics include a stochastic component to help avoid dead ends, as no heuristic is correct in all situations. For example, the selection of a repair method, the choice of which conflict or task to operate on, and the placement of a new task all involve a random element, generally weighted towards the options that appear to be most appropriate. Once a conflict is selected, another heuristic is applied to choose a repair method (line 5). Repair methods are hard-coded routines for resolving a conflict that may, or may not, succeed in a particular situation. Each method is applicable to a subset of the conflict types, and consists of a series of operations, with choice points interspersed. For example, Algorithm 3.2 details the *Add* repair method, which selects and adds a task to the schedule. It is applicable to all conflicts except open or violated constraints. If the user-specified heuristic invoked at any of the various choice points (e.g. Algorithm 3.2: lines 1, 5, 7, 8, and 10) fails to make a selection, the repair method fails, and the schedule reverts to its state prior to the start of the method's application. ASPEN's default repair methods are summarized in Table 3.1. We added several methods to more efficiently support proactive replanning, which are discussed in Section 3.1.3.

The formation of an initial schedule and the repair of an invalid schedule are treated in the same fashion. When designing a domain model, the user specifies the set of tasks that must be accomplished to complete the scenario successfully. Any missing task is treated as a conflict. As a result, the repair algorithms may be applied to an empty starting schedule to build a complete initial schedule.

Table 3.1: A summary of ASPEN's repair methods.

Method	Description
Move	Move a single task and recalculate its duration.
Add	Add a task to the schedule.
Delete	Delete a task from the schedule.
Connect	Find a sink activity for an open temporal constraint.
Disconnect	Disconnect a temporal constraint.
Move and Connect	Move a task, so as to allow the connection of an open temporal constraint. This is a macro-method, combining *Move* and *Connect*.
Add and Connect	Add a task, so as to allow the connection of an open temporal constraint. Combines *Add* and *Connect*.
Detail	Select a hierarchical decomposition for a task not yet decomposed, and add the chosen child tasks.
Abstract	The inverse of *Detail*: delete all children of a decomposed task.
Reserve	Place a reservation on a timeline.
Cancel	The inverse of *Reserve*: remove a reservation from a timeline.
Place	Place an existing activity onto the schedule.
Lift	The inverse of *Place*: lift an activity off of the schedule, but do not delete it.
Ground Parameter	Select a single value for a task parameter with a range of valid possibilities.
Apply Dependency	Propagate the PCN to resolve any unsatisfied parameter dependencies.
Redetail	Select a new decomposition for a hierarchically decomposed task. This is another macro-method, following an *Abstract* with a *Detail* on the same task.
Satisfy Goal	Add an activity to satisfy a goal (a user-specified requirement for a particular task to occur on the final schedule).

Table 3.2: A summary of ASPEN's choice points, which are decision points within the repair and optimization methods at which user-specified heuristics are applied. A heuristic may be applicable to one or more of these choice points.

Choice Point	Description
Conflict	Select the next conflict to repair.
Resolution Method	Select the repair method to apply.
Culprit to Move	Select an activity to move.
Task Schema to Add	Select the type of task to add.
Culprit to Delete	Select a task to delete.
Culprit to Connect	Select a task to utilize in fulfilling an open constraint of a second task.
Culprit to Abstract	Select a task to abstract.
Culprit to Lift	Select a task to lift from the schedule.
Culprit to Change Duration	Select a task whose duration will be changed.
Culprit to Change Parameter	Select a task containing a parameter whose value will be changed.
Valid Interval	Determine if the task in question may be placed within the specified time interval.
Start Time	Select the start time for a task.
Duration	Select the duration for a task.
Parameter	Change the value of a parameter.
Ground Parameter	Select a value for a parameter from a range of valid options.
Decomposition	Select a hierarchical decomposition for a task from a set of possibilities.
Timeline	Select a timeline upon which to operate.
Reservation to Cancel	Select a reservation to be cancelled.
Preference	Select a preference (i.e. metric) to be optimized.
Satisfy Goal	Select a goal to be satisfied. A goal is a user-specified task that must occur on the schedule.
Constraint	Select a violated temporal constraint to modify.
Preference Optimization Method	Select the optimization method to apply next.

Optimization

ASPEN's approach to optimization is analogous to its repair strategy, with metrics replacing conflicts as the focal point. The domain designer is able to specify metrics (or *preferences* in the ASPEN literature) using a flexible language. It is possible to encode preferences such as *tasks of type T should complete earlier*, *fewer instances of tasks of type T should occur*, or *the sum of all instances of parameter P for tasks of type T should be greater than 20, but less than 100*. For example, when minimizing makespan, we define a preference for the latest task end time to be as early as possible. Each preference is assigned a weight, which guides the heuristic that selects which preference will be addressed in the next iteration of optimization. When optimizing, all preferences are evaluated, the resulting values weighted, and combined into a total score. This score may be used to determine when optimization is complete. Alternatively, the user may specify a number of optimization iterations to perform or a maximum amount of time to spend optimizing. This allows the use of an appropriate amount of optimization for the available time and computational power.

Algorithm 3.3 details ASPEN's basic optimization algorithm. Many of the choice points within ASPEN's optimization methods overlap those used during repair, and the same heuristics may be applied to both. The choice point list in Table 3.2 includes all choice points used in either repair or optimization. Table 3.3 summarizes ASPEN's optimization methods. By iteratively optimizing the schedule, each optimization method may focus on a specific situation, allowing the creation of a suite of simple, directed heuristics and methods that are applied only when interest in relevant preferences has been indicated.

3.1.3 Extensions to ASPEN

In the course of researching proactive replanning, we extended the ASPEN core in a number of ways to add useful functionality and increase the efficiency of repair and optimization. We implemented several new repair and optimization methods, added the concept of user-defined callback functions, extended the expressiveness of ASPEN's state timeline reservations, and added a combined optimize and repair algorithm.

Repair and Optimization Methods

We extended ASPEN's suite of repair and optimization methods with four new methods: *transfer*, *add & setup*, *swap*, and *right shift*. *Transfer* and *add & setup* are applicable to both repair and optimization, while *swap* is used only during optimization and *right shift* is used only during repair.

Algorithm 3.3 ASPEN's basic optimization algorithm.

```
1:  int numOptimizations = 0;
2:  double bestScore = calcPreferenceScore();
3:  while numOptimizations < maxOptimizations
            ∧ calcPreferenceScore() < maxScore
            ∧ timeElapsed < maxTime do
4:      Select a preference (Choice Point)
5:      Select an applicable optimization method (Choice Point)
6:      Apply the method (Multiple Choice Points)
7:      if calcPreferenceScore() > bestScore then
8:          Store current schedule as s;
9:          bestScore = calcPreferenceScore();
10:     end if
11:     numOptimizations++;
12: end while
13: if calcPreferenceScore() < bestScore then
14:     Reload schedule s.
15: end if
```

Table 3.3: A summary of ASPEN's optimization methods.

Method	Description
Move	Move a single task and recalculate its duration.
Add	Add a task.
Delete	Delete a task.
Change Duration	Modify the duration of a task.
Change Parameter	Modify the value of a task parameter.
Abstract	Remove all children of a hierarchically decomposed task.
Lift	Remove a task from the schedule, but do not delete it.
Pack	Shift all activities as close to the start of the planning horizon as possible, without causing conflicts.
Repair	Invoke one iteration of ASPEN's repair algorithm.
Satisfy Goal	Add an activity to satisfy a goal (a user-specified requirement for a particular task to occur on the final schedule).

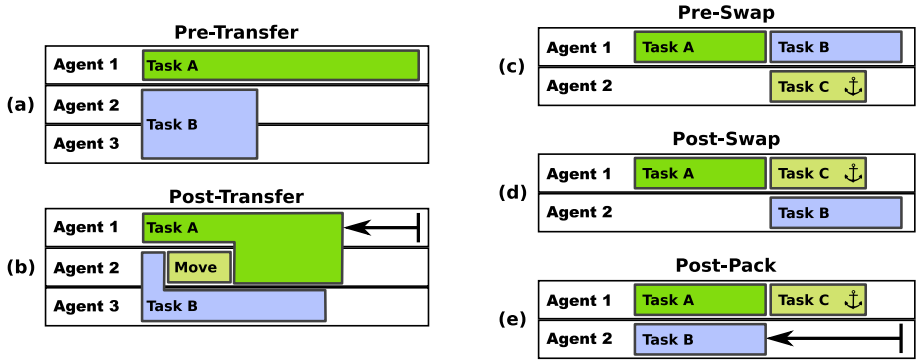


Figure 3.1: Examples of the transfer (a and b) and swap (c - e) methods. The anchor symbol on task C indicates that it is externally constrained to occur at a specific time.

These methods have been added for efficiency: in all cases, it is possible to create the desired changes using ASPEN's default set of methods, but would be difficult to accomplish in practice. Each of our methods consists of a series of steps that must be performed in the correct order and on the same set of tasks. ASPEN's methods generally operate without memory: there is no knowledge of the actions of previously applied methods, or even which methods were recently invoked. This makes the chaining of methods difficult, and suggests the use of macro-methods, such as our new methods or the *move & connect* method in ASPEN's default set of methods.

Transferring encapsulates the transfer of an agent between two mutable teams, and consists of removing an agent from one mutable team (potentially in the middle of a task), adding any necessary setup tasks, and adding the agent to another team. For example, in Fig. 3.1(a), task A dominates the schedule's makespan. By transferring agent 2 from task B (Fig. 3.1(b)), the overall makespan is reduced.

The *add & setup* method is a subset of the *transfer* method used to add an idle agent to a task. It combines the creation of any necessary setup tasks (e.g. movement tasks) with the addition of the agent to a mutable team, performing the same actions as *transfer*, with the exception of removing an agent from a team. *Add & setup* is separate from *transfer* to simplify the reasoning about which method to apply: when transferring, the method-selection heuristic considers the potential impact of the transfer on candidate donor tasks. This reasoning is not necessary when adding an idle agent.

The *swap* method is used to rearrange agent assignments to allow a shorter makespan. It consists of replacing an agent on one team with either an idle agent,

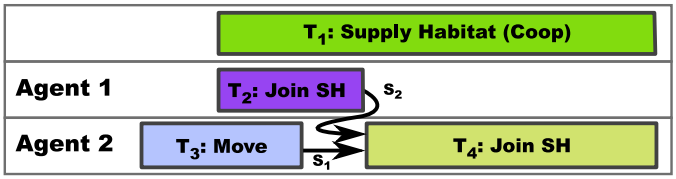


Figure 3.2: A task's slack is the minimum of the slacks associated with all of its predecessor constraints. For purposes of illustration, assume T_1 has a single required role. Here, T_4 has an implicit agent resource constraint to T_3 (with slack S_1) and a role constraint to T_2 (with slack S_2), resulting in zero slack for T_4.

or an agent from another team. If the latter, the replaced agent is assigned to the second team. Fig. 3.1(c) depicts a schedule in which tasks A and B tasks are serialized due solely to contention for a single agent. Task C is performed by a second agent, but is pinned in place due to external constraints. By swapping agents 1 and 2 between tasks B and C (Fig. 3.1(d)), task B may be executed earlier, reducing the schedule's makespan (Fig. 3.1(e)). This often is useful in domains with many multi-agent tasks, where an overlap of one agent between two tasks may needlessly prevent the schedule's compaction.

The *right shift* method is used to repair short conflicts resulting from overlapping tasks assigned to the same agent or violated temporal constraints. When applied, it shifts the start of a conflicting task far enough into the future to resolve the conflict in question. In a dense schedule, this will often result in a new conflict, as the shifted task overlaps the tasks scheduled to follow it. To avoid this cascade of conflicts, *right shift* calculates the tree of successor tasks that depend on the shifted task, then shifts the entire tree into the future. Once the shift is complete, the schedule is repacked, to remove any introduced inefficiencies. The successor tree is constructed by following explicit temporal constraints and implicit constraints due to the use of atomic resources, such as agents.

Critical Path Calculation and Packing

Many of our optimization heuristics are focused on reducing the *critical path*: the sequence (or sequences) of tasks that control the overall length of the schedule, and form a chain of constraints. The *slack* of a task is defined as how much earlier the task may be started than currently scheduled, without violating any constraints, and is calculated as the minimum of the slacks associated with each of the task's constraints. As ASPEN does not include a critical path detection routine, we developed our own that is also aware of a variety of domain-specific constraints.

When calculating the critical path, we first calculate the set of tasks that con-

strain the start of each task. The critical path is the chain of tasks with the minimum total slack (Fig. 3.3(a)), ending with the latest-ending task. Note that multiple critical paths may exist simultaneously, as the latest end time may be shared by multiple tasks.

Many of our optimization heuristics make use of the critical path: operations on tasks outside of the critical path(s) cannot directly reduce the makespan of the schedule. For example, when optimizing the makespan by adding an agent to a team, our heuristics prefer tasks on the critical path with an open optional role. Similarly, when optimizing by deleting a task, we first examine the join tasks representing agents filling an optional role on the critical path to determine if the optional agent is constraining the cooperative task. Finally, the total slack on the critical path is used when determining whether an expensive schedule packing action should be undertaken.

Packing and right-shifting involve similar calculations. *Packing* is the removal of as much slack as possible from the schedule, and is commonly used as an optimization method. When packing, the set of unexecuting tasks is iterated across, starting with the earliest task and proceeding to the latest. Each task is moved as far forward on the schedule as possible, removing all of its slack, as illustrated in Fig. 3.3(b). Note that packing is performed on *all* unexecuting tasks in the schedule, not just those on the critical path.

Unfortunately, our representation of mutable teams complicates this conceptually simply operation. We represent a mutable team as a single cooperative task and multiple join tasks, each of which represents the commitment of an agent to a role in the cooperative task. When packing, we consider constraints on the cooperative and join tasks as a single group, and shift the entire group of cooperative and join tasks as a unit. While this maintains the cohesiveness of the mutable team, it is possible for another task to become "trapped" by the mutable team, preventing both from being packed. Consider the scenario in Fig. 3.4. The cooperative task T_1 has three associated join tasks: agent 1 fills the required role of T_1 with join task T_2, while agent 2 fills an optional role during the beginning and end of T_1 with the join tasks T_3 and T_4. During the time agent 2 is not part of the team performing T_1, it is completing an unrelated task, T_5. As a result, when computing the slack prior to packing, the group of (T_1, T_2, T_3, T_4) has zero slack, due to the agent resource constraint between T_4 and T_5. The slack of T_5 is also zero, due to the agent resource constraint between T_5 and T_3. This prevents a naive packing algorithm from packing any of the tasks, even if both agent 1 and 2 are free prior to the beginning of T_1.

In order to prevent such situations from freezing portions of the schedule in place, we search for any constraint loops with a total slack of zero that lead from one member of a mutable team back into a member of the same team, such as the

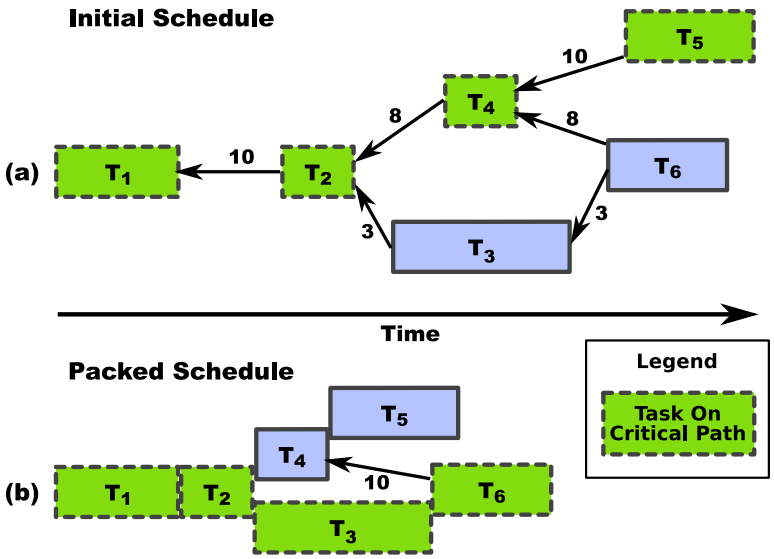


Figure 3.3: In (a), an unpacked schedule is depicted, with constraints between tasks labeled with the associated slack. (b) presents the schedule that results after packing, with tasks in contact having a slack of zero.

Figure 3.4: T_5 is "trapped" by the agent resource constraints of T_3 and T_4 and their constraints relative to their associated cooperative task, T_1. In order to pack a schedule containing this type of fragment, the planner must find constraint loops that begin and end with join tasks of the same cooperative task, then move all tasks in the loop, as well as the cooperative task, as a unit.

$T_4 \rightarrow T_5 \rightarrow T_3$ loop. We group any tasks that are members of such loops with the mutable team for purposes of packing, in the same fashion as the cooperative and join tasks are grouped. Similar loop-detection and grouping is performed during a right shift.

The example presented in Fig. 3.4 is relatively simple; in actual schedules, the trapping of tasks may be arbitrarily complex, especially when another cooperative task becomes involved.

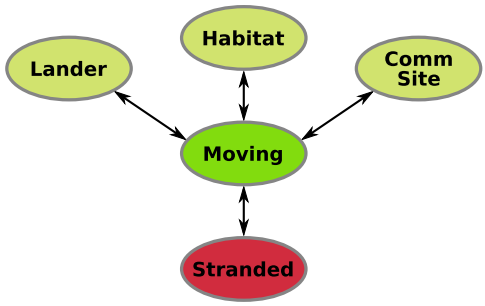


Figure 3.5: The planner's model of agent position in our domain, and the allowed transitions. An agent becomes *stranded* when it leaves a moving task before it reaches its destination. This is an indication to the planner that it must add a *move* task to reposition the agent prior to its next task.

Callbacks

Our broadest extension to ASPEN is the creation of user-defined callback functions. We have defined a series of *callback points*, analogous to choice points, throughout ASPEN's core. These points occur wherever changes are made to tasks or the schedule, and allow user-written code to be executed before, or after, the associated action takes place. This allows the efficient maintenance of domain-specific constraints, timely updating of duration predictions, and any other domain-specific book-keeping. The current set of callback points is listed in Table 3.4. We primarily make use of callbacks to maintain the relationship between *cooperative* and *join* tasks. Cooperative tasks represent a multi-agent task, while join tasks encode the commitment of a specific agent to participate in the task for a particular interval of time. Cooperative and join tasks are discussed in detail in Chapter 6.

Reservations

The standard version of ASPEN is able to place a reservation on a state timeline requiring that the timeline maintain a specific state for the duration of a task. If that requirement is not met, a conflict occurs. However, in modeling our experimental domains, we identified the need for pre- and post-conditions on state timelines: we needed to specify that the state timeline must have a particular value prior to the start of the task, or after the task completed. We extended ASPEN's domain representation language and concept of state reservations to support the specification and enforcement of pre- and post-conditions.

Specifically, this was needed when modeling agent motion. As discussed in Section 3.1.1, we model the position of each agent with a set of discrete locations,

Table 3.4: A summary of the callback points we have added to ASPEN. User callback functions may be invoked at any point, in the same manner as heuristics are invoked at choice points.

Point	Description
Add	Occurs after a task is added to the schedule.
Delete	Occurs prior to the deletion of a task.
Pre-Move	Occurs prior to a task movement, allowing the user to cache the position of the task prior to the move.
Move	Occurs after a task has been moved.
Pre-Duration Change	Occurs prior to a change in a task's duration, allowing the user to cache the duration prior to the change.
Duration Change	Occurs after a task's duration has changed.
Place	Occurs after a task has been placed on the schedule.
Lift	Occurs after a task has been lifted from the schedule.
Detail	Occurs after a decomposition has been selected for a task.
Redetail	Occurs after the decomposition of a task has been changed.
Abstract	Occurs after the children of a decomposed task have been deleted.
Pre-Repair	Occurs prior to each iteration of ASPEN's repair algorithm.
Post-Repair	Occurs after each iteration of ASPEN's repair algorithm.
Pre-Optimize	Occurs prior to each iteration of ASPEN's optimization algorithm.
Post-Optimize	Occurs after each iteration of ASPEN's optimization algorithm.
Pre-Step	Occurs prior to each simulated execution step.
Post-Step	Occurs after each simulated execution step.
Commit	Occurs after a task has been committed to the executive.

a moving state, and a stranded state (Fig. 3.5). Agents are *moving* while they are participating in a task that is transitioning between discrete locations. They enter the *stranded* state if they leave a moving task prior to its completion, and an individual move task must be added to the schedule in order to exit the *stranded* state. As a result of this model, an agent participating in the entirety of a moving task must be in the original location prior to the task's start, in the moving state during the task, and in the destination location after the task's end. While this could have been modeled with three end-to-end tasks, the resulting schedule would be excessively complicated, artificially increasing the difficulty of repair and optimization.

Optimization and Repair

When we began to optimize schedules in ASPEN, we encountered two shortcomings in ASPEN's approach: it was very difficult to interleave optimization and repair, and there was no way to backtrack if adequate progress was not being made during either repair or optimization. We developed an alternative to ASPEN's optimization and repair algorithms to address these shortcomings (Algorithms 3.4 and 3.5, respectively). Our alternative optimization algorithm is used exclusively during the experiments reported in this thesis, while our repair algorithm is used when repairing the schedule during execution and after optimization.

In ASPEN, it is possible to invoke a set number of optimization iterations after each step of execution, preceded and/or followed by the repair of any conflicts in the schedule. Since optimization methods in ASPEN are allowed to create conflicts, this requires the optimization heuristics to be able to reason about schedules containing conflicts, if more than one iteration of optimization is to be performed. This requirement greatly increases the heuristics' complexity. Our alternate algorithm instead interleaves optimization and repair by repairing the schedule after each iteration of optimization (Algorithm 3.4, lines 8-11), allowing the optimization heuristics to assume that the schedule is conflict-free.

ASPEN's iterative, memoryless approach to repair and optimization also created inefficiencies. If a poor decision was made that created many new conflicts or greatly increased the makespan of the schedule, there was no mechanism to retract the decision or otherwise backtrack to a previous point in the repair or optimization process. As a result, a single poor decision significantly degraded the efficiency of the process, as many iterations had to be expended to compensate. In order to lessen the impact of incorrect heuristics, we introduced a form of backtracking into our alternative optimization and repair algorithms (Algorithms 3.4 and 3.5, respectively).

After each optimization (and any resulting repair), the schedule's score is evaluated. If the schedule has the best score observed to date, it is stored for future use

Algorithm 3.4 Our extended optimization algorithm. Differences from the basic algorithm (Algorithm 3.3) are italicized.

```
1:  int numOptimizations = 0;
2:  double bestScore = calcPreferenceScore();
3:  int bestScoreI = 0;
4:  double reloadFraction = 0.9; {User-specified.}
5:  int reloadDelay = 10; {User-specified.}
6:  while numOptimizations < maxOptimizations
            ∧ calcPreferenceScore() < maxScore
            ∧ timeElapsed < maxTime do
7:     Invoke Algorithm 3.3 with numOptimizations = 1, but without reloading.
8:     if numConflicts() > 0 then
9:        Invoke Algorithm 3.5 to repair all conflicts.
10:       Pack (left-shift) the schedule.
11:    end if
12:    if calcPreferenceScore() > bestScore then
13:       Store current schedule as s;
14:       bestScoreI = numOptimizations;
15:    end if
16:    if calcPreferenceScore() < reloadFraction * bestScore
          ∧ numOptimizations - bestScoreI ≥ reloadDelay then
17:       Reload schedule s;
18:    end if
19:    numOptimizations++;
20: end while
21: if calcPreferenceScore() < bestScore then
22:    Reload schedule s.
23: end if
```

(Lines 12-15, Algorithm 3.4). If instead the score is less than a specified fraction of the best score observed so far, and at least a given number of optimizations have occurred since the last reload (e.g. backtrack), the best schedule is reloaded before optimization continues (Lines 16-18, Algorithm 3.4). This form of backtracking allows the optimization algorithm to make decisions that may temporarily worsen the score, while ensuring that the schedule does not degrade significantly over time.

We applied this form of backtracking to repair by taking an iterative deepening approach, rather than using ASPEN's standard linear repair algorithm. The standard approach continues to apply repair methods until a valid schedule is built, with

Algorithm 3.5 The iterative deepening repair algorithm.

```
1:  double repairDepth = 20; {The initial value is user-configurable.}
2:  double deepeningScale = 1.5; {Also user-configurable.}
3:  Store schedule as s;
4:  while numConflicts() > 0
            ∧ timeElapsed < maxTime do
5:     Invoke Algorithm 3.1 with numRepairs = repairDepth.
6:     if numConflicts() > 0 then
7:        Reload schedule s;
8:        repairDepth *= deepeningScale;
9:     end if
10: end while
```

no backtracking. When repairing conflicts resulting from an optimization attempt or the update of a task's predicted duration, a valid schedule often may be achieved after a relatively few iterations of repair, but a poor choice at one of the repair algorithm's choice points may cause additional conflicts, resulting in an excessively long and disruptive repair cycle. Our approach (Algorithm 3.5) initially performs N iterations of repair. If, at the end of this repair cycle, the schedule is still invalid, we backtrack to the initial schedule, increase N, and repeat. This continues until a valid schedule is found. N is multiplied at each stage by a user-defined value, steadily increasing the allowed number of repair iterations. This approach is successful because of the stochastic nature of our heuristics: a sequence of decisions will almost never be repeated, even when starting from the same initial state. While a poor choice still will result in wasted effort, the mistake is recoverable during the next backtrack. We use Algorithm 3.5 when repairing conflicts resulting from execution-time anomalies or optimization attempts.

3.2 CASPER Execution System

CASPER (Continuous Activity Scheduling, Planning Execution and Replanning) (Chien et al., 1999) (Estlin et al., 2000) extends ASPEN to address dynamic planning and scheduling applications. It adds a real-time system (i.e. executive) capable of monitoring the execution of a plan and providing updates to ASPEN as tasks begin or end, states change, and resources are used (Fig. 3.6). The updates may introduce conflicts or opportunities into ASPEN's plan, which are addressed through plan repair and optimization. The vision for CASPER, as articulated in Estlin et al. (2000), is to support distributed, dynamic, and continuous replanning.

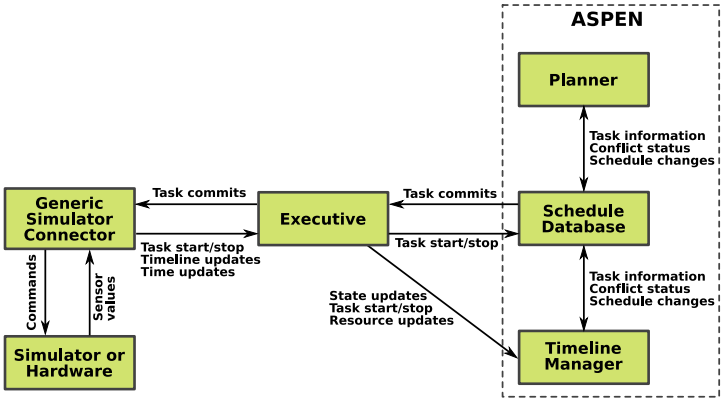


Figure 3.6: The CASPER architecture. The *Planner*, *Schedule Database*, and *Timeline Manager* components comprise the ASPEN proper. Adapted from Chien et al. (1999), where the *Executive* is termed a *Real Time System*.

Algorithm 3.6 CASPER's execution algorithm, per Chien et al. (1999).

Initialize P to the null plan
Initialize G to the null set
Initialize S to the current state

Given a current plan P and a current goal set G:
loop
    Update G to reflect new goals or goals that are no longer needed.
    Update S to the revised current state
    Compute conflicts on (P, G, S)
    Apply conflict resolution planning methods to P (within resource bounds)
    Release relevant near-term activities in P for execution
end loop

A central batch-mode planner is suggested, working at a relatively abstract level to assign tasks among a set of agents. Each agent utilizes a local planning and execution system to accomplish the assigned tasks, with updates to the plan occurring as tasks complete. The individual agents repair and optimize their plans in response to the provided updates, without the need for interaction with the centralized planner or remote human controllers. The CASPER framework provides each agent with this localized replanning capability.

We have utilized CASPER in a centralized fashion, planning for multiple (sim-

ulated) rovers while accepting state updates from the entire team. In addition, we have extended CASPER and ASPEN to provide state updates throughout the execution of a task, which we utilize to predict the remaining duration of a task as execution proceeds. This provides our proactive replanning framework with the ability to detect conflicts and opportunities earlier than is possible with the standard CASPER system, and allows more efficient schedule execution.

CASPER provides two implementations of the executive[3] in Fig. 3.6. The first implementation is embedded in the same thread of execution as ASPEN, and provides a tight link between the two. The embedded executive is able to update the schedule database and timeline manager directly, minimizing the delay between when an execution anomaly is detected and when the planner is informed. This executive models tasks simplistically by maintaining start and end times, and ensuring that resource bounds are not violated. It has no knowledge of the temporal or other constraints on tasks that are modeled in the planner, and thus is unable to react in any meaningful fashion to unexpected situations. The other disadvantage of this approach is that the executive and planner may not operate in parallel. Since they are running in the same thread, they must perform computation in an interleaved fashion. As a result, if plan repair is less efficient than expected, the executive's next cycle will be delayed, potentially postponing the arrival of useful state updates. We have used an embedded executive throughout this work. The tight planner-executive integration simplified our implementation, and the synchronous nature of the embedded executive and simulator allowed the evaluation of proactive replanning without the confounding effects of real-time execution.

The alternate executive is TDL-based (Simmons and Apfelbaum, 1998)[4], and executes in a separate thread. TDL allows this executive to duplicate the hierarchical task structure of the planner, as well as explicitly include temporal constraints between tasks. This increases the complexity of the executive, as much domain information must be duplicated between planner and executive. However, the additional information allows the executive to maintain temporal constraints in the face of execution-time anomalies. For instance, if a task is delayed, the TDL-based executive is able to infer that the start of all tasks constrained to begin after the delayed task should be delayed in turn. Since the embedded executive tracks only task start and end times, with no knowledge of temporal constraints, it would be unable to take such action, and would instead have to rely on the planner to detect the delay and adjust the remainder of the schedule quickly and appropriately. As the TDL executive is operating in a separate thread from the planner, it also is

[3] Some CASPER publications, such as Chien et al. (1999), refer to the executive as the *Real Time System*. We shall use the term *executive* for consistency with the remainder of the thesis.

[4] TDL is an extension to the C++ language that provides mechanisms for modeling and executing hierarchical task networks.

able to monitor execution while the planner is repairing or optimizing the future schedule. This increased responsiveness is offset to a degree by the need to acquire mutexes whenever a state update is applied to the database shared by the executive and planner. This executive has significant advantages, especially when applied to a real-world robotic team. However, we chose to evaluate the potential of proactive replanning in isolation from the vagaries of asynchronous, real-time execution, which made the increased complexity of the TDL-based executive not worthwhile.

The standard CASPER also provides a simple, deterministic simulator in which tasks complete in exactly the scheduled amount of time, and no failures occur. The simulator is designed to be extended for the domain in question. We have done so through the use of our TaskSim library, which stochastically simulates the execution of tasks. TaskSim is based on Augmented Transition Networks (Woods, 1970), and is discussed in detail in Section 8.1.3.

3.3 Kernel Density Estimation

We use a form of kernel density estimation (KDE) (Parzen, 1962) (Silverman, 1986) to predict duration distributions. KDE is a non-parametric method related to histograms that is used to estimate an arbitrary distribution from training data without making *a priori* assumptions about the form of the underlying distribution. A histogram can be thought of as a set of unit-height blocks, where each observation generates a block. The blocks are aligned with the histogram bins into which the corresponding observations fall, and are stacked (summed) when multiple blocks fall into a single bin. A simple kernel density estimator performs in a similar fashion, except that each block (or *kernel*) is centered on the observation, rather than on a discrete bin. Summing these blocks results in a step-wise function. In practice, rather than using a discrete square-wave kernel, KDE often utilizes a smoother function, such as the normal distribution. An example of this is depicted in Fig. 3.7, where the five observations are denoted with circles. A normal kernel (dashed lines) is centered at each observation. The kernels are summed to yield the estimated distribution (solid line). The selection of the shape and bandwidth of the kernel affects the resulting distribution. In the case of a normal kernel, the bandwidth is the standard deviation of the kernel distribution. If it is too narrow, the result will have too many modes; too wide, and the distribution will become an undifferentiated mass. When predicting duration distributions, we use a normal kernel, with a bandwidth appropriate to the average spacing of observations.

Denote the bandwidth as h, the kernel function as $K(x, h)$, and let there be n observations with values x_i. The density of the distribution at a value x is then the sum of the densities contributed by the n kernels: $f(x) = \frac{1}{n} \sum_{i=1}^{n} K\left(x - x_i, h\right)$,

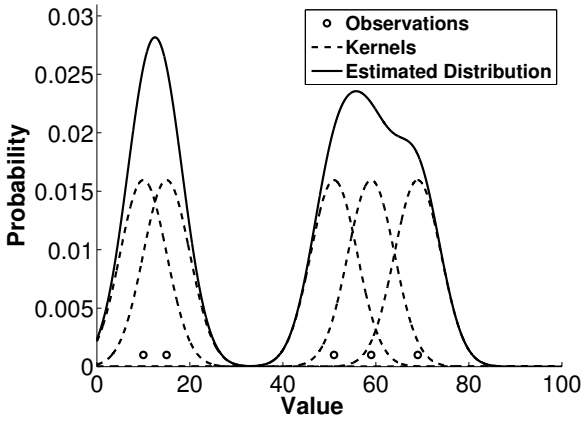


Figure 3.7: A simple example of kernel density estimation. Kernels (dashed lines) are centered at each of the five observations (plotted as 'o's), then the kernels are summed to build the estimated distribution (solid line).

where in our case $K(x,h) = \frac{1}{h\sqrt{2\pi}} exp\left(-\frac{x^2}{2h^2}\right)$.

Under weak assumptions, it has been shown that no non-parametric function approximator may converge to the underlying distribution faster than kernel density estimation (Wasserman, 2003). This rapid convergence, combined with ease of use and flexibility, allows the application of KDE in a wide variety of fields, from archaeology (Beardah, 1997) to economics analysis (Estany and Losilla, 1998).

3.4 Summary

This thesis builds upon the flexible and modular ASPEN planner. ASPEN's design allows its natural extension to support a wide variety of domains. While we have augmented the ASPEN core with several new capabilities, they were largely intended to increase efficiency: ASPEN's core functionality is well-suited to the task of proactive replanning. The CASPER execution architecture supports continuous updating of ASPEN's resource and state timelines in response to execution-time events. This provides the rapid state updates needed by proactive replanning. By extending CASPER to provide state updates throughout the execution of a task, we are able to perform live duration prediction, and accrue the resulting benefits. Finally, our approach to duration prediction makes use of kernel density estimation as a flexible, non-parametric function approximator. KDE's convergence properties, ability to model arbitrary distributions, and direct use of training data make it ideal for our domain.

Chapter 4

Approach

4.1 Proactive Replanning

Proactive replanning is a two stage process: first, prediction of future conflicts and inefficiencies and, second, use of the predictions to replan and modify executing tasks. The goal, of course, is to make modifications early enough in the task to avoid problems or to take maximum advantage of opportunities. Any discussion of proactive replanning divides naturally into two broad topics: prediction and replanning. A discussion of prediction could include a wide variety of interesting topics, such as: task duration, exogenous events, resource consumption, and the likelihood of component failure. This thesis, however, addresses *duration prediction*: the prediction of task duration, both prior to, and throughout, execution.

Similarly, a discussion of replanning gives rise to many issues, incorporating a plethora of replanning and repair strategies and techniques. In this work, we concentrate on the concepts of *mutable teams* and *live task modification*. *Mutable teams* allow the addition or removal of agents while a task is underway. *Live task modification* is the act of modifying a task that is already being executed, generally through changing the assignment of agents to the task.

4.1.1 Live Duration Prediction

Duration prediction is the prediction of how long a task will take to complete, given some measure of the its current state and any available knowledge about future changes to the task, such as the scheduled arrival or departure of agents. Specifically, we predict a distribution across possible durations for the task at hand, given the task's current state and a corpus of training data. Predicting a distribution, instead of just the mean, enables a variety of replanning and repair strategies, such as multi-metric optimization (e.g. simultaneously minimizing makespan and the

amount of schedule repair), principled jitter management (reducing the amount of replanning caused by small changes in predicted duration), and the most efficient use of mutable teams (evaluating the performance of executing teams and reassigning agents appropriately). We make use of a database of example execution traces to build non-parametric estimates of the underlying duration distribution.

Throughout this document, we will distinguish between the general act of *duration prediction* and the more specific *live duration prediction*. Duration prediction is the act of predicting a task's remaining duration, while live duration prediction is the use of this technique throughout a task's execution. Live duration prediction serves to continuously update the planner's schedule and allows it to rapidly detect execution anomalies.

By predicting events such as time over-runs and under-runs, the planner is able to rework its schedule to be more efficient or value-laden, thus taking advantage of predicted "holes" in the schedule and resolving conflicts in time to issue corrected instructions to the executive. Live duration prediction is useful in its own right, yielding an 11.7% greater reward in our experimental scenario, as compared with an otherwise identical planner that only receives task execution updates when the task completes or over-runs its scheduled time. This improvement corresponds to 45% of the gain achievable by an omniscient planner. When used as part of a larger proactive replanning framework, duration prediction provides the information necessary for live task modification to have a significant effect.

4.1.2 Mutable Teams: Required and Optional Roles

Mutable teams are those that agents may join or leave throughout the execution of a task. This implies a degree of flexibility in the design of the executive and the task itself. We characterize tasks as having a set of roles, each of which may require a certain number of agents and may make use of a number of additional optional agents. We define a role as having a number of slots, some required, and some optional. All required role slots must be filled for the task to proceed, while optional slots are not necessary, but may improve the task's speed, efficiency, or reliability. Thus, each role has a lower bound on the number of agents that must fill it for the task to proceed and an upper bound on the number that may usefully contribute to the task. For conciseness, we will refer to the set of roles that include optional slots as *optional roles*, and the set of roles that contain required slots as *required roles*. A role may be in both sets simultaneously, if its upper and lower bounds are different and are both greater than zero.

For instance, it may be possible to transport a heavy component from a lander to the worksite with only one agent in the transporter role. However, one or two more transporter agents may be used to distribute the load, increase the

team's speed, and reduce the likelihood that the team will become mired in sand. In addition, if a robot is available to act in the scout role, the team's route may be improved, increasing the probability of avoiding terrain related problems. In this example, the transporter role has bounds of $[1, 3]$: at least one transporter slot must be filled, while up to three agents may be of use. The scout role is purely optional, with bounds of $[0, 1]$: no scouts are required, but one may be utilized if it is available.

In our formulation of mutable teams, all required role slots must be filled throughout the execution of a task, although agents may hand off a required role slot in mid-task, as long as it is continuously filled. Optional role slots may be filled for any fraction of the task's duration, and also may be handed off between agents. In general, filling optional role slots reduces the likelihood of failure, increases the team's rate of progress, or reduces the impact of any failures. For conciseness, we refer to a schedule of agent arrivals and departures from a team as a *team profile*; a specific arrival or departure is termed a *team change*.

When used with immutable teams, optional roles allow the proactive replanner to coarsely trade resources (the number of assigned agents) for reduced task duration and/or increased task reliability, increasing the planner's flexibility[1]. This flexibility is further enhanced when mutable teams are incorporated. They allow the trade-offs of resources against duration or reliability to be much finer-grained, since agents may be assigned to optional roles for only a portion of a task. Live task modification further leverages optional roles by allowing the planner to shift resources assigned to tasks already underway in response to the realities of execution by taking actions such as balancing the schedule or adding assistive resources to teams in trouble.

The planner may make use of these capabilities in a variety of ways. The load-balancing characteristics of mutable teams allow the planner to utilize idle agents in supporting roles to speed the execution of critical tasks. Mutable teams also provide additional tools to the planner when resolving scheduling conflicts: it may be possible to resolve an agent oversubscription conflict by removing an optional agent from the team, while adding optional agents may reduce the task's predicted duration sufficiently to avoid a conflict entirely.

As with live duration prediction, mutable teams are useful when utilized in isolation. Initial schedules constructed with mutable teams for our evaluation scenario were on average 5.65% (33.04 minutes) shorter than those built with immutable teams, a statistically significant difference. Mutable teams provide additional benefits when used within the overall proactive replanning framework. For example, when live duration prediction and live task modification are both available, it is

[1]This is somewhat similar to the resource smoothing process used in project management.

possible for the planner to identify a team that is operating inefficiently and compensate by transferring additional agents to fill vacant optional roles.

4.1.3 Live Task Modification

We define *live task modification* as the ability to make changes to an already executing task. In particular, this thesis addresses the ability to adjust the *team profile* of an executing task by adjusting the arrival and departure times of assigned agents, as well as assigning new agents to an ongoing task or removing currently assigned agents. This implies a much tighter integration between the planner and executive than is common in most planning and execution systems. In such systems, once a task has been committed[2], it is out of the control of the planner, which is restricted to adjusting the uncommitted tasks on the schedule in response to any reported execution anomalies. In most existing systems, the loose connection between executive and planner yields only a minimal flow of information. Because this does not allow the planner to perform any sort of duration prediction, its ability to adjust is further limited.

In contrast, this work emphasizes a close relationship between planner and executive, with the executive providing frequent state updates to the planner and the planner adjusting the team profiles of executing tasks in response to the realities of execution. Live task modification requires additional flexibility of the executive, above and beyond that required by mutable teams. If only mutable teams are utilized, the executive is provided a fixed team profile that it may depend on, subject to potential delays of arriving agents due to execution problems with other tasks. The executive may utilize a fixed team profile in a variety of ways. For instance, if a transport team has become mired in the sand, the executive may forego a potentially damaging recovery maneuver if it knows an agent arriving shortly will enable a safer method. The addition of live task modification requires the executive to accept adjustments to the team profile on the fly. Needless to say, live task modification as formulated here is of little use in the absence of mutable teams.

While live task modification is meaningless with immutable teams, it is able to utilize both live duration prediction and mutable teams to improve the execution of a schedule when used as part of a comprehensive proactive replanning strategy. We have found that live task modification reduces the makespan of an executed schedule by a statistically significant 6.5% when added to a system already using live duration prediction and mutable teams.

[2]Following the conventions of ASPEN and CASPER, we refer to a task that has begun execution as being *committed* to the executive by the planner.

4.2 Requirements for Proactive Replanning

Implementing proactive replanning requires a number of capabilities in the planner and executive, as well as a richer interaction between them. The planner must be able to represent duration distributions, as well as complex cooperative tasks and the weave of agents joining and leaving them. The executive must support the addition and removal of agents throughout the execution of a task, a capability not normally available. Most importantly, proactive replanning necessitates a much closer relationship between the planner and executive than is normally provided: the executive must provide the planner with frequent state updates, while accepting changes to the team profiles of executing tasks. While proactive replanning requires a more tightly integrated and more complex system than other approaches, the marked improvements in the executed schedules make the effort well worthwhile.

4.2.1 Planner

The planner must support durative actions, temporal constraints, and be able to quickly replan or repair a plan in response to feedback from the executive. To make full use of the opportunities provided by live duration prediction, it must be possible either to represent task durations as distributions, or to be able to cache predicted distributions to allow the repair or replanning mechanisms to utilize them. While it is more elegant to directly represent task duration distributions, it is often computationally inefficient to do so. Mutable teams require that the planner be able to encode the duration of a particular agent's participation in a task: it is insufficient to represent solely which agents are participating in a task. In addition, the planner must be able to encode the existence of distinct roles within a task, as well as lower- and upper-bound constraints on the number of agents assigned to each role. It must also be possible to assign multiple agents to the same role slot at disjoint times. Finally, live task modification requires the planner to reason about executing tasks and rapidly repair and optimize the schedule in response to events occurring during execution.

4.2.2 Executive

The primary requirement that proactive replanning places on the executive is flexibility. The executive must be able to support the addition and removal of agents from a team during the performance of a task, as well as the dynamic adjustment of the team profile. This requires careful design of the executive and the tasks themselves: it must be possible to add, remove, or exchange agents in any role at any

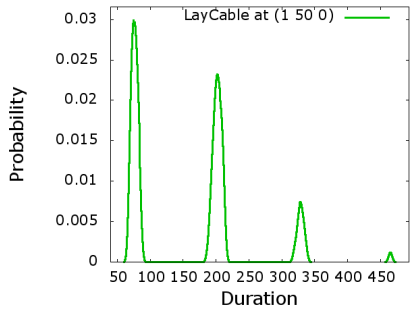
time. Needless to say, not all tasks will be amenable to this approach: such tasks are encoded as having being immutable. Other tasks may be amenable to team changes only at discrete points during their execution. We have not explored the implications of such tasks; how to effectively encode and reason about such tasks remains an open area of research. While we envision a distributed set of executives coordinating with a centralized planner, this thesis explored proactive replanning in a centralized, simulated fashion: there may be unexplored opportunities and difficulties that arise from a distributed executive. Finally, the executive must be able to track the state of executing tasks and frequently provide updates to the planner, in order to support various aspects of proactive replanning.

4.3 Domain Complexity

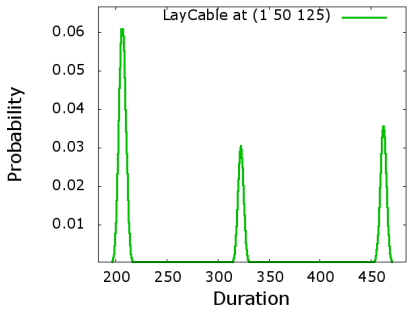
The three components of proactive replanning are evaluated throughout this thesis in two distinct domains. Both domains involve eight distinct types of tasks, each with a unique structure with respect to execution, potential failures, and the effects of filling optional roles. We model tasks at a mid-range level of detail: stochastic elements of execution and nonterminal failures are encoded, but low-level aspects of the agents are not considered. For example, while we model the effect of the weight of a large cable spool on the transporting team, we do not explicitly consider the likelihood that the back-left wheel of agent 2 may become packed with sand. We also do not model terminal failures, nor agent breakdowns: given sufficient assigned agents, tasks will always complete, although the time needed may be arbitrarily long.

The *Lay Cable* task is representative of the complexity of the tasks in our domains, and is performed in the experiments of Chapters 6 and 8. The objective of *Lay Cable* is to deploy a data cable between the communications array and the habitat. It has a single *Cabler* role, with agent bounds of $[1, 3]$. The weight of the cable slows the team, although its impact decreases as the task nears completion and the weight of the remaining cable falls. One type of nonterminal failure is modeled: if the cable becomes caught on the terrain, the team must stop until the cable is untangled. Additional agents reduce the effect of the cable weight, as well as drastically decreasing the time needed to detangle the cable, resulting in modest reductions in task duration in the nominal case and significant time savings when failures occur.

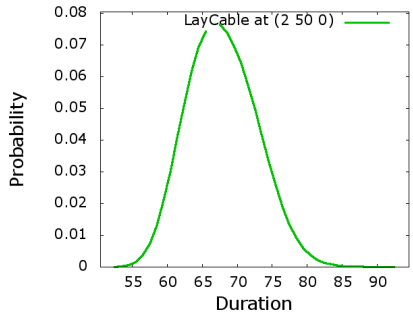
There are three sources of uncertainty in *Lay Cable*: the progress made towards the habitat during nominal operation includes a Gaussian noise component, there is a 0.75% likelihood that the cable will become tangled during each step of execution, and the delay induced by a tangled cable is drawn from a uniform distribution.



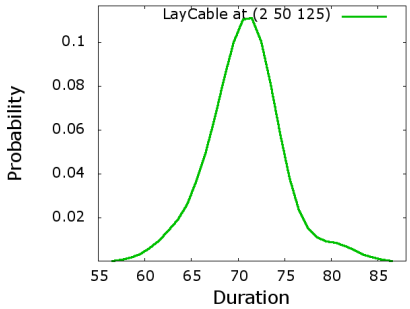
(a) Start of task, one agent.



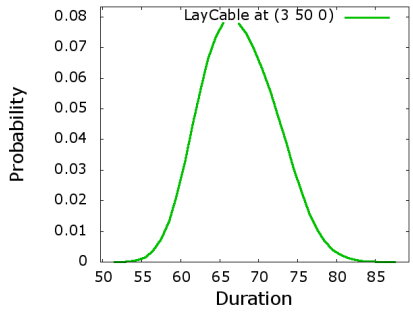
(b) After failure, one agent.



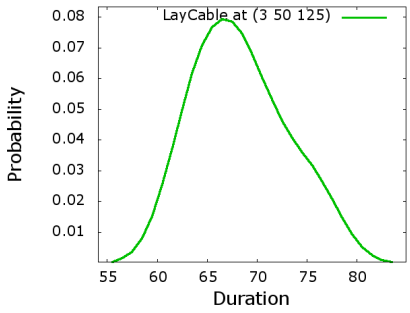
(c) Start of task, two agents.



(d) After failure, two agents.



(e) Start of task, three agents.



(f) After failure, three agents.

Figure 4.1: The filling of at least one optional role in the *Lay Cable* task nearly eliminates the effects of failures, which significantly increase the task's duration when only one agent is available. Each row of images corresponds to a different number of agents, the left column depicts the estimated duration distribution at the start of the task, and the right column plots the duration distributions if a failure occurs immediately following the start of execution. See Appendix B.2 (Listing B.14) for structural details of the task.

When no failures occur, the expected duration of the task is a normal distribution, with optional agents slightly reducing the mean (left-most mode of Fig. 4.1(a); Figs. 4.1(c) and 4.1(e)). The uncertainty in this nominal case is a result of the stochasticity of the progress made during each timestep.

When operating with a single agent, the impact of a failure is significant, as can be seen in Figs. 4.1(a) and 4.1(b). These figures depict the estimated duration distribution for a *Lay Cable* task prior to and immediately after a failure has occurred. Each mode of the distribution corresponds to a number of failures that may occur during the course of the task. As can be seen from Fig. 4.1(a), we have observed up to three failures in our training data. When at least one optional agent is available (Figs. 4.1(d) and 4.1(f)), the impact of the failure is greatly reduced.

By modeling tasks that incorporate significant uncertainty and multi-modal duration distributions, we are able to explore a number of the complexities inherent in real-world systems while retaining the ability to collect a large quantity of data through simulation.

4.4 Summary

Proactive replanning is the act of predicting future problems or opportunities and rapidly replanning to ameliorate or take advantage of them. This thesis addresses three aspects of proactive replanning: *(live) duration prediction*, *mutable teams*, and *live task modification*. Each places different requirements on the planner and executive, which must be much more tightly integrated than in other planning and execution systems. A continuous flow of information from the executive to the planner drives rapid replanning and schedule repair. Each aspect of proactive replanning is able to leverage capabilities provided by the others to form a replanning system that is able to fluidly react to the realities of execution by shifting resources on the fly to address identified inefficiencies and opportunities. While each component is useful to a varying degree in isolation, when used together in a unified proactive replanning framework, they form a system significantly more capable than the sum of its parts.

Chapter 5

Live Duration Prediction

5.1 Overview

When working with others, humans often exchange information about their progress on the tasks at hand and whether they expect to complete their work on time. This allows each individual to adapt his schedule to make the best use of his time. For instance, the foreknowledge that a group meeting will be delayed by an hour because the team leader is caught in traffic allows everyone to take on an appropriate task during the now-free hour. Many planning and execution systems, however, do not predict how long executing tasks will take to complete. Instead, they assume each task will take as long as it was scheduled for and react only when tasks complete early or over-run their scheduled times, resulting in suboptimal execution.

Proactive replanning encompasses the prediction of problems, or opportunities, such as these, and the adaptation of the schedule to avoid, or take advantage of, them before they occur. This allows the proactive replanner to modify its schedule early enough to accommodate the realities of execution: by predicting the team leader's late arrival from his current location and the state of the roads, a proactive replanner would schedule additional tasks into the now-empty hour for the remainder of the team, and move tasks aside to accommodate the delayed meeting. Live duration prediction is a vital element of proactive replanning, allowing the planner to foresee problems and opportunities far enough in advance to allow appropriate action to be taken. Throughout this chapter, we will discuss duration prediction and live duration prediction. The former is the construction of an estimate of a task's remaining run time, given a measurement of its current state, while the latter consists of repeatedly constructing such estimates throughout a task's execution.

Rather than building a single estimate of the remaining duration, we estimate a distribution across the possible task durations. Predicting a duration distribution

allows the planner to engage in several new strategies that are unattainable if we simply compute a scalar duration, such as multi-metric optimization, reasoning about deadlines, the efficient use of mutable teams, and prediction jitter compensation.

We need training data to predict the duration distribution of a task, since most realistic tasks involve stochasticity that cannot be accurately modelled *a priori*. This training data is necessarily relatively sparse: the state space of tasks usually involves several continuous dimensions, making it extremely difficult to collect a dense set of data outside of simulation. This necessitates the use of function approximation techniques to estimate the duration distribution. We have developed a kernel density estimation-based approach to duration prediction that enables the estimation of duration distributions given sparse training data.

We have evaluated our approach using the ASPEN planner (Chien et al., 2000b) and a stochastic execution simulator in a variety of ways. Section 5.7.1 discusses the accuracy of our prediction method as a function of the amount of training data. Section 5.7.2 experimentally evaluates the effects of adding live duration prediction to a planning and execution system. Our experimental results indicate that the sole use of live duration prediction increases the total reward over the baseline by a statistically significant degree. The increase amounts to 45.0% of the possible improvement, as compared with an omniscient planner.

5.2 Applicability

Live duration prediction allows the planner to recognize future scheduling problems and opportunities in time to address, or take advantage of, them. For instance, if a task is predicted to over-run, it will delay other tasks, potentially creating opportunities to insert tasks into the predicted window of now-idle time. Without duration prediction, the planner would miss such opportunities. In addition, predicting future events provides the planner with a longer time window in which to repair or optimize the plan before execution reaches the problem point. This reduces the likelihood that execution must be paused to allow the planner to resolve scheduling difficulties, and increases overall efficiency.

Live duration prediction allows the prediction of two classes of execution anomalies: under-runs and over-runs.

5.2.1 Under-runs

When a task is predicted to under-run, setup actions for any subsequent tasks may be started early, decreasing or eliminating dead time between tasks. Fig. 5.1 de-

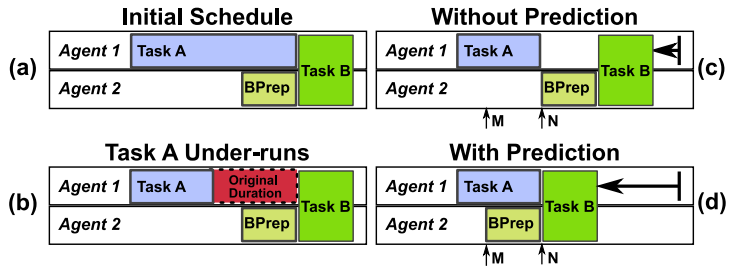


Figure 5.1: Duration prediction allows the planner to start setup tasks early when a preceding task is predicted to under-run.

picts a canonical example of an under-running task. Here, agent 1 performs the single-agent task A, after which agents 1 and 2 are scheduled to execute the multi-agent task B. The BPrep task is a setup task for task B, and must be performed immediately prior to B. The initial schedule is depicted in Fig. 5.1(a).

If task A completes early (Fig. 5.1(b)), BPrep and B may in turn be started early, reducing the overall makespan. If the planner does not predict this early completion, the only optimization available is to start BPrep immediately upon A's (early) completion (Fig. 5.1(c)). However, this is inefficient, as BPrep may be executed in parallel with A. If the planner were able to predict A's true completion time prior to point N, it would be able to start BPrep even earlier, realizing a further reduction in makespan. Ideally, the prediction would be made prior to point M, allowing B to be scheduled immediately after A, and BPrep to be executed entirely in parallel with task A (Fig. 5.1(d)). By providing the planner with forewarning of under-runs such as this, duration prediction enables the execution of more efficient schedules.

Note that since the planner is relying on a prediction, it is possible that A will under-run by less than the expected amount, delaying B and opening a gap between BPrep and B, if BPrep is started too early. The planner may leverage the duration distribution to minimize the likelihood of such an occurrence. For example, it may choose to use a duration for task A corresponding to a point further along its duration distribution. By setting A's duration such that there is, for instance, a 90% probability that A will complete at or before its scheduled end time, the planner ensures that there is only a 10% chance of the BPrep-B constraint being violated.

5.2.2 Over-runs

When an over-run is predicted, agents participating in now-delayed multi-agent tasks are able to fill the window with useful work, rather than idling until the slow

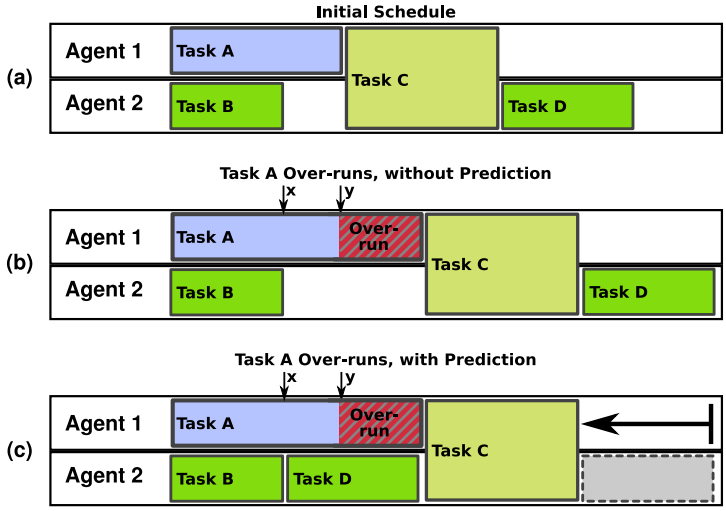


Figure 5.2: Predicting the remaining duration of executing tasks allows the planner to make use of opportunities presented by task over-runs.

task completes. If prediction were unavailable, there would be no way to know whether the over-running task would complete in the next second or in half an hour, and agents committed to the delayed multi-agent tasks would stand idle until the slow task completed, unable to perform any useful work in the meantime.

Fig. 5.2 depicts a canonical example of an over-running task. In the initial schedule (Fig. 5.2(a)), agents 1 and 2 perform individual tasks (A and B) prior to a group task (C). Suppose that task A over-runs, as presented in Fig. 5.2(b), with the over-run occurring at point Y. If duration prediction is not being performed, the planner will not realize in time that there is space to execute task D earlier than planned. However, if the planner is able to predict the over-run by point X, agent 2 will be able to execute task D earlier (Fig. 5.2(c)). This both reduces the makespan of this segment of the schedule and provides additional time for other tasks to be scheduled later on.

5.3 Use of Distributions

It may initially appear that the prediction of a distribution across duration, rather than a scalar estimate, provides little additional benefit. However, a duration distribution encodes significantly more useful information about the task in question that may be utilized in a number of ways, such as multi-metric optimization, rea-

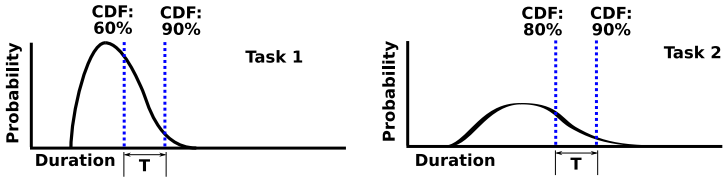


Figure 5.3: Reducing the available time for two tasks by T has different effects on the likelihood that each will finish within the reduced interval. This can be leveraged when simultaneously optimizing schedule makespan and the likelihood of tasks over-running their allotted time.

soning about deadlines, the efficient use of mutable teams, and prediction jitter compensation.

Duration distributions make certain forms of multi-metric optimization possible. Because the form of the duration distribution may vary greatly between tasks, reducing the time allocated to two tasks by the same amount will have different effects on the likelihood that each task will overrun its new scheduled time. For example, the duration distributions for two tasks are plotted in Fig. 5.3. Reducing the scheduled duration of them each by T has differing effects on the likelihood that they will complete within the scheduled time. While each initially has a 90% chance of doing so, the steeper form of task 1's distribution results in a 30% drop in the probability of timely completion, while task 2 only loses 10%. By maintaining duration distributions, the planner is able to reason in a principled fashion about the relationship between time and the likelihood of timely completion, allowing it to balance that likelihood against other metrics, such as makespan or total reward.

Predicting a distribution also increases the planner's ability to reason about deadlines. The form of duration distributions varies significantly by task, but is often multi-modal, with each mode reflecting a different number of failures (and recoveries) that occur during execution. The differentiation between modes will increase as the delay induced by each failure increases, leading to significant separation between peaks in the duration distribution. For instance, consider a task with a hard deadline, such as that diagrammed in Fig. 5.4(a). Assume that the planner has one additional agent available, and can use it to fill one of two optional roles. One reduces the delay associated with a failure (Fig. 5.4(b)) while the other increases the team's rate of progress (Figure 5.4(c)). As we can see, it is possible for the two options to result in identical means, while only one provides any chance of meeting the deadline. If only scalar predictions of duration were calculated, it would be impossible for the planner to distinguish between the relative usefulness

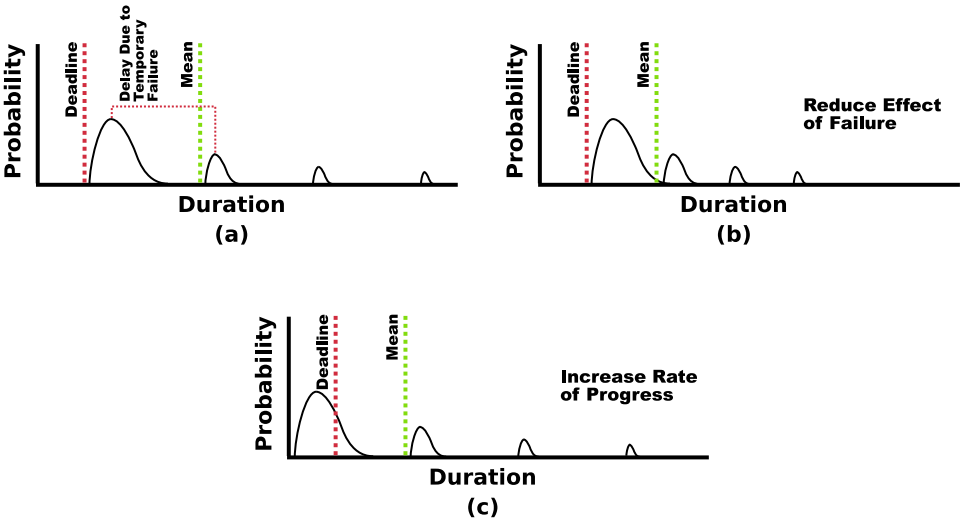


Figure 5.4: An illustration of the advantages of modeling multi-modal durations. (a) is the current team's duration distribution. The team has two unfilled optional roles: one reduces the effect of failures (b), and the other increases the rate of progress when operating normally (c). Note that the means of (b) and (c) are identical, but only (c) provides a non-zero probability of meeting the deadline. This distinction cannot be made using only the overall mean or a unimodal model of duration.

of the two roles.

The prediction of distributions is also useful in the context of *mutable teams*, which enable the transferring of agents between teams while tasks are executing. To realize the full benefits of such a transfer, we must be able to predict its effects. Since a physical agent cannot be moved instantaneously, there will be some uncertainty as to when it will join the receiving team, affecting the utility of the transfer. To evaluate the effect of a proposed live task modification in a principled manner, we must begin with distributions of the arrival time and the duration of the receiving task. Scalar estimates provide insufficient information to accurately reason about the utility of a live task modification. For instance, with a scalar duration estimate, it is impossible to predict the likelihood that the transferred agent will arrive in time to be useful.

Finally, duration distributions may be leveraged to overcome prediction jitter. The planner we are utilizing (ASPEN (Chien et al., 2000b)) schedules tasks with fixed durations. Since it was infeasible to extend ASPEN to reason explicitly about

duration distributions everywhere, we use the mean of the distribution, while maintaining the distribution itself to allow our heuristics to make use of it. Because the mean is used, and the training data is drawn from a stochastic process, the predicted duration of executing tasks on ASPEN's schedule will fluctuate ("jitter") as execution proceeds. This may result in superfluous conflicts with other tasks that would require significant time to repair. To ameliorate this problem, we update the predicted duration on ASPEN's schedule only when the prediction varies from the previous estimate by at least the standard deviation of the duration distribution.

5.4 Prediction Method

5.4.1 Kernel Density Estimation

We use a form of kernel density estimation (KDE) (Silverman, 1986) to predict duration distributions. As discussed in Section 3.3, KDE is a non-parametric method related to histograms that is used to estimate an arbitrary distribution from training data without making *a priori* assumptions about the form of the underlying distribution. By centering a kernel function, such as a normal distribution, at each observation, then summing the kernels, an estimate of the underlying distribution may be easily constructed. The sole parameter to KDE is the bandwidth of the kernel distributions, which controls the width of each kernel. For the normal distributions we utilize, the bandwidth is the standard deviation of the distribution.

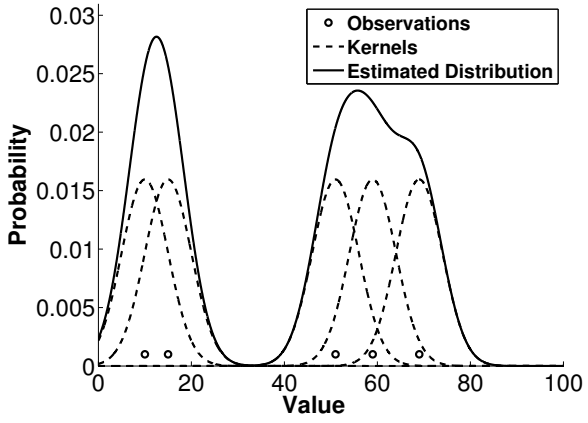


Figure 5.5: A simple example of kernel density estimation. Kernels (dashed lines) are centered at each of the five duration observations (plotted as 'o's), then the kernels are summed to build the estimated distribution (solid line). This figure is the same as Fig. 3.7.

Denote the bandwidth as h, the kernel function as $K(x, h)$, and let there be n observations with duration values x_i. The density of the distribution at a duration x is then the sum of the density contributed by the n kernels: $f(x) = \frac{1}{n}\sum_{i=1}^{n} K(x - x_i, h)$, where in our case $K(x, h) = \frac{1}{h\sqrt{2\pi}} exp\left(-\frac{x^2}{2h^2}\right)$ and $h = 2.5$.

5.4.2 Weighted Kernel Density Estimation

We use a weighted form of KDE in order to represent the belief that observations from points near the task's current state are more relevant. In this version of KDE, each observation is assigned a relative weight w_i, where $\sum_{i=1}^{n} w_i = 1$. The w_i are calculated by examining the distance in state space between the current state and each observation; details of these calculations are presented in the following section. The density function is nearly identical to the canonical KDE approach, simply replacing the uniform weighting with the observation-specific weight: $f(x) = \sum_{i=1}^{n} w_i K(x - x_i, h)$. This allows the contribution of individual observations to be adjusted; note that x in this equation is not the current task state, but the domain of the duration distribution (e.g. a point along the duration (value) axis in Fig. 5.5).

5.4.3 Application to Duration Prediction

Let us refer to the current state of the task whose duration is being predicted as the *query point*, a tuple Q of length d, where d is the dimensionality of the task's state space and Q_j is the current value of the jth state variable. Note that each observation in the training data is a tuple of length $d + 1$, consisting of the state and the duration observed there, and that Q_{d+1} is equivalent to x_i above. We must now determine the set of observations and associated weights that our KDE will use to build the duration distribution. We do so by applying a *query kernel* along each dimension of the state space (Fig. 5.6). A query kernel is a normal distribution centered at Q_j, with bandwidth h_j, that is used to calculate the weight of each observation for dimension j. The values of h_j are empirically selected, and depend upon the characteristics of the task. For instance, a continuous dimension may have a relatively large h_j, while a discrete dimension that represents a few very different cases may use a very small h_j to keep the cases segregated. If no points lie within the query kernel, the h_j are incrementally scaled up until data is found to support the query.

The weight in dimension j of the ith observation is simply the likelihood that $o_{i,j}$ (the observation's value for dimension j) would be randomly drawn from the

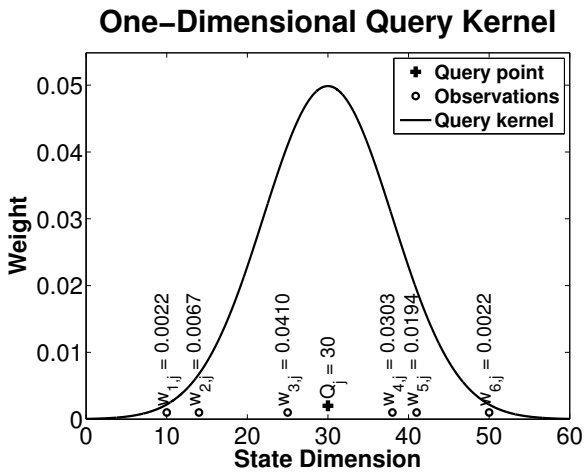


Figure 5.6: The query point (the current value of this dimension's state variable) is denoted with a '+', candidate observations with 'o's, and the query kernel as the solid curve. The weight of an observation i for this dimension, $w_{i,j}$, is the likelihood that the observation would be drawn randomly from the query kernel.

query kernel: $w_{i,j} = K\left(o_{i,j} - Q_j, h_j\right)$. To limit computation, we consider observations only where $w_{i,j} > \epsilon$, where ϵ is reduced if the query point lies in a sparse region of data, and the h_j were scaled up.

This weight calculation is performed for all dimensions, resulting in a set of weights $w_{i,j}$ for the observations. The final weight of an observation to be used for KDE is the normalized product of these per-dimension weights: $w_i = \frac{\prod_{j=1}^{d} w_{i,j}}{\sum_{i=1}^{n} \prod_{j=1}^{d} w_{i,j}}$. Once the per-observation weights are calculated, building the duration distribution is simply a matter of performing KDE as outlined above with the weights w_i and the observed durations $o_{i,d+1}$. The resulting distribution is used by our heuristics during schedule creation and repair.

We maintain our database of observations using k-d trees in order to allow rapid retrieval of observations near the query point.

5.4.4 Other Approaches

Our initial investigations into duration prediction focused on creating a predictor capable of interpolating and extrapolating across areas of low density data, while maintaining estimates of large state spaces in a spatially tractable manner. We focused on predicting either a scalar estimate of duration or a unimodal duration distribution. We eventually concluded that the need to model multi-modal distri-

butions outweighed the cost of maintaining the requisite training data sets, and moved to KDE for its flexibility, robustness, and speed. While our scalar and unimodal work is not applicable to the highly multi-modal domain we have chosen to explore, it may be of use to those working with domains in which the duration distributions are more properly modeled with a scalar or single mode. We present here a summary of the approaches we evaluated and the results of an experiment comparing their accuracy and computational cost. KDE is compared with these approaches, although a number of significant caveats apply.

Predicting a Scalar Estimate

Initially, we formulated duration prediction as a standard function approximation problem: given a set of observations, predict the expected duration of the task at its current state. As in our kernel density estimation approach, each observation consists of a state space vector and an observed remaining duration. The state space is multidimensional, and may contain discrete and/or continuous dimensions. We applied a variety of approximation methods to estimate duration from a point in the task's state space, given a set of training data. Among others, we evaluated linear regression, thin-plate splines, regression trees, neural nets, and locally weighted projection regression. We used each to construct an approximation of $D(x_1, x_2, \ldots, x_n) \rightarrow d$, where d is a scalar estimate of the task's remaining duration, given that the task is currently at $Q = (q_1, q_2, \ldots, q_n)$ in the state space. We define an observation as consisting of a state-space vector Q and an observed remaining duration y. For conciseness, we refer to the function being approximated as the *duration surface*: the surface resulting from plotting the predicted duration for every point in the task's state space.

Linear regression (Chatterjee and Hadi, 1986) consists of determining the least-squares fit to the training data by solving the linear model $y = Q\beta + \epsilon$, where β is a length-n vector of parameters and ϵ is a length-n vector of random disturbances. The members of ϵ are assumed to be drawn from a normal distribution with zero mean. This approach fits a hyperplane to the available data, which assumes that the duration of a task is linear as a function of its current position in state space. While this may hold in some domains, it does not in general, and the approach suffers as a result. In evaluating linear regression, we utilized MATLAB's `regress` implementation.

The thin-plate spline approach (Reinsch, 1967) simultaneously attempts to minimize the least-squares error and a measure of the fitted surface's roughness for two-dimensional state spaces. It requires a unique duration value for any given state vector: the algorithm cannot fit a one-to-many training data set. Due to the stochasticity of the domain, the underlying duration data is often in this form, with

multiple training examples with different durations occurring at a given state. This necessitates a preprocessing step to average the observed remaining duration for any points in the state space with multiple observations. The thin-plate smoothing spline, s, produced by this method minimizes $pE(s) + (1-p)R(s)$, where p is a smoothing parameter. As p varies from 0 to 1, the resulting spline varies from a least-squares approximation by a linear polynomial ($p = 0$) to a thin-plate spline that interpolates to all available data ($p = 1$). The first term is the least-squares error between observed and predicted durations: $E(s) = \sum_i |y_i - s(X_i)|^2$, where i varies across the training data set, X_i is the state space vector of a point in the training set, and y_i is the corresponding duration observation. The roughness measure, $R(s)$, is $\int \left(|D_1 D_1 s|^2 + 2|D_1 D_2 s|^2 + |D_2 D_2 s|^2\right)$, where $D_i s$ is the partial derivative of s with respect to its ith argument, and $|a|^2$ indicates the sum of squares of all entries of a. Thin-plate splines are able to model a wider variety of structures than linear regression, and to do so in a way that provides somewhat effective interpolation and extrapolation. However, they are restricted to two-dimensional state spaces, cannot model the discontinuities seen in some tasks' duration surfaces, and may take a very long time to fit with a high p and large amounts of training data. Thin-plate splines were evaluated using MATLAB's `tpaps` implementation, allowing it heuristically select p.

A regression tree (Breiman, 1993) is a binary decision tree built from a set of training data, with each branching based on an inequality defined on the value of one of the state dimensions. Its leaves are scalar-valued, and provide the tree's estimated remaining duration. Regression trees make no assumptions about the form of the underlying structure; may fit arbitrary functions, including those incorporating discontinuities; and are quite compact. However, they are subject to overfitting, and perform poorly when interpolating or extrapolating: the result is similar to predicting the nearest neighbor from the training set, with no actual interpolation or extrapolation occurring. Regression trees may be pruned post-construction based on an error/cost metric to reduce their size and avoid, to a degree, overfitting. We evaluated both pruned and unpruned regression trees, using MATLAB's `treefit` and `treeprune` implementations.

We also trained a two-layer, twenty-neuron feed-forward neural network to predict duration, based on a state-space vector input. The network used Radial Basis Function hidden-layer transfer functions and a linear output transfer function (Park and Sandberg, 1991). This network generated reasonable predictions, but consumed a significant amount of training time. We trained and evaluated the network using MATLAB's `newff` and `train` routines.

Finally, Locally Weighted Projection Regression (LWPR) (Vijayakumar and Schaal, 2000) (Vijayakumar et al., 2005) (Vijayakumar, 2001) constructs a set of locally linear kernels, spanned by a small number of univariate regressions. LWPR

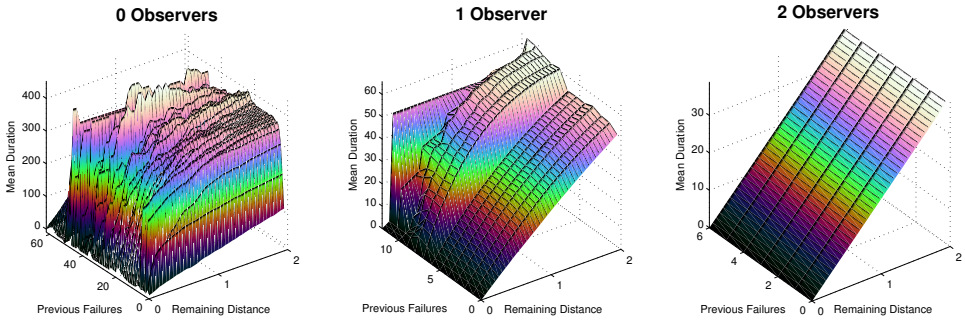


Figure 5.7: The duration surface of the *Place Panel* task. *Place Panel* has one required role with bounds $[1, 1]$ and one optional observer role, with bounds $[0, 2]$, and two continuous state variables.

is able to model nonlinear functions in high dimensional spaces with redundant and irrelevant dimensions. It requires large amounts of training data, and is able to incorporate new observations incrementally. The algorithm performed adequately. The quality of its fit initially improved rapidly as training data was added, but took a long time to plateau. We evaluated the MATLAB implementation available from the author of LWPR [1].

We used the the *Place Panel* task to compare these six approaches to predicting a scalar duration estimate. *Place Panel* has one required role with bounds $[1, 1]$ and one optional *observer* role, with bounds $[0, 2]$. Fig. 5.7 depicts the duration surface of the task. The smoothness of the duration surface varies with the number of observers, from linear (Fig. 5.7, right) to highly nonlinear (Fig. 5.7, left).

We collected 1000 example executions (traces) of *Place Panel*, totaling approximately 17,500 observations. Each approximation approach was then used to build a predicted duration surface with the first 10 traces. The resulting fits were compared with the ground truth (Fig. 5.7), and the point-wise mean squared error was calculated across the entire state space. This procedure was repeated, incrementally adding training data in sets of 10 traces. Table 5.1 presents a summary of the experiment. While no method was able to fit exactly the underlying surface, this is to be expected, given the nature of the 0-observer data. In the limit, thin-plate splines were able to provide the best fit, although since the algorithm cannot fit spaces with more than two dimensions, it was applied to each value of *observers* independently. LWPR learned quickly (Table 5.1, second column), and nearly matched the performance of thin-plate splines, but took somewhat longer

[1] Available at http://www.ipab.informatics.ed.ac.uk/slmc/software/lwpr/index.html, as of Fall 2008.

for its fit to stop improving (Table 5.1, fourth column).

Predicting a duration surface is insufficiently expressive for the domains in which we have explored proactive replanning, as knowledge about the distribution across duration proves to be valuable when making scheduling decisions. However, in a domain where a scalar estimate is sufficient, we recommend the use of LWPR. It is quite flexible, able to learn quickly, and able to interpolate and extrapolate between and beyond the available training data. Its final estimate was slightly less accurate than thin-plate splines; however, LWPR is not restricted to two-dimensional spaces. LWPR is also the slowest of the algorithms when queried: 24 thin-plate spline queries could be performed in the time needed for a single LWPR query.

We also evaluated kernel density estimation (KDE) in the same fashion, to provide an approximate comparison between the duration surface approaches and our final solution. We examined the means of the estimated duration distributions, in order to provide data in the same form as the original experiment. These results are reported in the final row of Table 5.1. However, a number of significant caveats apply. KDE solves a significantly different problem than the above approaches, focusing on generating a distribution using local data, rather than estimating the mean by fitting and extrapolating a surface. As a result, there is no learning stage: the sixth column in Table 5.1 for KDE is instead the time needed to load the data set and construct the necessary data structures. In addition, our KDE implementation is in C, while the duration surface methods were evaluated in Matlab. As a result, no strong conclusions may be drawn by comparing the KDE timing data to the remainder of the table. These results are provided solely to provide a degree of context.

Predicting a Parameterized Distribution

We also evaluated the possibility of fitting a parameterized distribution, such as the gamma distribution, to the training data. Specifically, we were interested in the possibility of fitting a set of surfaces, with each surface's domain being the task's state space and the range being a different distribution parameter. By discretizing the state space, then fitting individual distributions to the training data in each bin, we were able to transform the training data into a mapping from the task's state space to the parameters of the distribution in question. In theory, we then would have been able to apply one of the methods discussed in the previous section to fit a surface to each parameter, providing a compact representation able to generate duration distributions while interpolating across gaps in the training data. In practice, this proved problematic. Deviations in the parameters induced by the fitting procedures frequently combined to result in estimated distributions that varied significantly from the initial binned fits. We investigated several approaches, in-

Table 5.1: A comparison of the ability of various fitting algorithms to produce a scalar duration prediction for the *Place Panel* task, given a number of training traces varying from 10 to the full dataset of 1000 (approximately 17,500 data points).

Algorithm	**Mean Squared Error After 333 Training Traces**	**Minimum Mean Squared Error**	**Traces to Plateau**	**Learning Style**	**Time to Learn**	**Time to Query**
Linear Regression	5283.87	5501.02	300	batch	0.015s	5e-7s
Thin-Plate Spline	3620.13	247.55	650	batch	0.2s	5e-5s
Regression Tree	3591.57	331.50	650	batch	0.3s	1.5e-4s
Pruned Regression Tree	3822.83	1369.10	650	batch	4.0s	7.5e-6s
Neural Net	3570.16	369.94	650	batch	10.0s	1.3e-4s
Locally Weighted Projection Regression	2509.08	308.90	800	incremental	2.5s	1.2e-3s
Kernel Density Estimation[a]	7432.50	10.77	500	incremental	0.44s	3.3e-4s

[a] The KDE results differ from the duration surface approaches in a number of ways: the implementation is in C, instead of Matlab; there is no learning step ("Time to Learn" is the time to load the 1000-trace data set and construct the associated data structures); and KDE does not fit surfaces, leading to much tighter local fits, but less ability to extrapolate beyond the training data. This is the cause of KDE's high initial error and very low error once sufficient data becomes available.

cluding fitting the parameters independently and fitting them sequentially, using the prior fits to inform the latter, but were unable to produce satisfactory results. We abandoned this line of investigation when we determined that the underlying distributions were largely multi-modal, requiring a non-parametric function approximator. Before doing so, we evaluated the gamma, log-normal, normal, and Weibull distributions. All were able to form a reasonable approximation to a few unimodal example tasks, but clearly none can fit multi-modal distributions.

5.5 Planner Integration

We have integrated duration prediction with the repair-based ASPEN planner (Chien et al., 2000b) and CASPER executive (Chien et al., 2000a). Although ASPEN lacks explicit multi-agent support, its capabilities are a good fit to the needs of duration prediction during execution.

We have tightened the integration between planner and executive by providing a conduit for state updates to flow to the planner at every timestep. As the updates arrive, new duration predictions are triggered via ASPEN's Parameter Constraint Network (Fig. 5.8). If the mean of the resulting distribution differs from the duration of the task on the schedule by more than the standard deviation of the previous predicted distribution, ASPEN's schedule is updated. By delaying schedule updates, we eliminate much of the jitter that has affected our proactive replanning work in the past. We define *jitter* as small changes in the predicted duration distribution over time, resulting from the imprecise nature of training data, which in turn is due to stochasticity of the underlying task. Jitter is a problem because it may trigger unnecessary repair and replanning operations. By delaying the schedule update until the new prediction appears to be significantly different from the old, we eliminate much of the jitter, while retaining the ability to detect and react to significant changes in the predicted duration.

Once a schedule update occurs, ASPEN's usual mechanisms are able to repair any resulting conflicts or take advantage of optimization opportunities that have become apparent.

5.6 Predicting Resource Usage

While not the focus of this thesis, it is relatively straightforward to extend our duration prediction approach to predict resource usage instead. Rather than storing observations of task duration in the training data, instead observe the amount of a resource used during the task (e.g. 10 watt-hours of battery, 2.1 megabytes of memory, etc.). If the desired result is a distribution across the amount of resource

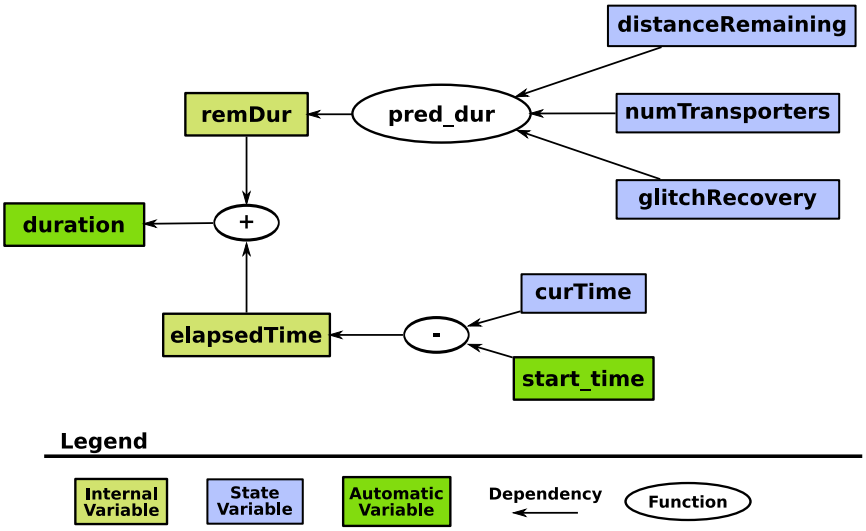


Figure 5.8: The *Transport* task's fragment of the Parameter Constraint Network (PCN). State variables are updated by the executive, and trigger propagations through the PCN, which result in the recalculation of duration predictions. When *duration* is updated, the task's duration on the schedule is changed. Here, the current time (*curTime*) is used to calculate elapsed time, which is added to the predicted remaining duration. The task state variables *distanceRemaining*, *numTransporters*, and *glitchRecovery* are updated by the executive: when they change, a new duration prediction is triggered via *pred_dur*.

that will be consumed during the remainder of the task, no modification to our method above is needed.

However, it also should be straightforward to extract a profile of the expected usage across time. Our data set maintains the links between successively observed points, allowing the traversal of individual training runs through the data set. If the stored resource data is instead interpreted as a delta usage over the next timestep, by taking the difference of consecutive observations, our prediction method yields a distribution across resource consumption in the next timestep. The points contributing to that distribution can then be projected forward in time to incorporate the future arrival of additional agents. A distribution can then be built at each step along this projection, yielding a stack of distributions across time. Depending on the need, the stack could either be used directly, or a resource usage profile could be extracted. For instance, if the mean and standard deviation of each distribution was calculated, plotting the cumulative sum of means would yield a plot of expected resource usage against time.

While the application of our prediction approach to the prediction of resource profiles appears straightforward, experimental exploration of this concept is outside the scope of this thesis.

5.7 Experimental Results

We have performed three experiments to evaluate the efficacy of live duration prediction. In the first, we examine how varying amounts of training data affect the accuracy of our duration predictions. We then, fix the size of the training corpus and compare a live duration prediction-enabled planning system with a baseline and an omniscient planner. The addition of live duration prediction yields 45% of the improvement achieved by the omniscient planner. Finally, we evaluate how live duration prediction affects the execution of a schedule, while again varying the amount of training data provided.

In these experiments, task execution is stochastically modeled using TaskSim, a representation similar to Augmented Transition Networks (Woods, 1970) and discussed in detail in Section 8.1.3. The models introduce a degree of uncertainty akin to that found in real-world robotic teams, and may be arbitrarily complex. In general, TaskSim models incorporate mid- to high-level task state, including non-terminal failures, and report task-level state to the CASPER executive. While individual components of agents (such as manipulators or sensors) are not simulated, events such as an agent becoming stuck in the sand and environmental conditions such as the slope of the terrain are represented. Fig. 5.9 depicts the model used for the *Move* task. Execution begins in the Moving state, and one state transition is made per time unit. During normal conditions, the distance traveled is incremented by a value drawn from a normal distribution, and there is a 1% chance of the agent becoming stuck during any given time step. Once the agent becomes stuck, it must recover before further progress can be made towards the goal. Models such as this are used to generate state traces during execution. Prior to execution, the models are used in the same fashion to generate the state traces that form the training database needed to predict duration distributions.

5.7.1 Accuracy of Prediction

Given an infinite amount of training data, our kernel density estimation approach will produce precisely the "true" duration distribution. However, training data is often difficult and expensive to collect. We have evaluated the accuracy of our duration prediction method (Section 5.4) as a function of the amount of available training data.

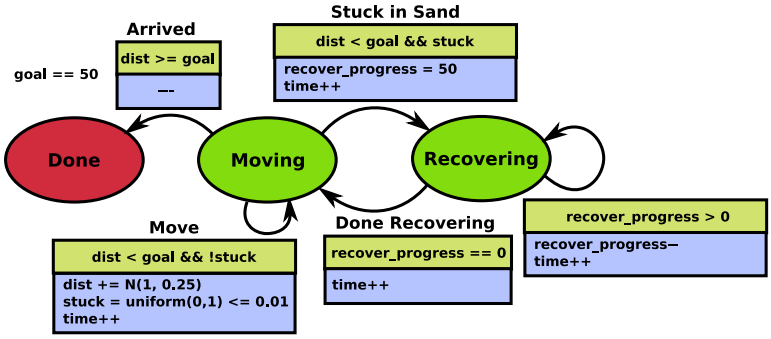


Figure 5.9: The stochastic simulation model for the *Move* task. This is used to simulate execution, and provides the state that is used for duration prediction during execution. In addition, the state traces generated are used offline to train the KDE-based predictor. Note that we assume a similar distribution of terrain during all moves, as the likelihood of becoming stuck during a given timestep is constant. The only modelled difference between moving from location A to B and A to C is the duration of the move, and hence the expected number of recoveries that must be performed.

Experimental Conditions

In order to characterize the accuracy of duration prediction, we built a corpus of data for each of eight tasks, consisting of 512 runs from the initial state. This yielded between 11,021 and 177,795 observations per task, depending on the task's inherent length and stochasticity. We then created 17 sample sets for each task, each with an ever larger portion of the corpus, beginning with 2 runs and proceeding to 512 runs in increments of 32.

We built a test duration distribution from each sample set at every point in a finely-spaced grid spanning the state space of each task, and compared them with the corresponding distribution constructed from the 512-run reference set. This comparison was made by computing the Kullback-Leibler (K-L) divergence (Kullback and Leibler, 1951) between the test and reference distributions. The divergence value increases as the distributions being compared become more different, while identical distributions have a divergence of zero.

Data and Discussion

Since a single execution run does not provide data spread evenly across the state space, plotting the accuracy of a prediction as a function of the number of runs in the training set is not useful. Instead, we plot the number of kernels (observations)

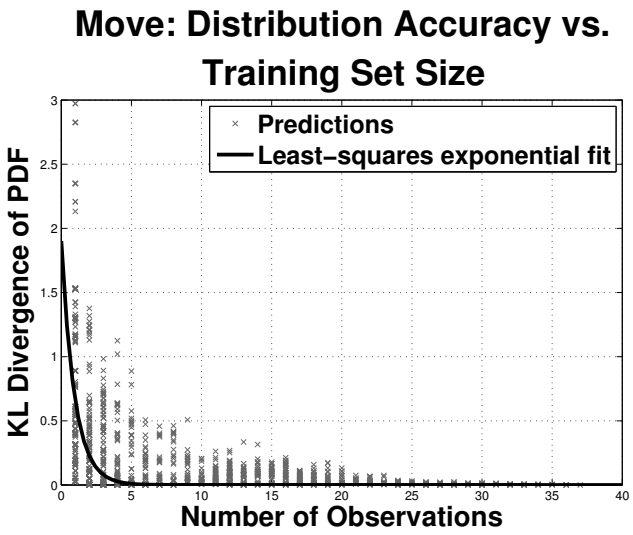


Figure 5.10: As the number of observations used to form the duration distribution increases, the resulting distribution becomes more accurate. The Y axis indicates the K-L divergence from a distribution built from all available data at the relevant query point.

used to construct a prediction against the resulting K-L divergence in Fig. 5.10, for all test points from all sample sets for the *Move* task. In general, test points from larger sample sets will have more observations, but the relationship depends heavily on the task's structure: a task may spend most of its time in one portion of the state space, yielding few observations in the outlying state space, even with large training sets. The fitted curve is the least-squares fit of $y = a * exp(b * x)$ to the available data. The exponential distribution also fits the data from the other seven tasks, although a and b will of course vary. As can be seen from Fig. 5.10, little utility is gained from having more than 5 data points in the vicinity of the query point for this particular task. Ideally, 5 observations would be within the query kernel bandwidth of every point in the state space. Given a kernel bandwidth and knowledge of the task's structure, this can provide guidance as to how many training runs are useful for a particular scenario.

5.7.2 Effects of Live Duration Prediction: Maximizing Reward

We have evaluated the effectiveness of duration prediction for proactive replanning by performing a series of experiments with CASPER and ASPEN, using TaskSim for execution simulation. The results reported here focus on the effects of duration

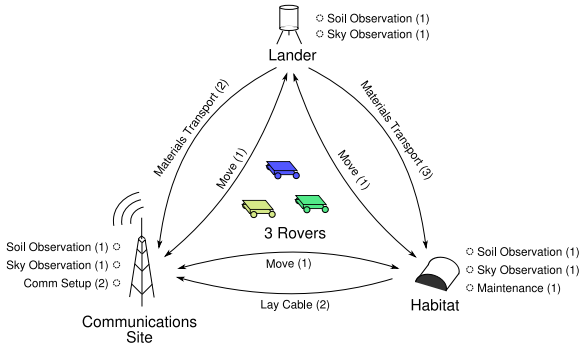


Figure 5.11: The Lunar Outpost scenario. Numbers in parentheses denote the number of agents needed to perform each task. Tasks during which the agent must remain at a site are denoted with a dashed circle.

prediction alone, and do not incorporate mutable teams or live task modification.

Scenario

The scenario represents a subset of the construction of a lunar outpost, and includes three agents, three sites, and eight types of tasks (Fig. 5.11, Table 5.2). Agents are assumed to be homogeneous, and can participate in only one task at a time. Each task has an associated reward, may be constrained to occur at a given site, and/or may involve the moving of agents from one site to another. Different tasks require different numbers of agents. In general, there are two classes of tasks: cooperative and solo. Cooperative tasks require more than one agent, and generally have higher rewards and durations than single-agent solo tasks.

The objective is to maximize total reward within a fixed horizon, which roughly equates to maximizing the number of reward-laden tasks that are executed. During execution, at every timestep ASPEN evaluates whether there are any conflicts in the schedule. If there are, it repairs the conflicts by first right-shifting tasks. If this fails, ASPEN's general repair algorithm is invoked until the schedule is conflict-free. After all conflicts have been repaired, ten iterations of optimization are performed, with repair of any new conflicts occurring after each. This allows the planner to repair any inefficiencies introduced by the initial repair cycle.

If there are no conflicts, ASPEN performs one iteration of a limited optimization algorithm at each timestep. In particular, it checks for agents that are free at the current time and have sufficient time prior to their next task to perform a solo task with non-zero reward. If such an agent can be found, a task is heuristically selected and scheduled for that agent.

Table 5.2: Lunar Outpost scenario tasks and relevant statistics.

Task	**Reward**	**Agents Needed**	**Location**	**State Variables**	**Progress Per Timestep**	**Duration from Start** (μ, σ)
Move	0	1	Start: Anywhere End: Anywhere	Distance moved: [0,50] Recovery progress: [0,50]	$N(1, 0.25)$	59.12 (39.59)
Sky Observation	15	1	Anywhere	Progress: [0,1]	$N(0.05, 0.01)$	18.16 (4.50)
Soil Observation	15	1	Anywhere	Progress: [0,1]	$N(0.025, 0.01)$	33.11 (7.21)
Habitat Maintenance	30	1	Habitat	Progress: [0,1] Recovery progress: [0,20]	$N(0.025, 0.01)$	41.65 (15.75)
Materials: Lander $\rightarrow$ Habitat	300	3	Start: Lander End: Habitat	Distance moved: [0,50] Recovery progress: [0,50]	$N(0.5, 0.1)$	113.72 (27.82)
Lay Cable	120	2	Start: Habitat End: Comm	Distance moved: [0,50] Recovery progress: [0,50]	$N(0.3, 0.25)$	205.53 (45.64)
Materials: Lander $\rightarrow$ Comm	100	2	Start: Lander End: Comm	Distance moved: [0,50] Recovery progress: [0,100]	$N(2, 0.25)$	34.06 (70.71)
Comm Setup	50	2	Comm	Progress: [0,1] Recovery progress: [0,200]	$N(0.05, 0.01)$	49.54 (83.80)

Experimental Conditions

We evaluated three experimental conditions: a baseline, an "oracle" case, and live duration prediction. We generated 20 initial schedules, each solving the same problem, using ASPEN, which we have augmented with heuristics that address multi-agent tasks. Duration prediction was used during the initial schedule generation to provide the (constant) durations for scheduled tasks. We executed each of the schedules five times under the baseline condition and fifty times under the live duration prediction condition, resulting in 100 and 1000 runs, respectively. In the oracle condition, the planner was provided with complete foreknowledge of the executed task durations, making execution extraneous. Instead, each of the 20 schedules was stochastically optimized five times, again yielding 100 data points.

Baseline

In the baseline case, no live duration prediction is performed. Instead, when a task finishes early, its duration on the schedule is instantly changed from the expected to the final value. When a task over-runs, its duration is increased incrementally, until it completes. If any conflicts are created, ASPEN's repair algorithm is invoked, followed by five iterations of the optimization algorithm. If no conflicts are present, one iteration of optimization is performed per executed timestep.

Live Duration Prediction

The live duration prediction condition is identical to the baseline, except that the durations of all currently executing tasks are updated at every timestep, as discussed in Section 5.5. When tasks are predicted to over-run or under-run, opportunities arise for the optimization algorithm to schedule additional tasks.

Oracle

In the oracle condition, the planner is provided with the precise final durations of each task, obviating any need for prediction. The specific durations were generated by simulating the task once for each task instance. As a result, the actual task durations were stochastic across runs, but the durations for each run were known by the planner at plan time. Under this condition, the planner updated an initial schedule (built using average task durations) with the actual durations, repaired any resulting conflicts, then performed 5000 iterations of the same optimization routines as used during execution in the other two experimental conditions. This was done five times for each of the 20 initial schedules.

Table 5.3: Live duration prediction experimental results.

	Reward [a]	**Executed Tasks** [a]	**Rewarded Executed Tasks** [a]	**Planning Time**	**Runs**
1. **Baseline**	2041.73 (475.53)	122.96 (28.26)	118.18 (31.24)	20.14s (7.79s)	100
2. **Oracle**	2571.10 (419.59)	131.10 (114.32)	114.32 (19.71)	66.16s (11.18s)	100
3. **Duration Prediction** [b]	2280.17 (433.25)	153.89 (27.73)	161.80 (32.22)	27.63s (17.57s)	1000
4. **Prediction - Baseline (i.e. PB)**	238.44 (+11.7%)	30.93 (+25.2%)	43.62 (+36.9%)	7.49 (+37.2%)	—
5. **Oracle - Baseline (i.e. OB)**	529.37 (+25.9%)	8.14 (+6.6%)	-3.86 (-3.3%)	46.02s (+228.5%)	—
6. **OB - PB** ($\frac{PB}{OB}$)	290.93 (45.0%)	-22.79 (380.0%)	-47.48 (—)	38.53s (16.3%)	—

[a] Values are the change of each metric between the initial schedule and the schedule at the completion of execution.

[b] Using 32 training runs per task.

Data and Discussion

The results of our experiment are detailed in Table 5.3, where the data are reported as *mean (standard deviation)* or *difference (percent difference)*. The data is the difference between the initial and final schedules – this removes the variation due to differences in the initial schedules. "Executed Tasks" includes zero-reward tasks, such as *Move*, while "Rewarded Executed Tasks" includes only rewarding tasks (see Table 5.2). The reported planning time is the time needed to perform all repairs and optimizations during the execution or construction of a schedule.

When using live duration prediction, the planner is able to schedule 25.2% more tasks on average than the baseline and achieve a 11.7% greater total reward (Table 5.3, row 4). Besides scheduling additional rewarding tasks, some unrewarding tasks were replaced with rewarding ones (the increase in rewarded tasks is greater than the increase in executed tasks).

Under the oracle condition, which represents the best that the planner could accomplish with the provided heuristics, 6.6% more tasks were scheduled than

the baseline, yielding 25.9% more reward (Table 5.3, row 5). This shows that by predicting task durations as execution proceeded, the planner was able to achieve 45.0% of the maximum possible improvement (Table 5.3, row 6). Note that under the oracle condition significantly fewer tasks were scheduled, yet more reward was garnered: with the provided foreknowledge, more of the long, high-reward tasks were scheduled. In contrast, when using duration prediction, many smaller low-reward tasks were added as small opportunities presented themselves. The specific results will vary according to the composition of the scenario, but the gains due to prediction are a function of the point at which over- and under-runs are predicted, the duration of tasks available for addition, and the reward per unit time of the available tasks.

While the average increases are promising, the data is quite noisy, as we can see from the large standard deviations. This is due to the stochastic nature of both execution and ASPEN's heuristic approach to repair and optimization. In particular, note the large standard deviations in expected duration for tasks such as *Comm Setup* and *Materials: Lander* $\rightarrow$ *Comm* (Table 5.2). We performed a repeated measures ANOVA with the initial schedule as the repeated sample to compare the baseline, duration prediction, and oracle results. There were statistically significant differences between all combinations (at a confidence level of $p = 0.0001$), except for rewarded tasks between the baseline and oracle cases. As can be seen from Table 5.3, the number of rewarded executed tasks was very similar between these conditions, although which tasks were executed varied significantly, as can be seen by the difference in reward.

These improvements come at a cost, however: overall planning time increases by 37.2% on average when using prediction (Table 5.3, row 4). This is due to an increased number of repair and optimization attempts made possible by the predictions, as well as the cost of prediction itself.

Effects of Varying Training Set Size

We also performed a series of simulated scenario executions while varying the amount of available training data from 2 to 128 runs per task. A run consists of the data points generated by a single simulated execution of the task. As can be seen in Fig. 5.12(a), reward increases as the amount of training data increases, asymptoting at roughly 32 runs of training data per task. While the data points for 32 and fewer runs of training data form a smooth curve, there is significant noise as the training set size is increased further. We do not currently have an explanation for this change. The experiment discussed in the previous section utilized data sets equivalent to the 32-run sample set. The prediction condition improves significantly over the baseline as long as more than 4 training runs are available: the

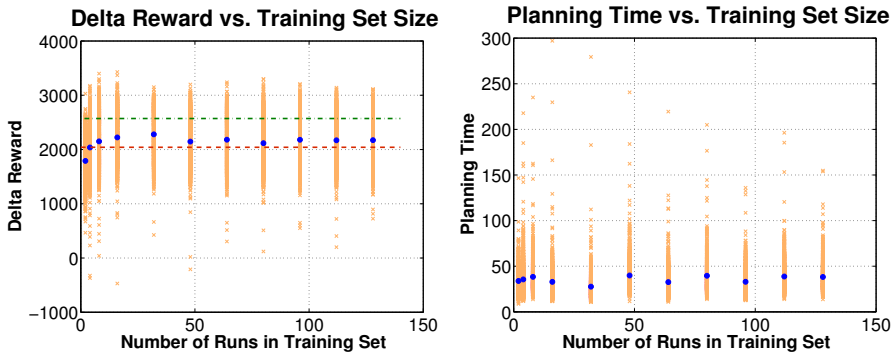


Figure 5.12: Reward achieved and planning time needed as functions of the number of runs in the training set for each task. Each point corresponds to an execution of a schedule, with the mean of each size of training set plotted as a square. In 5.12(a), the average delta reward for the baseline case is denoted with a dashed line, while the oracle condition is plotted as a dashed-dotted line.

difference between the baseline and every prediction experiment (except for the 4-run case) is statistically significant at a confidence level of 0.0001, according to a repeated measures ANOVA.

Fig. 5.12(b) shows that planning time increases very slightly, if at all, with the amount of training data. While we use k-d trees to quickly perform queries, prediction time will always increase as the size of the training corpus increases. This is offset by a reduction in the amount of repair needed, due to the more accurate predictions.

5.8 Summary

Live duration prediction is the repeated prediction of a task's remaining duration as execution proceeds. Even when used in isolation, it is a worthwhile addition to a planning and execution system, allowing achievement of 45% of the maximum possible improvement, when compared with an omniscient planner. It is useful in scenarios with scarce training data, while being able to improve its estimates as additional data becomes available.

Live duration prediction, while useful in its own right, is also a tool that can enable much greater improvements in planning and execution by providing the planner with the forewarning needed to appropriately adjust its schedule. The addition of mutable teams and live task modification provide the planner with additional tools with which to exploit the information provided by live duration prediction.

Chapter 6

Mutable Teams

6.1 Overview

Mutable teams are teams that agents may join or leave while the execution of a task is underway. By contrast, immutable teams are those in which any agents involved must participate for the entire duration of the task. The use of mutable teams approximates the ability of humans to jump into a task when others are having trouble, then leave when the problem is resolved. For example, the foreman on a factory floor often serves as a roving troubleshooter, lending assistance to any workers encountering difficulties, then moving on once the problem has been resolved. Humans excel at this type of fluid shifting between tasks: mutable teams extend some of the same capabilities to multi-agent robotic systems.

Our formulation of mutable teams is based on the concept of *roles*. We represent each task as having a set of roles, each of which requires, or may make use of, differing numbers of agents. We define a role as having a number of *role slots*, some *required*, and some *optional*. All required role slots must be filled for the task to proceed, while optional slots are not necessary, but may improve the task's speed, efficiency, or reliability.

For instance, a *Transport* task may have *transporter* and *scout* roles: agents filling a transporter role are responsible for moving the load, while scouts serve to actively find the best route for the transport team. It may be possible to move the component with a single agent, but an additional 1-2 agents in the transporter role may increase the team's speed and decrease the likelihood of becoming mired in treacherous terrain (the *transporter* role includes one required and two optional slots). It is possible to accomplish the *Transport* task with no scouts whatsoever, but if a robot is available for the *scout* role, it may reduce the probability of a terrain-related mishap by actively searching for problem areas (the *scout* role has a

single optional slot).

When describing a role, we specify the number of agents that may fill it with lower- and upper-bounds: the lower bound corresponds to the number of required slots in the role, while the difference between the bounds is the number of optional slots. For instance, the *transporter* role requires $[1, 3]$ agents, while the *scout* role requires $[0, 1]$ agents.

Note that roles are applicable to mutable and immutable teams: the difference lies in *when* agents may start or end their participation in a role. Immutable teams require that agents participate in their assigned role for the duration of the task, while mutable teams allow participating agents to come and go. With immutable teams, the planner has the flexibility to fill optional roles or leave them vacant, although if they are filled, a single agent must take on the entirety of a single role slot.

When discussing mutable teams, we refer to the schedule of agent arrivals and departures for a task as its *team profile*. This profile consists of a series of *team changes*, each of which is the arrival or departure of an agent at a specified time.

Allowing the planner to reason about mutable teams results in additional flexibility during repair or optimization of the overall schedule. For instance, idle windows in an agent's schedule may be utilized by temporarily assigning the agent to an optional role, allowing it to leave the team in time to perform its next scheduled task. Conflicts in a schedule due to an agent oversubscription may be resolved by allowing the oversubscribed agent to depart earlier than scheduled. Alternatively, additional agents may be added to a task to reduce its expected duration and either eliminate a conflict or reduce the schedule's makespan.

The addition of mutable teams requires extensions to our approach to duration prediction. With mutable teams, the planner has knowledge of future changes in the team's state (scheduled agent arrivals and departures) that should be taken into account when building duration distributions. We have developed two approaches to doing so: *distribution transfer functions* and *particle projection prediction*. We have evaluated their strengths and weaknesses experimentally, and report the results, showing that particle projection prediction is the proper choice for most contexts. In our experiments, agents are allowed to join or leave teams instantly, although they must spend time moving between teams. In tasks such as *Transport*, this is not realistic, as adding an agent to the *transporter* role will require the existing *transporter*s to reposition themselves to balance the load. We propose several approaches to estimating the effects of agent addition and removal on the team.

While mutable teams are useful in their own right, their effect is magnified when combined with other aspects of proactive replanning, such as live duration prediction and live task modification. For example, if two tasks are running in parallel, but one is progressing slower than expected, agents may be shifted from one

task to the other to balance the schedule. If a team runs into trouble during execution, additional resources may be added to ameliorate the problem. In addition, if a team is taking longer than planned, agents may depart the team prior to the completion of the task if they have other, higher priority, commitments. The interaction of mutable teams with other elements of proactive replanning is discussed further in Chapters 7 and 8.

We have experimentally evaluated the effects of mutable teams on the length of the optimized schedule generated by the planner prior to execution. On average, a statistically significant reduction in overall schedule length of 5.65% (33.04 minutes) was realized. The effects of mutable teams during schedule execution are examined in Chapter 8, where we determine that mutable teams are responsible for approximately half of the 11.5% reduction in makespan possible with a complete proactive replanning system.

6.2 Applicability

Mutable teams are beneficial at plan and repair time, even when used in the absence of the other aspects of proactive replanning. However, they are most effective when utilized in conjunction with live duration prediction and live task modification. Mutable teams allow the planner to more effectively utilize the agents at its disposal by avoiding many of the "holes" that often occur in multi-agent schedules. While this discussion is framed in terms of optimizing the schedule's makespan, the same techniques are of use when repairing conflicts. By reducing a task's duration or removing conflicting agents from a task early, the planner is able to resolve many conflicts with less schedule disruption than is otherwise possible.

The concept of mutable teams is applicable in a wide range of scenarios. If a task may be performed by a minimal team, but could be carried out in a more efficient, reliable, or cost effective manner with additional agents, mutable teams are applicable. Mutable teams are effective when applied to either type of multi-agent task: tasks that may be performed in parallel or cooperative tasks. In general, tasks that may be performed in parallel by many agents require little close interaction. Many mapping, inspection, and foraging activities fall into this category. Mutable teams are particularly useful in such domains because there is little cost involved in an agent joining or departing a team: the executive remains relatively simple, and the planner is free to shuffle agents as the scenario evolves.

In contrast, team members in cooperative tasks must closely interact to perform the desired action, such as coordinated manipulation. Joining or leaving a cooperative task may be a complex or slow procedure, as the remainder of the team may need to adjust. For instance, if a transporter agent leaves a *Transport* task,

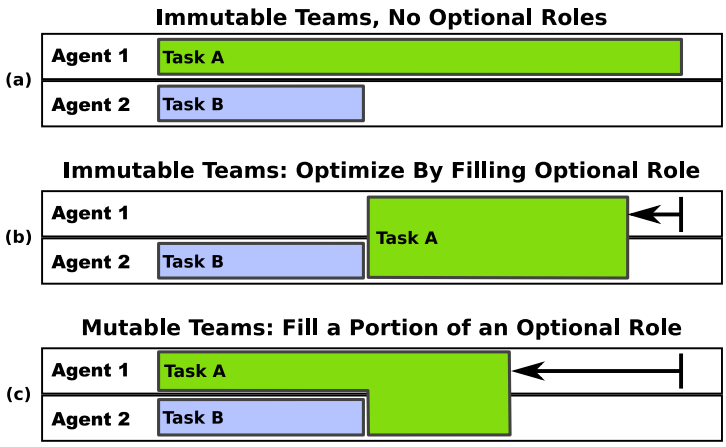


Figure 6.1: Mutable teams allow agents to join critical-path tasks once they finish their prior tasks. While the use of optional roles alone allows mild optimizations (b), mutable teams are needed to make full use of the opportunities afforded by optional roles (c).

the remaining transporters must shift to balance the load properly. Cooperative tasks may also include roles that may be filled or vacated with little effect on the remainder of the team, such as the scout role of the *Transport* task. These roles are often those involving non-contact sensing, advance scouting, or distributed computation. While the implementation of mutable teams for cooperative tasks may be complex, the resulting flexible, robust, and efficient planning and execution system makes the effort well worthwhile.

Fig. 6.1 is a canonical example of the use of mutable teams and an optional role to minimize the schedule's makespan. Here, two tasks must be performed: task A requires $[1, 2]$ agents, while task B requires a single agent. The schedule as formulated with immutable teams and no optional roles is shown in Fig. 6.1(a). With the addition of the optional role, a mild optimization may be performed by delaying the start of task A so that agent 2 may participate (Fig. 6.1(b)). However, by allowing mutable teams, agent 1 may make progress on task A while agent 2 completes task B, after which the agents join forces to rapidly complete task A (Fig. 6.1(c)).

Fig. 6.2 depicts the inverse: by allowing agent 1 to depart the team early, task B may be parallelized with the final stages of task A. While this increases the overall duration of task A, the schedule's makespan is reduced. The same approach may be used with live task modification to reduce the impact of an over-running task A: agents needed for subsequent tasks on the critical path may be removed from the

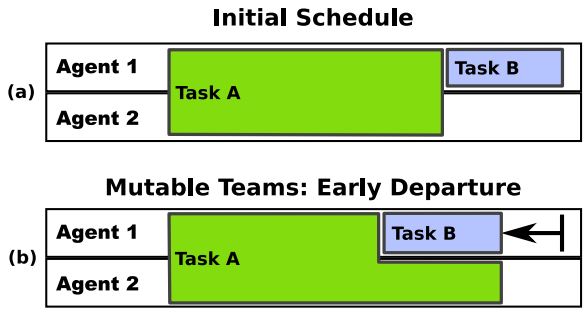


Figure 6.2: The use of mutable teams allows the planner to remove agents from a task early in order to reduce the schedule's overall makespan.

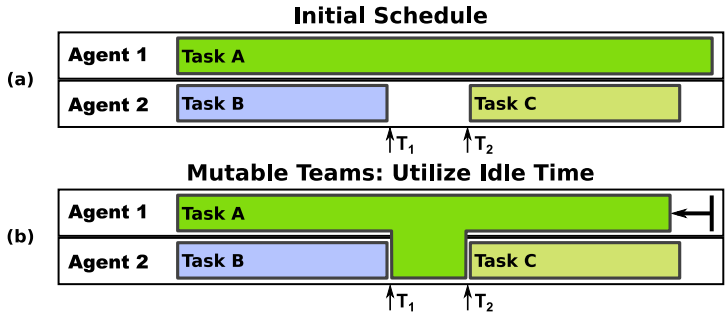


Figure 6.3: Mutable teams allow the planner to make use of otherwise idle portions of an agent's schedule. For illustrative purposes, assume tasks B and C are constrained to occur at their scheduled times.

team earlier than scheduled in order to avoid impacting the schedule's makespan.

Finally, the use of mutable teams allows the planner to make use of holes in an agent's schedule. In Fig. 6.3(a), agent 1 performs task A by itself, while agent 2 is scheduled to perform tasks B and C, with a span of idle time $[T_1, T_2]$ due to other constraints on B and C. With immutable teams, agent 2 would be unable to perform any useful work in $[T_1, T_2]$. When mutable teams are available, such small holes may often be utilized by temporarily filling an optional role, decreasing the schedule's makespan by a slight but measurable amount.

In addition to their uses at plan time, mutable teams make live task modification possible. Live task modification is the modification of an executing task's team profile, and allows the planner to adjust the profiles in response to the realities of execution. This gives the planner the ability to adjust for tasks that are operating more or less efficiently than expected by reallocating agents. See Section 7.2 for examples of the use of mutable teams in concert with live task modification.

6.3 Mutable Teams and the Planner

6.3.1 Required Planner Capabilities

To make effective use of mutable teams, the planner must be able to represent and reason about them. It must be possible to represent required and optional roles, as well as roles with both types of role slots. In addition, it must be possible to represent team profiles: the schedule must be expressive enough to encode agents arriving and leaving in the middle of a task. The representation must be flexible enough to allow rapid repair if an agent arrives later than scheduled. The planner must ensure that required role slots are filled at all times and optional roles are not oversubscribed. Due to these requirements, it is insufficient to simply record which agents are to be used by a particular task: instead, agent assignments must be durative, and represented such that different agents may fill the same role slot during non-overlapping times.

Reasoning about mutable teams is a complex problem that may be accomplished in a variety of ways. The degree to which the planner is able to effectively reason about them will limit the potential gains from adding mutable teams. The planner must enforce constraints between the mutable team task and agent assignments to ensure that the agent assignments are moved along with the task when the task is moved on the schedule, and that proper adjustments are made as the task's predicted duration changes. The planner should be able to efficiently locate tasks with (partially) unfilled required role slots, and be able to reason about how to fill them. Similarly, it should have the ability to quickly locate unfilled optional role slots, as they often represent optimization opportunities. To make full use of the potential of mutable teams, the planner should incorporate mutable teams into its repair mechanisms. For instance, filling an optional role slot often reduces the duration of a task, which may suffice to resolve a conflict with an overlapping task. Mutable teams may be leveraged in a similar fashion during optimization. Useful strategies include transferring agents between teams as an atomic operation and removing an agent from a team early if it is needed on a more critical task. See Section 8.2 for a discussion of the repair and optimization heuristics we have developed.

Representing and reasoning about mutable teams places a significant demand on the planner's capabilities. However, with the correct representation, much of the reasoning dovetails with the planner's existing repair and optimization strategies, allowing a straightforward extension to support mutable teams.

6.3.2 Representing Mutable Teams and Roles in ASPEN

While roles may be represented in a wide variety of ways, we have chosen to represent roles using a combination of two broad types of tasks (*activities* in the ASPEN literature) and metric resource timelines. This approach, which we term *agent request timelines*, allows us to reason naturally about the participation of each agent in a task, while rapidly detecting any open required or optional role slots.

Cooperative and Join Tasks

We represent a mutable team in ASPEN with two types of tasks: *cooperative* and *join*. A *cooperative* task represents the team as a whole, and serves as the source and sink for any temporal constraints related to the overall task. Tasks A and B in Fig. 6.4 are cooperative tasks. A *join* task represents the assignment of an agent for a period of time to a role slot of a particular task. We define a *join* task type for each combination of agent, role, and cooperative task type. For example, in Fig. 6.4(b), we define a task of type *Agent1-A-Required* (abbreviated *Ag1-A-Req*) to represent the participation of agent 1 in the required role of a cooperative task of type A. A separate task type is needed for each agent-role-cooperative task triple because ASPEN does not support the parameterization of reservations with respect to the timeline they are to be placed on. A join task must place a reservation on the lock timeline for the agent being represented, as well as the agent request timeline appropriate to the role and cooperative task with which it is associated. Because the timelines on which these reservations are to be placed cannot be parameterized, a multitude of join task types are necessary. Each join task maintains an internal pointer to its associated cooperative task (represented by the arrows in the Fig. 6.4(b)), to allow heuristics reasoning about the join tasks to determine which cooperative task the join is associated with.

By splitting a team task into cooperative and join components, we cleanly separate any constraints on the task as a whole from agent specific concerns, such as ensuring that an agent is not oversubscribed or in the wrong location to perform a task. This dichotomy allows the planner to reason about the join tasks separately when resolving conflicts or optimizing the schedule, subject to constraints between the join and its cooperative task. When scheduling a team task, our ASPEN heuristics first schedule the cooperative task, then incrementally add join tasks for various agents to fill its required and (possibly) optional roles. This allows the planner to consider iteratively which agents to add to the task, rather than forcing it to build the complete team profile the instant the cooperative task is added to the schedule.

As discussed in Section 3.1.3, we have extended ASPEN to support callbacks

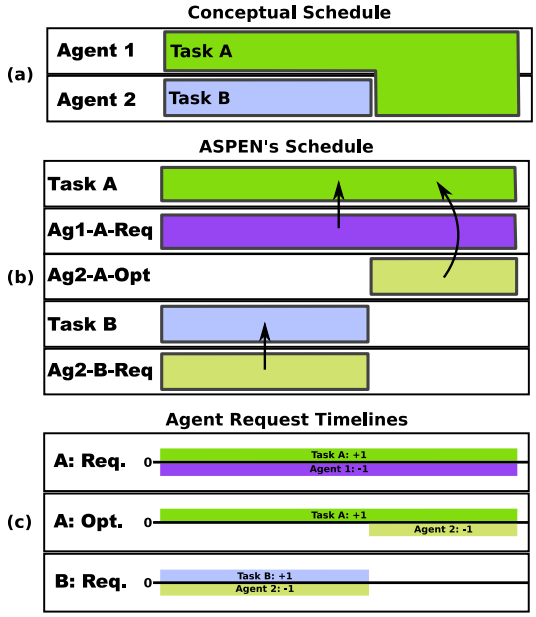


Figure 6.4: Mutable teams are encoded in ASPEN as a combination of *cooperative* and *join* tasks, as well as metric resource timelines serving as records of agent requests.

when various changes to the schedule are made. These were intended primarily to support the efficient maintenance of constraints between the cooperative and join tasks. For instance, when a cooperative task is moved, all associated join tasks are moved with it by the appropriate callback in order to maintain their positions relative to the cooperative task. While this could be represented by constraints in ASPEN's temporal constraint network, significant time would be required to repair the violated constraints whenever a cooperative task is moved. Instead, we maintain the following implicit constraints through callbacks that trigger when a cooperative task is modified:

1. If a cooperative task is moved, all join tasks also are moved, such that their placement relative to the cooperative task remains unchanged.

2. If a cooperative task's estimated duration shrinks, all join task durations are reduced so as not to extend past the end of the cooperative task. Any join task now scheduled to start after the end of the cooperative task is deleted.

3. If a cooperative task's estimated duration increases, the duration of all join tasks scheduled to end at the same time as the cooperative task will be in-

creased to match. An alternative strategy would be to not extend the join tasks, then repair any resulting conflicts by either extending an existing join or adding a new agent. We have taken the former approach, as it proved to be appropriate in the majority of situations. No flexibility is lost, as the planner is free to shorten a join task if its extension creates a conflict.

4. All join tasks of a deleted cooperative task will be deleted.

5. If a join task is deleted, the cooperative task's duration estimate is updated.

Roles as Resources

We encode the bounds on the agents that can be utilized in each role using ASPEN's metric resource timelines. These timelines track the value of an arbitrary resource over time, and may be constrained such that any value outside of a specified range indicates a conflict that must be repaired. Tasks may place reservations on the resource timelines that change the value of the timeline throughout the duration of the reservation.

We establish one or two resource timelines for each role/task pair, which we refer to as *agent request timelines*. If a role includes both required and optional slots (that is, if its agent bounds are $[M, N]$, $M > 0$ and $N > M$), two timelines are created: one for the required slots and one for the optional. If a role is purely required or optional, only the relevant timeline is created. This separation into required and optional resources allows the planner to treat any shortfalls on the required timeline as conflicts, while ignoring empty optional roles during plan repair.

When scheduled, the cooperative task places positive reservations on all associated agent request timelines with values equal to the number of agents required (or useful) in each role. An example of these reservations is diagrammed in Fig. 6.4(c): the blocks above 0 are reservations by the cooperative tasks. In this example, task A has a single role, with agent bounds of $[1, 2]$; task B also has a single role, with bounds of $[1, 1]$.

Join tasks place reservations with a value of -1 and a duration corresponding to that of the join task on the relevant agent request timeline. These offset the cooperative task's positive reservation, and balance at a value of 0 when enough agents are assigned to a given required role. Required agent request timelines are specified as having a valid value of 0: if at any point a required role is not filled, the timeline will have positive value, and ASPEN will place the conflict on its list to be repaired in the next plan repair cycle.

In contrast, optional agent request timelines do not need to balance for a schedule to be valid. Instead, the presence of a nonzero value indicates that an optional

role is open. Several of our optimization heuristics use this mechanism as a means for rapid identification of inefficient areas of the schedule.

Ideally, we would create a separate timeline for each role of each task instance, ensuring that reservations placed by two tasks of the same type never interact. Unfortunately, ASPEN does not support the dynamic creation of timelines, nor the parameterization of reservations with respect to the timeline they are placed on. As a result, we instead create one timeline for each role in each task *type*, not per task instance: multiple task instances may place reservations on a given timeline.

This representation increases the complexity of reasoning about open roles and reduces the utility of the agent request timelines. For instance, we must ensure that a role is never oversubscribed, even temporarily. If a task of type T requires one agent, tasks T_1 and T_2 are scheduled concurrently, 2 agents are (incorrectly) assigned to T_1, and none are assigned to T_2, the agent request timeline will not be in conflict, despite the oversubscription of T_1 and the undersubscription of T_2. We enforce this constraint through our callback mechanism by deleting any join task that creates an oversubscription condition shortly after it is added.

Because reservations from multiple tasks interact, we cannot directly infer which tasks have open required or optional roles from the timeline alone. Instead, we maintain explicit links between the join and cooperative tasks that allow our heuristics to rapidly construct the team profile for all tasks placing reservations upon a segment of a particular timeline. This additional computation negates some of the efficiency benefits provided by agent request timelines.

Complete Scheduling Example

We now present the a complete example of the addition of a cooperative task and its associated join tasks. The planner defines:

- One task instance T_j per cooperative task of type T to be performed
- One join task J_k of type $J(A_i, R, T)$ per agent A_i that is filling a role of type R in a task of type T. T_j maintains a mapping from role to associated join task(s), if any, and vice-versa.
- Two agent request timelines per role type for tasks of type T:
  - One for required role slots: $TL_{T,r}$
  - One for optional role slots: $TL_{T,o}$
- One agent lock timeline TL_{A_i} per agent A_i to ensure that agents are not oversubscribed

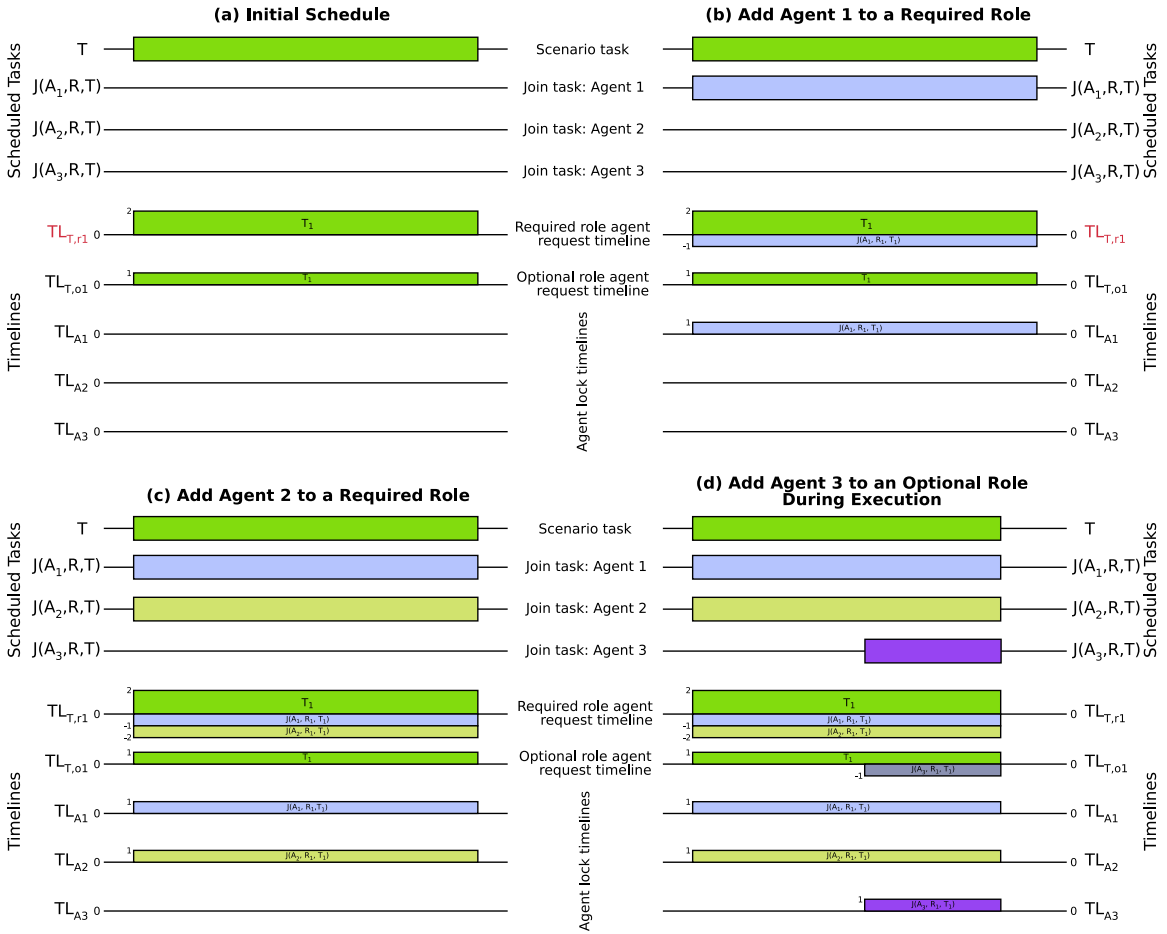


Figure 6.5: An example of scheduling a task with required and optional roles. Note that at the end of step (c), the schedule is valid (e.g. all required timelines ($TL_{T,r1}$) are balanced). Step (d) fills an optional role partway through the task, resulting in a shorter task duration.

Fig. 6.5 illustrates the steps involved in the planner adding a cooperative task to the schedule and filling its required and optional roles. The task being scheduled (T_1) includes one role, with agent bounds of $[2, 3]$. The planner schedules and optimizes a task of type T as follows:

1. Add a task T_j of type T to the schedule. This places a reservation on the associated required and optional agent request timelines (Fig. 6.5(a)).

2. The schedule is now invalid, since T requires at least two agents, unbalancing $TL_{T,r1}$ (Fig. 6.5(a)).

   Note that the timeline consists of increment/decrement segments associated with task instances, making it trivial to derive the list of tasks contributing to any given invalidity.

3. The planner iteratively selects a task that is contributing to the conflict and has unfilled required role slots, then adds an agent to it by adding a join task of type $J(A_i, R, T)$ (Fig. 6.5(b) and (c)).

4. Suppose the current task length over-runs a deadline, producing a conflict that must be repaired. One way to do so is to fill an optional role, reducing the task's duration, as is diagrammed in Fig. 6.5(d). By adding agent A_3 to a portion of T_j's optional role, T_j's duration is reduced, as are the durations of the other join tasks.

Repair continues until all required timelines are balanced. Note that additional conflicts may be introduced by adding an already occupied agent to the task; the planner's heuristics attempt to avoid such cases. If unavoidable, further repair will be triggered by the conflict on the agent lock timeline (TL_{A_i}). The order in which conflicts are repaired is subject to the design of the planner's heuristics[1].

6.3.3 Alternative Representations

Before selecting the representation discussed above, we investigated several alternatives, each of which has distinct advantages and disadvantages. Several variants on agent request timelines provide tradeoffs between different types of complexity. ASPEN's parameter constraint network could be leveraged to encode mutable teams, but at the cost of forgoing many of ASPEN's advantages. Finally, we

[1]Note that the predicted duration of the task is updated whenever the team assigned to a task changes. The expected duration is undefined when the task's required roles are not filled, and will be set to a default value (the expected duration under the assumption that all required roles and no optional roles are filled) to allow planning to proceed.

considered several approaches utilizing hierarchical decomposition, but they were insufficiently flexible to represent mutable teams.

Role Timeline Variants

The agent request timelines approach makes use of up to two timelines per role type per task type: one for any required slots in the role, and one for any optional slots. We considered two variants on this approach that modify the fashion in which timelines are used.

The first variant is to utilize a single timeline per role type per task type, combining all agents filling the role on a single timeline. This has the advantage of flexibility: an agent may be removed from the role, and as long as the task still has the required number of agents, a conflict will not result. Our current approach introduces an artificial distinction between the required and optional slots of a role in order to ease the detection of open required roles. At times, this results in unnecessary repair. One of our repair heuristics is designed to resolve conflicts resulting from a lack of required agents, by examining the assigned agents to determine if an agent in an optional slot may be moved to a required slot for the same role to resolve the conflict. The primary disadvantage to using a single timeline per role type is that it becomes impossible to use the mechanics of the timeline to detect timespans in which a required agent is missing: an overlapping task with a filled optional role slot may mask the lack of a required agent in separate task instance.

The second variant builds upon the first, by creating one timeline per role type per task *instance*, where the bounds on the timeline correspond exactly to the role bounds. This avoids the masking problem, at the cost of maintaining a large number of resource timelines. Representationally, this approach is superior to our current agent request timelines approach, as the timelines may be used to directly determine which tasks have open roles or over- or under-subscription conflicts. In addition, the artificial distinction between required and optional slots in a role is eliminated, removing one source of spurious conflicts. This is our preferred representation. However, ASPEN does not support the dynamic creation of timelines, so we selected the dual-timeline approach out of pragmatism.

Parameter Constraint Network

Another possibility is to maintain a single cooperative task on the schedule, while encoding all information about the task's roles and which agents are assigned to them in ASPEN's Parameter Constraint Network (PCN). The PCN is a network of constraints between tasks and their parameters that may be augmented by arbitrary user-supplied functions. It would be feasible to generate reservations against the

agent-specific timelines (e.g. position), eliminating the need for join tasks. Further, the bounds on each role could be directly encoded, rather than relying on a pair of timelines for each role type.

This approach is the most flexible of the options we considered, but its very flexibility is the source of many of its disadvantages. All conflicts raised in ASPEN as a result of problems with agent assignments to cooperative tasks would appear as unsatisfied dependencies, sidestepping much of ASPEN's ability to apply appropriate repair methods to each type of conflict. In our current implementation, all unsatisfied dependency conflicts may be resolved by propagating the PCN. Encoding role assignments in the PCN offloads the majority of the repair effort onto external code, negating many of ASPEN's advantages as a planner. In addition, to represent mutable teams, the PCN would need to be extended to support array types in order to cleanly store constraints between a cooperative task and a variable (technically unbounded) number of join tasks. Extending the PCN in this fashion would be a complex and error-prone undertaking.

Hierarchical Decomposition

Another class of representations hierarchically decompose the cooperative task, refining the task definition with each level of decomposition. While there are advantages to the two hierarchical representations we have examined, their disadvantages are significant. Most importantly, representing mutable teams is awkward at best, nullifying many of their advantages.

We have evaluated a decomposition in which each cooperative task decomposes at two levels: *role set* and *role assignment* (Fig. 6.6). The choice of role set determines which roles will be filled, while the role assignment level selects specific agents to fill the roles. This explicitly encodes the search tree that the planner must explore to find a viable assignment of agents to roles, and dovetails with ASPEN's ability to specify alternate decompositions for tasks. In domains with immutable teams and homogeneous agents, the choice of role set is sufficient to perform duration prediction, although this is not the case in our domains of interest. The role assignment level requires the planner to select the entire team at once, a more complex decision than the incremental approach to agent assignment we utilize. In addition, the branching factor is combinatorial in the number of agents and role types.

Another approach is to decompose the task into one task per role slot, then decompose each of these into specific agent assignments, as exemplified in Fig. 6.7. The task must be fully decomposed before duration prediction can be performed in any domain, but this approach allows the incremental selection of agents and reduces the branching factor. The only significant advantage of this alternative

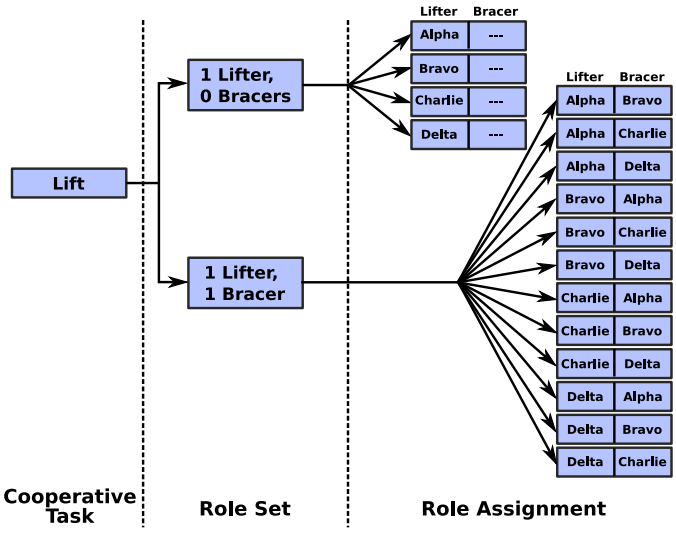


Figure 6.6: An alternative encoding for teams with optional roles is to hierarchically decompose the cooperative task into the set of filled roles, then the assignment of agents to the roles.

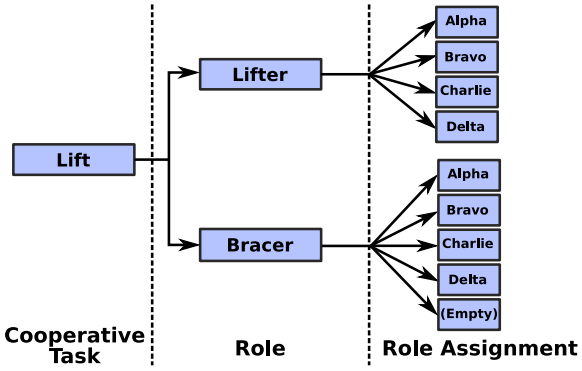


Figure 6.7: Instead of first decomposing the cooperative task into role sets, an alternative is to decompose it into a set of role slot tasks, each of which is then decomposed into a task assigning a specific agent (or no agent) to the slot.

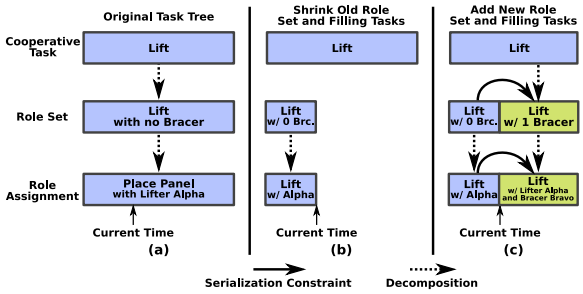


Figure 6.8: Representing mutable teams with a hierarchical decomposition approach requires the planner to break the decomposition link to the role set before re-decomposing the cooperative task into the new role set.

over agent request timelines is that the ties between the cooperative task and its children are explicit, instead of using internal join → cooperative links (curved arrows in Fig. 6.4).

However, all hierarchical decompositions share a significant flaw with respect to proactive replanning: representing mutable teams is awkward. While a decomposition tree naturally represents a particular team, there is no clean way to represent team changes, as ASPEN (and most other planners) assume a task can be decomposed in a only one fashion. Instead, as Fig. 6.8 illustrates, the planner must break the decomposition link between the cooperative task and the role set task (or set of role slot tasks), shorten the current role set and assignment tasks, then re-decompose the cooperative task into the new team. This creates many difficulties when planning for mutable teams that may involve a number of team changes throughout the execution of a task. Additional constraints must be introduced to ensure that the separated role sets maintain their assigned positions relative to the cooperative task, and the natural tie between cooperative and role set tasks, the primary advantage of the decomposition approach, disappears.

C-TAEMS

C-TAEMS (Wagner and Lesser (1999), Boddy et al. (2005)) is a hierarchical task representation used by many scheduling research groups, recently the Coordinators project (Smith et al. (2006), Musliner et al. (2006), Smith et al. (2007), Maheswaran and Szekely (2008)). C-TAEMS, and its predecessor TAEMS, are domain-independent task modeling frameworks that are used by a variety of planners and schedulers to represent and reason about complex problems. C-TAEMS describes tasks as having a variety of alternative approaches that the scheduler selects between, in order to optimize some set of criteria, often involving task duration and

quality (an abstract measure of the utility of a given approach to a task). Duration may be represented either as a scalar or a discrete distribution, usually with five or fewer potential durations. A parent task's duration is defined as the length of the time interval that contains all of its children, and is calculated prior to execution by combining its children's duration distributions.

Quality is used to manage which tasks from a set of potential tasks are chosen. Quality accumulation functions (QAFs) determine a parent's quality based on the quality of its children, and are used to represent how children should be selected. Examples include *max*, which functions as a logical *OR*; *exactly_one*, which provides quality if exactly one child is executed; and *sync_sum_and*, which provides quality only if all children begin execution simultaneously.

In addition to duration and quality, C-TAEMS provides for hard and soft constraints between arbitrary sets of tasks, such as enablement, facilitation, and hindering. Facilitation and hindering modify the quality or duration of the affected task in a positive or negative fashion, respectively. Facilitation and hindering constraints imply temporal ordering: the facilitating task must execute prior to the facilitated task.

C-TAEMS may be used to construct a hierarchical representation that partially captures the characteristics of a team with required and optional roles (Fig. 6.9). Each role slot is represented as having a set of alternatives corresponding to the different agents that are able to perform it, similar to the representation depicted in Fig. 6.7. A QAF of *exactly-one* ensures the selection of a single agent for each role slot, while a null task (with zero quality and duration) represents that the Bracer role may be left unfilled. A QAF of *sync_sum_and* ensures that both role slot tasks start simultaneously, and implicitly ensures that the roles are not filled by the same agent: in C-TAEMS, an agent may only perform a single task at a time.

Unfortunately, while C-TAEMS supports a flexible representation of duration, it does not provide a synchronous constraint allowing a task to modify the duration of another. Without such a mechanism, we cannot cleanly model the effect that filling an optional role may have on the length of the required roles. While we could enumerate the possible combinations of agents, such as in Fig. 6.6, the number of leaf tasks is combinatoric in the number of agents.

In addition, C-TAEMS shares the same flaw as the other hierarchical approaches discussed above: it is not possible to represent mutable teams naturally. Proactive replanning must not only be able to add or remove agents from a team, but be able to plan to do so in advance. The arbitrary nature of when agents may be scheduled to arrive or depart greatly increases the complexity of any representation that uses hierarchical tasks with defined sets of children, and results in unsatisfying solutions such as that presented in Fig. 6.8.

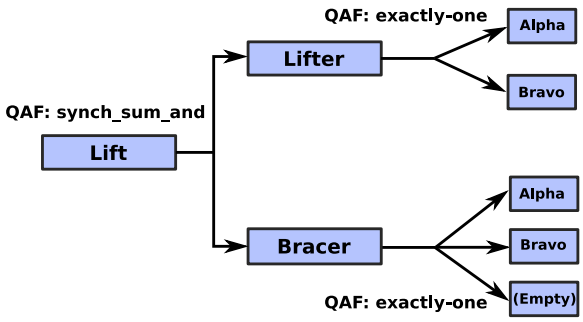


Figure 6.9: Optional roles may be partially represented in C-TAEMS through appropriate quality accumulation functions, the use of leaf tasks representing the commitment of an agent to a specific role slot, and empty tasks representing an open optional role slot.

6.4 Mutable Teams and the Executive

6.4.1 Required Executive Capabilities

Mutable teams require a much more complicated executive, primarily in the design of the tasks themselves. It must be possible to add or remove agents throughout the execution of a task: if live task modification is allowed, the team profile may change dynamically as execution proceeds. The planner also assumes that if mutable teams are allowed, it is possible to replace an agent with another in mid-task. This will range from easy to very difficult for the executive, depending on the type of task and the degree of coupling between the agents. Agents in non-contact roles, such as sensing, may be easily added or removed, while those performing tasks such as cooperative manipulation will require careful engagement and disengagement procedures to ensure the task does not fail during the addition or removal of an agent. We have considered many of the implications mutable teams have for execution, but have not fully explored these issues, as our experimentation has been performed in simulation. While significant effort is required to support mutable teams at the executive level, the nature of optional roles and mutable teams provides the task designer with the opportunity to construct tasks that are more flexible, fault-tolerant, and scalable than ones designed for immutable teams with no optional roles.

6.4.2 Designing Tasks to Utilize Optional Roles

Optional and required roles allow the designer to specify the minimum set of agents that may be able to accomplish the task (i.e. the set of required role slots), albeit slowly and in an unreliable fashion. If optional roles are not used, and a greater degree of reliability is desired, the designer must require additional agents beyond the minimum. This reduces the degree to which tasks may be executed in parallel, and prevents a minimal team from attempting the task, even if sufficient time is available for the team to (slowly) complete the task. Optional roles allow the designer to specify the task's *minimum* acceptable performance, rather than the desired *nominal* performance, increasing the team's flexibility and applicability.

Optional and required roles allow the task designer to explicitly specify what additional agents may be added to the team, and how they improve the team's performance. This increases the flexibility of the system, as the planner is able to allocate more or fewer agents as the situation warrants: if a task is far from the critical path, the optimal approach may be to attempt it with the minimum set of agents, reserving the remainder for more critical tasks.

The converse also is true: the task designer is free to specify the full set of optional agents that may improve the task, without regard to the needs of other tasks: optional roles shift the selection of the optimal team composition from design-time to execution-time. When working without optional roles, the designer must select a single point in the range of possible team configurations, balancing the desire for improved performance against the need to ensure other tasks may operate in parallel. Optional roles free the designer from this constraint, allowing him to instead specify the maximum set of useful agents, secure in the knowledge that the additional agents will be utilized only if they are not needed elsewhere. Roles that would have been discarded as not absolutely necessary may be included, and will be filled only as appropriate to the overall schedule. If the resources are available, the planner will allocate them appropriately among the active tasks, improving performance beyond the nominal case that is encoded without optional roles. As additional agents become available, the teams will incorporate them, improving performance without requiring entirely new tasks to be developed for the larger set of agents.

Teams incorporating optional roles are more fault-tolerant, scalable, and flexible. The planner is able to make use of optional roles to adapt the team to the situation it faces, rather than relying on the one-size-fits-all approach common to teams lacking optional roles.

Optional roles may be added gradually to an existing system: tasks initially may be encoded as having only the minimal set of required roles, with optional roles being developed incrementally. For instance, optional roles may be added to

ameliorate problematic cases that are discovered as the development and testing cycle proceeds. Common types of optional roles include non-contact sensing (e.g. scouting or redundant sensing), load-sharing (e.g. assisting with the transport of a heavy object), and failure prevention (e.g. bracing the other end of a connection being made).

When considering the encoding of a task, those role slots that absolutely must be filled for the task to complete should be specified as required. Once this base set of required role slots is identified, the designer should consider how additional agents could be used to increase the team's rate of progress, reduce the likelihood of a failure, reduce the impact of any failure, reduce the consumption of depletable resources, or reduce overall component wear. Optional roles should be added to the task as appropriate, building up from those that are easy to integrate (e.g. scouting) to those with more complex maneuvers required during agent addition and removal. Almost every task may be augmented with optional roles in some fashion: even small improvements are worthwhile, as the planner will fill the role only if its effect is useful in the scenario that obtains. Table 6.1 summarizes nine generic optional roles, and their potential effects on a task.

6.4.3 Designing Tasks to Utilize Mutable Teams

Optional roles provide significant benefits, while mutable teams magnify them. By designing a task to support mutable teams, the designer improves the planner's ability to fluidly reassign resources as the developing scenario warrants and to react to problems by adding the appropriate agents to teams that are under-performing. Mutable teams ensure that agents will be utilized efficiently, reducing their idle time by allowing them to assist a team just until their next scheduled task begins. While optional roles are useful in isolation, their full potential is realized only when used in combination with mutable teams.

The most challenging aspect of mutable teams that confronts the designer is the support of agent addition and removal in mid-task (*team changes*). The difficulty of a team change varies widely with the task and role. For instance, adding or removing sensing agents generally will not require a reconfiguration of any other agents in the team, while adding or removing a cooperative manipulation agent will involve complex maneuvers to safely redistribute the team's load. When considering whether to add a particular optional role, the costs of development can be weighed against the potential gains. Many optional roles are of the "non-contact" variety, as they are easier to implement, and have less integration overhead. However, "contact" optional roles should not be discarded out of hand: they often provide improvements in the task significant enough to warrant the development effort.

The executive may also make simplifying assumptions to achieve a compro-

Table 6.1: Tasks may be augmented with a wide variety of optional roles, with an equally wide range of effects. A few examples are detailed here. Effects noted in each example indicate the likelihood of improvement in the relevant category. Integration drag is the degree by which the team is slowed while the new agent joins the team, and is on a scale of 1 (no effect) to 5 (progress halts while the agent joins).

⇓ Role \ Effects ⇒	Integration Drag	Speed	Reliability[a]	Failure Mitigation[b]	Quality	Reduce Resource Usage	Reduce Wear & Tear
Scouting	1		X			X	X
Redundant sensing	2		X		X		
Support weight of beam to allow finer placement during docking	5	X	X		X		X
Share load of transported object	5	X	X	X		X	X
Tow a mired team out of sand	5	X		X			X
Brace a post while it is being raised	3		X	X		X	X
Map a subregion	1	X	X		X	X	X
Insert and tighten a fraction of a large set of bolts	2	X					
Provide additional processing node for data compression	1	X					
Monitor team to improve error detection and prevention	1		X	X			

[a] A role that improves reliability reduces the likelihood of a failure occurring.

[b] A role that mitigates a failure reduces the failure's impact, but does not necessarily reduce the likelihood of the failure.

mise between the challenges of implementation and the flexibility afforded to the planner. While the planner assumes that agents may be added or removed regardless of the task's state, the executive may need to simplify the problem by allowing team changes for specified roles to occur only at specific "interchange" states, such as positions in which a transported load is statically stable. It may be easier for the executive to redirect joining agents to the next interchange state and allow them to join later than the planner intended, rather than building an extremely complex task that allows agents to join in any state. If the training data used by the planner to form duration predictions includes examples of the use of these interchange points, the planner will implicitly include the effects of the interchanges into its reasoning.

6.5 Duration Prediction with Mutable Teams

The addition of mutable teams complicates the problem of duration prediction: the Markov assumption no longer holds, as the planner now has knowledge of future changes in the task's state. This knowledge of future agent arrivals and departures, and the need to reason about the benefits of these *team changes*, expands the complexity of duration prediction in two ways. The prediction algorithms must be able to take into account (1) the effects of the current team profile and any proposed team changes, as well as (2) any detrimental effects the act of integrating or disengaging an agent may have on the team. We have developed two methods to estimate the effects of the team profile: *distribution transfer functions* and *particle projection prediction*. We characterize the strengths and weaknesses of each, and utilize particle projection prediction in the remainder of this thesis. We have also examined a series of approaches to approximately model how the integration and disengagement of agents may affect a team. These are surveyed below, although none are yet part of the currently implemented system.

6.5.1 Predicting the Effects of Team Changes

While the application of kernel density estimation as discussed in Section 5.4 works well when utilizing only the duration prediction aspect of proactive replanning, mutable teams increase the complexity of the problem. With mutable teams, agents may join or leave the team during a task's execution, resulting in fluctuations in how many agents are filling each role of the task. The approach discussed in Section 5.4 works well with immutable teams, or if the changes in the profile closely parallel those in the training data. It is also applicable if there are a discrete (and small) number of states in which agents may join or leave the team in a well-defined fashion and sufficient training data is available to cover the possible team

profiles.

However, if agents may join or depart the team at any time, the number of potential team profiles explodes, making the collection of sufficient training data infeasible, even for small tasks. To avoid this problem, we have factored out the team profile from our task state. All training data is collected with immutable teams, the number of agents in each role is included in the task state, and we infer the effects of a given team profile.

The problem we address here is the prediction of the duration distribution for a team P at a particular task state S, given a series of team profile changes $T = ((t_0, p_0), (t_1, p_1)...(t_n, p_n))$, where t_i is how far in the future the change will occur, p_i is a vector of deltas to be applied to the number of agents filling each role, t_0 is the current time, and p_0 is the null-change vector. The fundamental difficulty is that the state of the task is unknown at each t_i, due to the uncertain nature of execution. This prevents us from directly applying our existing approach to prediction to each segment of the task. Our methods of incorporating team profiles into duration prediction take two very different approaches to estimating the task state at each of the t_i.

We have developed two methods to estimate the duration distribution for mutable-team tasks: *distribution transfer functions* and *particle projection prediction*. Transfer functions are relatively efficient to compute, and can be quite accurate, but only if a number of restrictive assumptions hold. Particle projection is more computationally expensive, but is not subject to the limitations of the transfer function approach. Both will be discussed here, but the remainder of the work utilizes particle projection prediction when mutable teams are enabled.

Distribution Transfer Functions

Distribution transfer functions make the simplifying assumption that the form of a task's duration distribution remains constant throughout execution for a given team. That is, we assume here that as execution proceeds with a fixed team, the distribution across remaining duration is translated steadily towards zero, with no or little change in the number of modes or their shape. This is a severely limiting approximation, as it assumes any execution-time events will only shift the distribution, rather than change its form.

Under this assumption, we can build duration distributions from the task's state at the time the prediction is being computed for each of the different teams that occur in the profile by applying the approach discussed in Section 5.4. This yields a library of distributions that we assume to be canonical for each team. We then compute a *transfer function* between the distributions for the teams at time t_i and t_{i+1}, for all i. The transfer function transforms one distribution into the other, and

is an approximation of the effect that the team change p_{i+1} has on the duration distribution (see below for a complete explanation of the construction of a transfer function). Starting with the canonical distribution for the currently assigned team, we are able to shift the distribution towards zero by $(t_1 - t_0)$ to approximate the effect of the passage of time, apply the transfer function between the canonical distributions for teams P and $P + p_1$, and repeat for all t_i. We shift the final distribution away from zero by t_n (the time of the last profile change), to make it relative to the current time, t_0. This yields a duration distribution that incorporates an approximation of the effects of the team profile.

The transfer function itself is a method of transforming a source distribution A into a destination distribution B. Once computed, the transfer function may be used to map new distributions similar to A into a form similar to B. To construct a transfer function, correspondences first are established between points on the probability density functions (PDFs) of A and B. This is done by sampling a series of probability values from the distributions' cumulative distribution functions (CDFs), then pairing the associated points on their PDFs. For instance, in Fig. 6.10 the points at dA_1 and dB_1 correspond to a CDF probability of 0.2 in their respective distributions. When applying the transfer function, the point on A's PDF at dA_1 should be mapped to dB_1. Only three correspondences are illustrated in Fig. 6.10 for clarity; in practice, many more points are sampled.

Once the correspondences have been established, we may calculate the transformation to be applied to each point on an input distribution in order to transform it from the domain of A to the domain of B. A point on A's PDF may be described with two values: the duration and probability: (pA_i, dA_i). A correct transfer function will exactly reproduce B, if A is presented as the input. Each point is thus mapped by multiplying by the ratio of A and B's probability and duration values: $(pA'_i, dA'_i) = \left(pA_i * \frac{pB_i}{pA_i}, dA_i * \frac{dB_i}{dA_i}\right)$

Given an input distribution I of a form similar to A, we first find the points on I's PDF corresponding to the CDF probability values sampled during the creation of the transfer function: $(pI_1, dI_1)...(pI_n, dI_n)$. Each point on I's PDF is then multiplied by the corresponding A-B ratio: $(pI'_i, dI'_i) = \left(pI_i * \frac{pB_i}{pA_i}, dI_i * \frac{dB_i}{dA_i}\right)$. A cubic splines is then fit to the resulting set of points $(pI'_1, dI'_1)...(pI'_n, dI'_n)$ and is used to interpolate all values of the transformed PDF. Algorithms A.1 and A.2 detail the steps involved in computing and applying distribution transfer functions.

Note that the assumption that a duration distribution has an unchanging form implies that the number of modes of the distribution will remain unchanged over the time span in question. Unfortunately, in our domains this does not hold: outlying modes tend to disappear as the task approaches completion or failures occur, as is illustrated in the experiments discussed in Section 6.6.1. This shortcoming drove

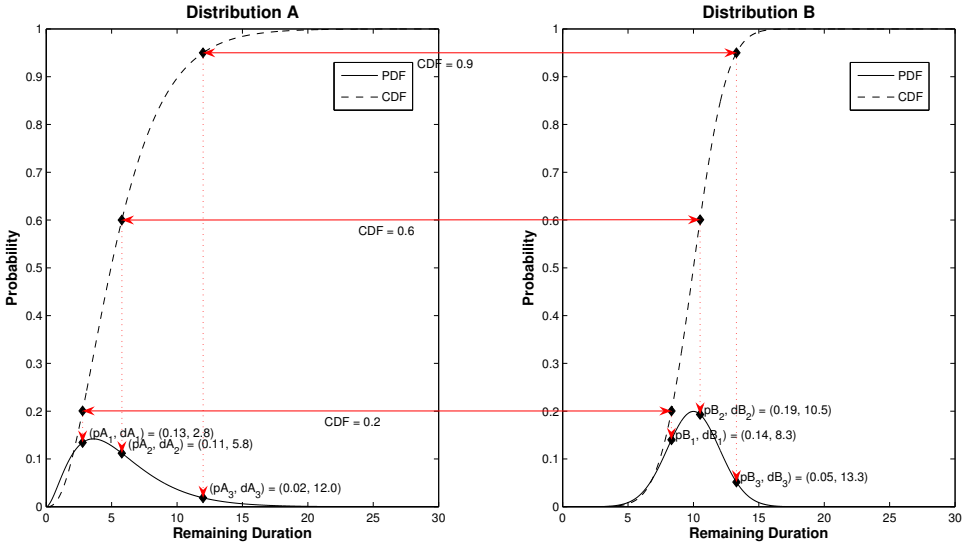


Figure 6.10: The transfer function between a gamma and a normal distribution is computed by calculating the ratios between the PDFs' probability and duration values at points where the CDF values are equal.

our development of particle projection prediction, discussed below. However, in domains where this assumption holds, transfer functions may prove attractive due to their low computational cost.

Particle Projection Prediction

In particle projection prediction, we take a more low-level approach. As in distribution transfer functions, the problem remains the prediction of the duration distribution for a team at a particular state S, given a series of team profile changes $T = ((t_1, p_1)...(t_n, p_n))$, where t_i is how far in the future the change will occur and p_i is a vector of deltas to be applied to the number of agents filling each role. Within the training database, we maintain links between successive observations of a given training run, allowing us to traverse forwards in time along the run associated with a particular point. Particle projection prediction first builds a query around the current task and team state. It then uses the observation links to project each observation supporting the query forwards in time to t_1. p_1 is applied to each resulting task state, and a query is performed centered at each of the modified states. We then subsample from the resulting set of points to negate the branching factor and repeat for $i = 2...n$. In essence, we build an implicit distribution across the task state at each of the t_i. The duration distribution formed at t_n is then shifted

away from zero by t_n to form our final estimate.

To provide concreteness, Fig. 6.11 will be referenced throughout this discussion. It plots a training database consisting of four runs of a task with two state variables: the number of agents assigned and the distance remaining to the task's goal. The range of the database is, as before, the remaining duration. In this example, the goal is to estimate the duration distribution for a task starting with one agent and 9.5 units from the goal, given that a second agent will join the team in 2 time units. Fig. 6.11 flattens the agent dimension, using dashed lines to denote single agent data or queries and dotted lines for the two-agent state. As distance remaining is the only state variable in the plot, query kernels are denoted as horizontal rectangles spanning a range of distances. The agent coordinate of the query kernel is denoted by the line type of the rectangle.

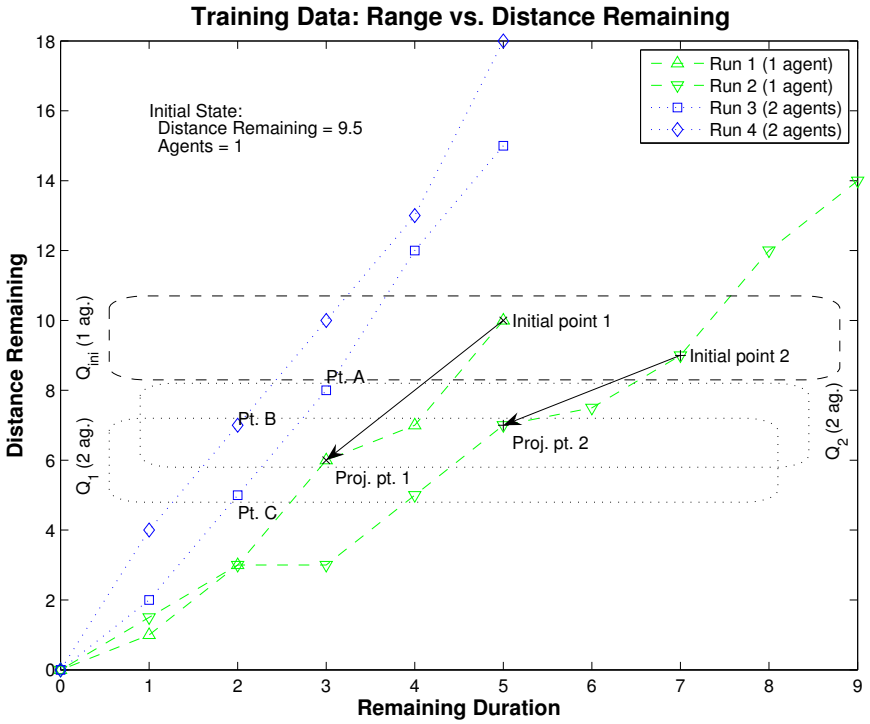


Figure 6.11: In this example of particle projection prediction, the task starts with 1 agent and 9.5 distance units from the goal. An additional agent will arrive in 2 time units. This diagram flattens the agents dimension of the state space, depicting data and query kernels for one agent as dashed lines and two-agent elements with dotted lines.

Given the initial state S, we first apply a query kernel around it, as in standard

duration prediction (Section 5.4, Q_{ini} in Fig. 6.11). This yields a set of N weighted points (*Initial point 1* and *2* in Fig. 6.11). We then trace forward along the training runs associated with these points t_1 time units. In Fig. 6.11, this consists of traversing two time units to the left, as depicted by the arrows, arriving at points *Proj. pt. 1* and *2*. The resulting set of projected points are the set of states that have been reached in the past in t_1 time units from the training points near our current state. We then modify the state of each projected point by p_1; in Fig. 6.11, this entails adding an additional agent. Note that the resulting states are no longer necessarily points in the training database: there are no observations from runs 3 or 4 with the same distance remaining value as *Proj. pt. 1* or *2*.

We then apply a query kernel around each of the projected, modified states (Q_1 and Q_2 in Fig. 6.11), resulting in N sets of weighted points. In Fig. 6.11, these sets consist of $\{B, C\}$ and $\{A, B\}$, with A and C receiving low weights, while B is assigned a relatively high weight in the second set (where it is near the center of the query kernel) and a low weight in the first. The per-point weights are then multiplied by the weight of projected points: $w_C = w_C * w_{initialpoint1}$. The N sets of weighted points are then merged: the result is the union of the sets. The weight of each point in the merged set is the sum of the point's weight across all sets that it of which it was a member. For example, B falls within both Q_1 and Q_2. After merging, B's weight will be $w_B = w_{B,Q_1} * w_{initialpoint1} + w_{B,Q_2} * w_{initialpoint2}$. The resulting weights represent an approximation of the likelihood that a given point would be reached from the current task state.

If there are remaining profile changes, we then project the merged set forward in time and continue as before. If this is the final profile change, a duration distribution is built, as in Section 5.4. The distribution is then shifted by t_n to yield the final duration prediction.

In this fashion, we are able to move through the training database to approximate the courses that may be taken by the current team. The approach outlined above results in a "complete" estimated distribution, but becomes infeasible when the team profile contains more than a few team changes. This is because with every change in the team profile, the number of points increases by a factor of the number of points within a query kernel. If the training data is even moderately dense, the number of points being tracked explodes and becomes computationally infeasible.

To address this, we use an approach similar to particle filtering (Gordon et al., 1993) by tracking O points, instead of all possible points. After performing the initial query at state S, we extract a weighted random sample (with replacement) of O points from the set of weighted points resulting from the query. These O points are then projected forward as before, and their state is modified by p_i. After applying a query kernel around each, we merge the resulting sets of weighted points as above and resample a new set of O points to again project forward. This eliminates

the compounding of the branching factor, and makes particle projection prediction computationally feasible, although accuracy is degraded. The system designer may select a value of O representing a reasonable tradeoff between computational expense and accuracy for his domain.

When projecting observations forwards, there is no guarantee that an observation will be in the training set exactly $(t_{i+1} - t_i)$ units ahead of the observation being projected. One approach would be to interpolate between the two observations preceding and succeeding the desired time. We instead use the observation nearest to the desired time, and track the cumulative error, e_j. With each projection, the distance between the time of the future observation and the desired time is added to e_j. Once the final projection and set of queries have been performed, the points are shifted by $e_j + t_n$ before kernels are placed around them and combined to build the estimated duration distribution. The complete algorithm is detailed in Algorithm A.3.

Particle projection prediction is subject to none of the disadvantages or restrictions of distribution transfer functions, and can achieve similar levels of accuracy, given sufficient particles. However, it is more computationally expensive, by a factor of approximately 30 (see Section 6.6.1), which make transfer functions attractive for domains in which their assumptions hold. Particle projection may also occasionally generate a predicted distribution that varies significantly from the "true" underlying distribution, if the subsampling step happens to select a group of low-probability runs. However, as in particle filtering, this is unlikely, and even more unlikely to occur for any number of consecutive predictions. Since we update our predictions at each timestep, any random error in prediction will be rapidly compensated for.

In this discussion, we have implicitly assumed that the exact arrival and departure times for agents joining and leaving the team are known (t_i is a scalar, not a distribution). This simplifying assumption has been made in this work for computational reasons. However, taking uncertain arrival (or, less likely, departure) times into account is a simple extension to our particle projection algorithm. Instead of projecting each point in $points$ by the scalar t_{j+1}, nest a loop inside the loop across all points. This nested loop would iterate across the set of weighted arrival times $t_{j+1,1}...t_{j+1,m}$ (either a discrete set, or a sampling of a distribution), and:

- Project pt_i into the future by $t_{j+1,k}$ time units
- Multiply $weight(pt_i)$ by $weight(t_{j+1,k})$ (the likelihood of this arrival time)
- Proceed as in lines 29-31 of Algorithm A.3

However, as this incurs obvious computational penalties, we chose instead to use discrete arrival times for the work reported here.

6.5.2 Reasoning About Engagement and Disengagement Costs

While particle projection prediction provides reasonable estimates of the effects of additional (or fewer) agents, it assumes that there is no cost to agents integrating into, or disengaging from, a team, and that these acts are instantaneous. Consider a *Transport* task that may use $[2, 4]$ transporter agents and $[0, 1]$ scout agents. Adding or removing a scout from the team will affect the team's ability to plan an effective route, but the *act* of addition or removal will not slow down the remainder of the team: the transporters will not make less progress while a scout agent is in the process of joining the team. In contrast, if a transporter is added, the rest of the transporters must stop, possibly set down the object being moved, and reposition to allow the new agent to grasp the cargo. In this case, the act of addition prevents further progress towards the goal from being made while the new agent is integrated into the team. Similar effects may occur while an agent is disengaging from the team. These degrading effects may reduce the benefit of a team change, and should be taken into account when the planner is reasoning about whether to add a team change to the schedule. We discuss here several potential methods for approximating these effects. However, note that our experimental system currently assumes that agents may join or depart a team instantaneously, although it does take setup actions (e.g. moves) into account.

The remainder of this section discusses a generic transfer of an agent from one team to another. We define such a transfer as consisting of *disengage*, *transfer*, and *integrate* tasks (Fig. 6.12), which represent the three stages of a transfer. When implemented, these may be combined into a single transfer task, but we will address them separately for the purposes of this discussion. The task that an agent is being removed from will be referred to as the *donor* task, and the receiving team as the *recipient*. During disengagement and integration, the agent is leaving or joining the relevant team, and may adversely affect the team's progress. The transfer task transitions the agent to a state from which it may join the recipient task; in practice, this is often a simple move task, but in some situations it may involve a complex plan.

When considering disengagement and integration costs, we must be able to calculate their effect on the donor's and recipient's duration distributions (timespans (a) and (e) in Fig. 6.12), in order to reason about whether a transfer is worthwhile. For clarity, we will refer solely to disengagement and donor tasks in the remainder of this section; integration and recipient tasks are representationally identical.

The disengage task represents that disengaging will take some non-negative amount of time (possibly zero) and will affect the donor task in no way (*nonintrusive*), by slowing its progress (*intrusive*), or by stopping its progress (*monopolizing*).

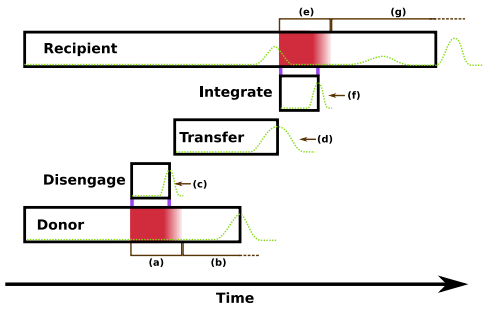


Figure 6.12: A prototypical agent transfer from a donor task to a recipient task. Example duration distributions are indicated by the dotted lines. Shaded areas indicate the portion of each task that is affected by the disengagement and integration tasks.

We have considered several approaches to representing the effects of a disengage task. While we have not experimentally evaluated the techniques discussed below, we feel that average drag is likely a reasonable compromise.

Expanded State Space

The most straightforward approach is to augment the donor task's state space with that of the disengage task, and perform particle projection prediction in this expanded space. The disengage task would be eliminated, and the start of the transfer task would be constrained to start after the end of the disengagement by specifying a constraint as a function of the donor task's state. This approach has the advantage of being able to model arbitrary effects, but has a number of detriments. Most notably, the donor task's state space explodes, as it will need to be able to represent the disengagement and integration of all of its role types throughout the task's state space. Since this amounts to the addition of one or more dimensions to the state space, a significant amount of additional training data will be required. Because our approach to prediction examines nearby points in the state space, examples of disengagement will need to be obtained throughout the execution of a task: a disengagement that is observed near the beginning of the task will not be included in a prediction for a disengagement scheduled to start near the end.

Expanding the donor task's state space is feasible when one of the following is true:

- The scenario's tasks have low-dimensional state spaces or are nearly deterministic, thus requiring only a small amount of training data.
- Training data is easy and cheap to collect.

- The effects of disengagement are not dependent on the donor task's state, which allows a few observations of disengagement to be synthetically inserted into the training database throughout the state space.

Besides demanding additional training data, this approach requires that the planner be able to represent a constraint on the start time of the transfer task as a function of the (continuously updated) state of the donor task. ASPEN's Parameter Constraint Network and Temporal Constraint Network are sufficiently expressive to represent this type of relationship.

Drag

If we do not include disengagement in the donor task's model, we must instead be able to transform the donor task's duration distribution to approximate the effects of the disengage task. We propose to do so by estimating the *drag* of the disengage task on the donor. We define *instantaneous drag* as the fraction of a timestep in the donor task that is consumed by the disengage task: a drag of 0 corresponds to a nonintrusive disengagement, while a drag of 1 represents a monopolizing disengage maneuver. In the (extremely unlikely) case that disengaging speeds progress in the donor task, drag would become negative, with a value of -1 representing an infinite increase in speed. By assuming that drag is neither dependent on the donor task's state nor on time, we are able to build an estimate of the *average drag* for each disengagement task, given a small set of examples.

In general, drag may be a function of the disengage task's state, and thus may change over time. For instance, a disengage task might require that the donor task pause during the initial portion of disengagement, but then resume normal (or slowed) operations for the remainder. While there are some very specific situations in which modeling drag as a function of time would prove useful, we feel the small increase in expressiveness does not warrant the additional complexity.

In order to estimate a disengage task's average drag, we wish to estimate the difference in the mean of the donor's predicted duration from one timestep to the next, where s is the length of the timestep: $\Delta = E_{dur}(t) - E_{dur}(t+s)$. During normal operation, Δ should always be equal (or nearly equal) to s: if the team works for s time units, the predicted duration should drop by s units, barring failures or other unexpected state changes. If the disengage task is interfering, Δ will be in the range $[0, s)$. Drag is the fraction of the timestep during which progress on the donor task was not made: $D = 1 - \frac{\Delta}{s}$.

By assuming that drag does not vary during the course of the disengage task, we may estimate D by comparing the expected duration of the donor task prior to and following the execution of the disengage task:

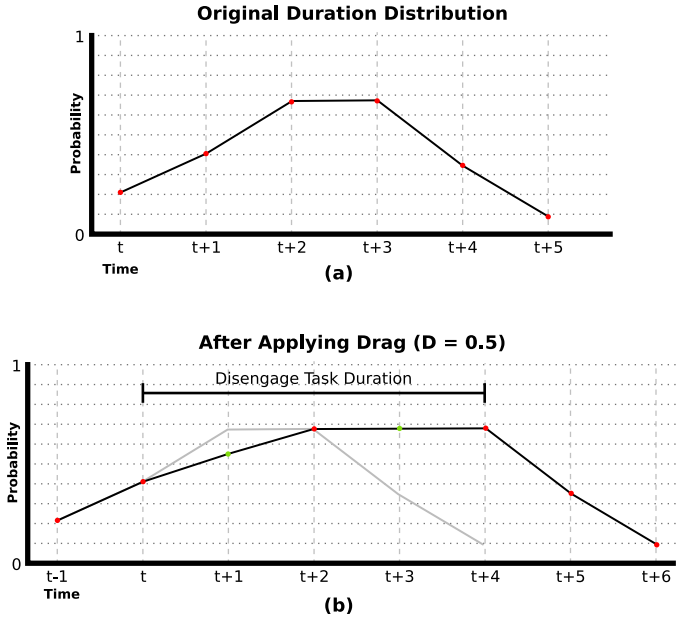


Figure 6.13: Applying the drag of a disengage task to the duration distribution of the donor stretches the distribution by the drag factor at each timestep during which the disengage task is active.

$$\begin{aligned} \Delta &= E_{dur}(t_n) - E_{dur}(t) \\ s &= t_n - t \\ D &= 1 - \frac{\Delta}{s} \end{aligned}$$

In order to estimate D for each disengage task, we require a set of training examples that do not include disengagements, as well as a small set that do. The former is used to construct $E_{dur}(t_n)$ and $E_{dur}(t)$, given the task states at the beginning and end of the disengagement from the latter data set. The no-disengagement training set is exactly the data set used by particle projection prediction. The final estimate of D is an average of the Ds estimated from all examples of a given disengage task.

Given the duration distribution for the donor task, an estimate of D for the disengage task, and an estimate of the disengage task's duration, it is straightforward to apply the drag to the relevant portion of the donor's duration distribution (Fig. 6.13). For each timestep t_i of length s, ranging from the beginning of the

disengage task (time t) to its end (time t_n), the $[t_i + s(1 - D), +\infty]$ segment of the duration distribution is shifted to the right by sD units, interpolating between t_i and $t_i + s$. This is repeated for $i = [0...n]$, where n is the estimated duration of the disengage task. Each successive shift is applied to the output of the previous step. The intuition is that only a fraction $(1 - D)$ of the progress that should have been made during timestep t_i actually is achieved, due to the effect of the disengage task. This has the effect of stretching or delaying the duration distribution. Note that only a portion of the duration distribution will be affected: the segment of the distribution prior to the start of the disengage task will remain untouched, while that portion after the end of the disengage is simply shifted.

Summary

In many robotic tasks, the act of adding or removing an agent from a team is neither instantaneous nor free. We have examined several approaches to estimating the effects of agent disengagement and integration on the team's ability to progress towards the goal. Average drag appears to be the preferable solution, although we have not experimentally evaluated either approach.

6.6 Experimental Results

We have performed two experiments to evaluate duration prediction for mutable teams and the effects the use of mutable teams have on generated schedules. We first evaluate the comparative accuracy and efficiency of predicting duration distributions for mutable team tasks with distribution transfer functions and particle projection predictions. While distribution transfer functions are much more computationally efficient than particle projection predictions, they are significantly less accurate, making projection the preferred method for building mutable team duration distributions. We conclude with an evaluation of the effect of mutable teams on the optimized schedules generated by ASPEN prior to execution. This allows us to examine the effects of mutable teams in isolation from live duration prediction and live task modification. The addition of mutable teams resulted in schedules a statistically significant 5.65% (33.04 minutes) shorter on average than was achievable otherwise.

6.6.1 Duration Prediction with Mutable Teams

Accuracy

We have evaluated the accuracy of distribution transfer functions and particle projection predictions, using the *Transport* task. *Transport* is the most complex task in our domain, with the largest state space. We chose 60 initial states to evaluate, scattered throughout *Transport*'s three-dimensional state space. The two approaches were used to build predicted duration distributions for each initial state with 1, 2, 3, 4, and 5 scheduled changes in the team profile. Particle projection prediction was evaluated with 5, 10, 20, 30, 40, and 50 particles.

We used particle projection predictions with 1000 particles as ground truth. Ideally, we would use the complete projection, in which no subsampling is performed and the entire tree of previously observed states is traversed. Unfortunately, this is computationally infeasible for the *Transport* task: there are a total of 660,556 points in the training data set, with an average of 38 included in each query. A branching factor this high renders complete projection impossible for more than two team changes.

In order to validate that 1000 particles is sufficiently accurate, we performed a brief experiment with the *Lift* task, which is simple enough that the complete projection can be computed. As can be seen from Fig. 6.14, the accuracy of particle projection rapidly asymptotes as the number of particles increases (note that the X axis is a logarithmic scale). While the rate of decrease is in part a function of the number of changes to the team profile, all of the curves have nearly reached zero divergence from the complete projection when 1000 particles are available.

As in the experiments evaluating the accuracy of duration prediction for immutable teams (Section 5.7.1), we evaluated the accuracy of the predicted distributions for the *Transport* task by computing the Kullback-Leibler (K-L) divergence (Kullback and Leibler, 1951) between the PDFs of each test distribution and the corresponding ground truth. As can be seen in Fig. 6.15, distribution transfer functions become progressively more inaccurate as the number of changes in the team profile increases, and the approximations in the transfer functions are compounded. In contrast, particle projection prediction is unaffected by the number of changes. The accuracy of the projected distributions can be adjusted by selecting the number of particles used, with as few as 30 particles closely approximating the ground truth.

Efficiency

While particle projection prediction offers significant accuracy gains over distribution transfer functions, it comes at a corresponding computational cost. Fig. 6.16

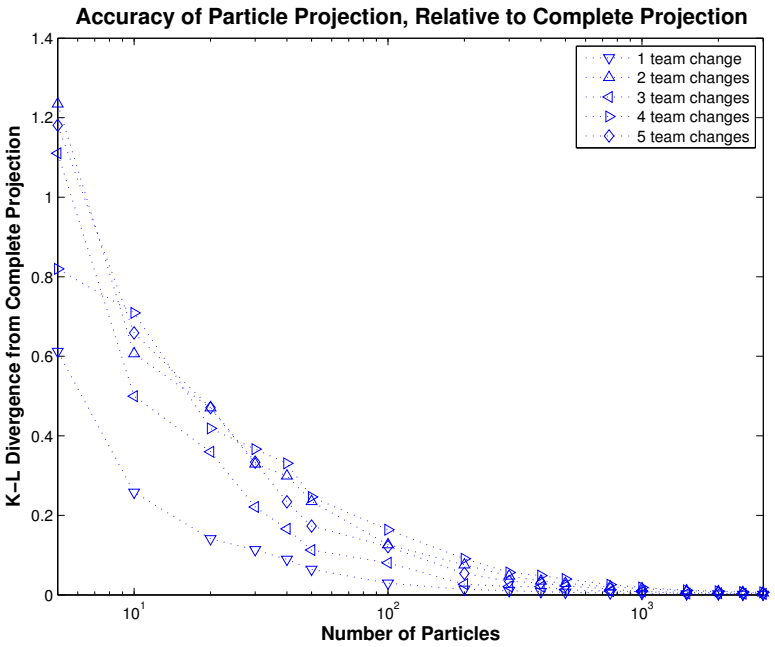


Figure 6.14: The accuracy of particle projection, as compared with the complete projection, for the simple *Lift* task. All curves asymptote to zero divergence once 1000 particles are used, validating our use of 1000 particles as ground truth when experimenting with the more complex *Transport* task. Note that the X scale is logarithmic.

plots the average query time as a function of the number of expected changes in the task's team. Each point in Fig. 6.16 is the average of 60 queries at initial states scattered throughout *Transport*'s state space. As would be expected, the query time increases linearly with the number of projections or transfers that must be calculated, and is significantly greater for particle projection, with the slope dependent upon the number of particles in use. By examining the characteristics of the tasks in a particular scenario, the designer may select a particle count representing an appropriate compromise between accuracy and speed. As can be seen from Fig. 6.17, accuracy for this task increases with the particle count, but begins to plateau in the neighborhood of 20 particles. The results in the remainder of this thesis utilize 25 particles. While optimization of the particle prediction code has been performed, it may be possible to increase its speed with further work. During scheduling operations, a cache of recently predicted distributions is maintained, lowering the amortized cost of repeated predictions.

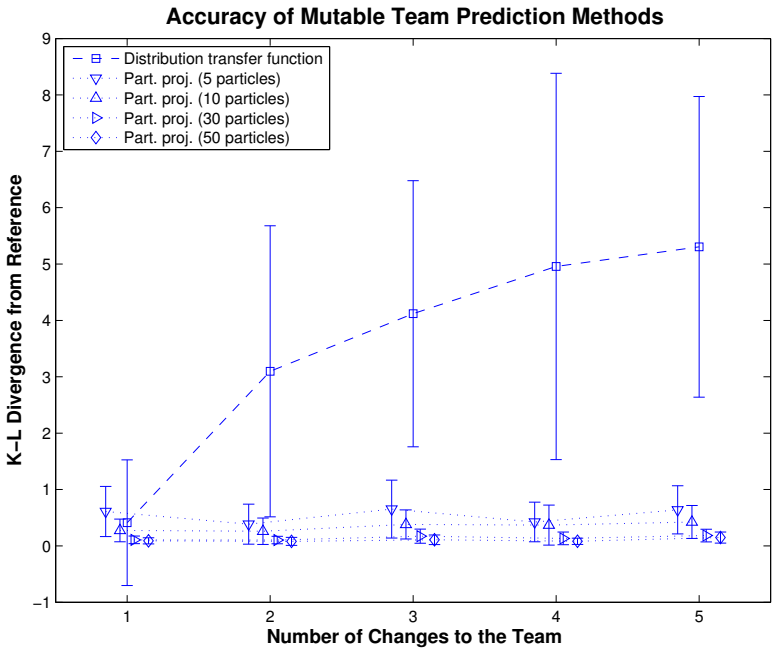


Figure 6.15: As the number of changes to a team (e.g. number of arrivals and departures) increases, the divergence of distribution transfer functions from the true underlying distribution increases rapidly, while the accuracy of particle projection prediction is unaffected, with the base accuracy determined by the number of particles. Note that the projection points are offset along the X axis for clarity, and the 20- and 40- particle data are not plotted.

6.6.2 Effect of Mutable Teams on Initial Plans

We have evaluated the effects of using mutable teams in the optimized plans generated by ASPEN prior to the start of execution. This experiment does not include execution of the schedule, as we are isolating the effects of mutable teams. The complete proactive replanning experiment reported in Chapter 8 evaluates the effects of mutable teams at execution time, along with live duration prediction, live task modification, and their interactions.

When used in isolation, mutable teams have a salutary effect on the schedule: we are able to build schedules on average 5.65% shorter than is possible with immutable teams and the same scenario. In addition to their utility while generating schedules, mutable teams account for approximately half of the 11.5% reduction in executed schedule length achieved by proactive replanning in the experiments of Chapter 8.

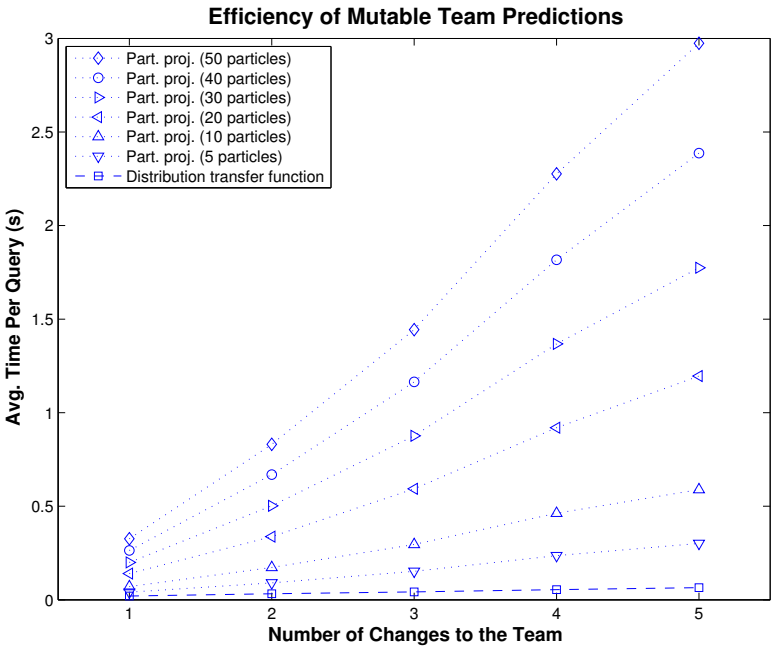


Figure 6.16: The computational cost of both particle projection prediction and distribution transfer functions increases with the number of changes, although the rate of increase for particle projection is much greater, and is determined by the number of particles.

Scenario

The scenario used in this experiment, as well as the experiments in Chapter 8, represents activities occurring shortly after a lunar landing: the construction of four communications arrays, the laying of cable from the arrays to a habitat, and habitat resupply operations. Five agents are available, three distinct locations are involved, and eight classes of tasks must be performed (Fig. 6.18, Table 6.2). Some groupings of tasks must be performed serially, while others may be placed freely by the planner. Each task or task group must be performed multiple times, with the repeat count denoted in the *Count* column of Table 6.2. In the case of task groups, such as [*Supply Habitat*, *Stow Supplies*], each instance of the group is internally serialized, but the group instances may be scheduled at any time with respect to each other. 24 cooperative tasks and 11 grouping tasks must be scheduled in all, with the total task count averaging 87.9 ($\sigma = 5.69$) in the mutable team case after join and move tasks are scheduled. The immutable team condition yields an average task count of 84.6 ($\sigma = 5.31$), the variation resulting from differences in which op-

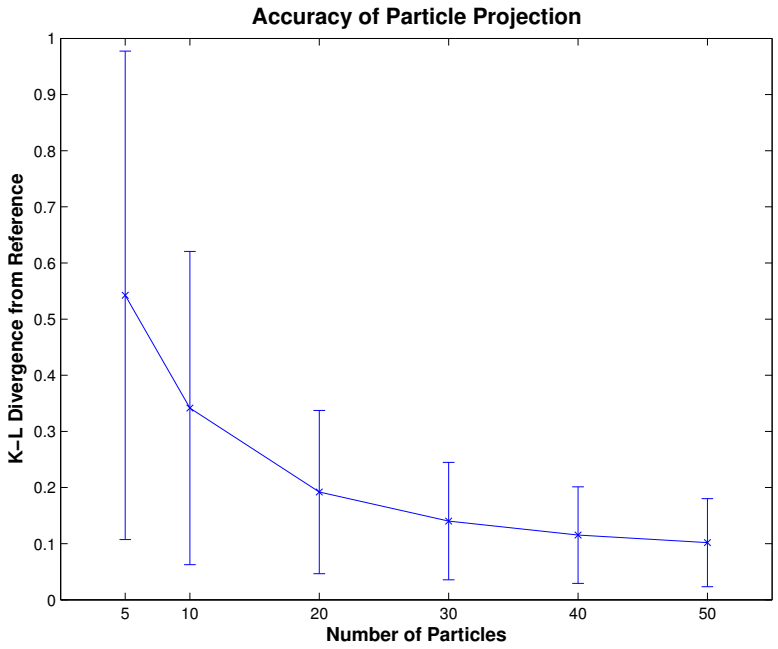


Figure 6.17: The accuracy of particle projection is determined by the number of particles used. This plot averages the K-L divergence across all team changes for each number of particles, using the same data set as Figs. 6.15 and 6.16.

tional roles are filled or left open. A one-way analysis of variance (ANOVA) shows that this difference is statistically significant ($p = 0.000035$, $F(1, 198) = 17.88$). Agents are again assumed to be homogeneous, and can participate in only one task at a time. Tasks differ in their initial and ending locations, duration, and available roles. The tasks are detailed in Table 6.2, while the temporal relationships of the grouped tasks are noted in Fig. 6.18.

The objective of the scenario is to minimize the executed makespan: that is, to complete the specified set of tasks in the minimum amount of time. In each experimental run, ASPEN begins with an empty schedule and a list of the cooperative tasks that must be accomplished. The iterative repair algorithm is executed until a valid (but unoptimized) schedule is constructed. Once a complete schedule has been formed, 500 iterations of ASPEN's heuristic optimization algorithm are executed, with the final schedule serving as the output of the run (note that no execution is performed in this experiment). The optimization algorithm will backtrack to a previous schedule if it has been repeatedly unable to improve the makespan, allowing it to climb out of local minima (Algorithm 3.4).

Table 6.2: CommTower scenario tasks and relevant statistics. *Count* denotes the number of times each task must be performed, while *Duration from Start* is the average duration for the task, given the minimal set of agents.

Task	**Count**	**Agents Needed**	**Location**	**State Variables**	**Progress Per Timestep**	**Duration from Start** (μ, σ)
Move	*	$[1, 1]$	Start: Anywhere End: Anywhere	Distance remaining: $[0, 100]$ Recovery progress: $[0, 50]$	$N(1, 0.05)$	106.98 (13.9)
Transport	4	$[1, 3]$	Start: Lander End: Comm	Distance remaining: $[0, 60]$ Recovery progress: $[0, 50]$	$N(0.5, 0.1)$	169.92 (62.34)
Lift	4	Lift: $[1, 1]$ Brace: $[0, 1]$	Comm	Progress: $[0, 1]$	$N(0.025, 0.01)$	59.28 (31.48)
Assemble	4	$[1, 2]$	Comm	Progress: $[0, 1]$ Pickup: $[0, 5]$	$N(0.05, 0.01)$	24.46 (8.02)
Connect	4	$[1, 2]$	Comm	Connectors Done: $[0, 5]$	$N(0.3, 0.25)$	20.00 (5.16)
Lay Cable	2	$[1, 3]$	Start: Comm End: Habitat	Distance remaining: $[0, 40]]$ Snag recovery: $[0, 50]$	$N(m, 0.1)$, $m = f(distance)$	76.43 (27.27)
Supply Habitat	3	$[1, 3]$	Start: Lander End: Habitat	Distance remaining: $[0, 60]$ Boxes loaded: $[0, 10]$	$N(1, 0.1)$	84.80 (4.41)
Stow Supplies	3	$[1, 2]$	Habitat	Boxes Remaining: $[0, 10]$	$N(0.5, 0.25)$	22.56 (5.66)

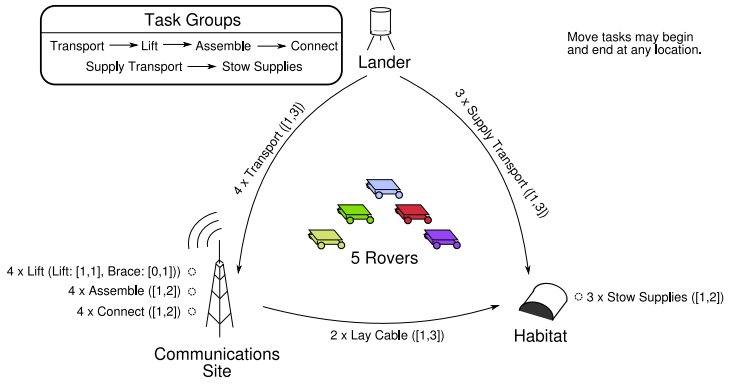


Figure 6.18: The CommTower scenario. The repeat count for each task precedes it, while the agent bounds for each role follow. The metric for this scenario is the makespan of the schedule.

The planner represents the location of each agent as one of five discrete states: three corresponding to the three locations (*Lander*, *Comm*, and *Habitat*), one representing an agent in motion (*Moving*), and one representing a non-moving agent that is not at one of the three discrete locations (*Stranded*). The only way to reach the *Stranded* state is by departing a moving task (such as *Transport*) prior to its completion. When an agent is moved from the *Stranded* position or moves to join a task already in motion, the planner applies a simple heuristic to estimate the distance that the agent must move, in order to schedule a *Move* task of the proper duration. This heuristic uses the already-predicted durations of all involved tasks, fixed coordinates for the three discrete locations, and assumes constant velocities to estimate the position of stranded robots, as well as where a moving robot will intercept a moving team. When this scenario is executed in the experiments discussed in Chapter 8, the executive and simulator track the metric position of each agent, as well as more refined locations at which each task instance must be performed. The executive inserts additional short moves into the schedule if agents are not precisely in the correct position. See Chapter 8 for details. We utilize a coarse model of position in the planner due to the difficulty of representing and reasoning about exact positions in ASPEN. Instead, our heuristics estimate the agents' metric locations when building duration predictions prior to execution, and utilize their actual locations, as reported by the executive, during execution. In this way, we are able to efficiently plan using a coarse location representation while implicitly taking a finer measure into account via our predictions.

This scenario differs from that in Chapter 5 in several ways. Most importantly, the goal here is to minimize makespan, while in Chapter 5 the objective was to

maximize reward within a fixed planning horizon. Consequently, this scenario specifies a fixed set of cooperative tasks that must be performed, while Chapter 5's scenario describes a set of task types that may be performed in any quantity. Chapter 5 did not consider temporal constraints between tasks, such as those within the task groups here (e.g. [*Supply Habitat*, *Stow Supplies*]). In addition, optional roles had a uniform effect on the duration of the task, while in this scenario some are useful only in particular states (e.g. when recovering from a failure). Finally, Chapter 5's scenario did not model position beyond three discrete locations, as without mutable teams an agent would never complete a task at any other position.

Experimental Conditions

In this experiment, the objective is to generate a schedule with the minimum makespan, but not to execute it, as we are isolating the effects of mutable teams from the other components of proactive replanning. We evaluated two experimental conditions: with and without mutable teams. When mutable teams are allowed, the planner may schedule agents to join or depart from teams in the middle of a task, and to participate in any subset of the task's execution. This provides additional flexibility to the planner, but will increase the overall number of tasks on the schedule and the cost of forming duration predictions, with potential consequences for the computational power required. The search space is expanded tremendously by the addition of variable arrival and departure times for each agent. We manage the combinatorics through the use of a suite of heuristics that make stochastic decisions and utilize some simplifying assumptions. Our heuristic suite is discussed in detail in Section 8.2. When mutable teams are not allowed, the planner must schedule agents to participate in the entirety of a task: they must join the task at its start time, and not depart prior to its completion. The planner still has the flexibility to fill optional roles or leave them vacant, but cannot divide a role slot between agents or have a role filled only during a portion of the task's scheduled duration.

We generated 100 optimized schedules in the fashion described above under each condition, yielding a total of 200 data points. We compared the makespan, iterations of repair, and total planning time of the two conditions to determine the effects and costs of mutable teams. One outlier in each data set was discarded when evaluating makespan, as they fell more than three standard deviations from the mean. These outliers were likely due to our stochastic optimization heuristics making a long series of poor optimization decisions.

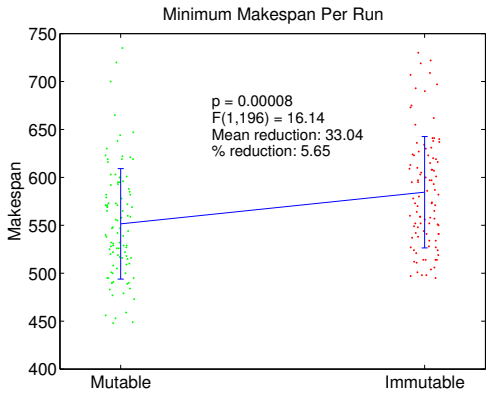
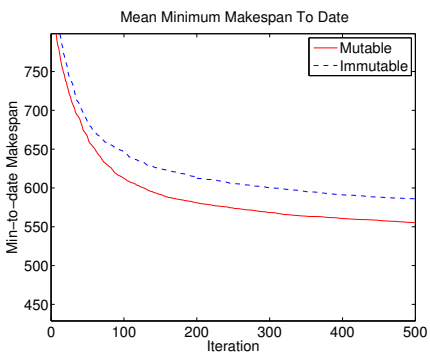


Figure 6.19: The use of mutable teams (left) results in schedules that are shorter than otherwise possible (right) by a statistically significant degree. Individual schedules are plotted as points behind the mean and error bars (at one standard deviation), and are randomly dispersed along the X axis for clarity.

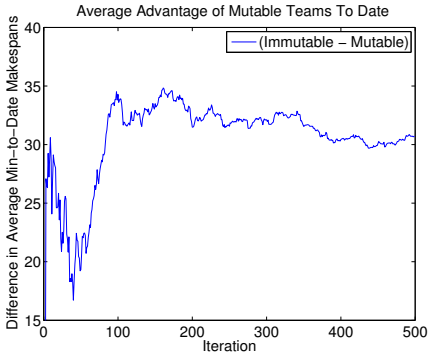
Data and Discussion

On average, schedules incorporating mutable teams were 5.65% (33.04 minutes) shorter than those constructed with immutable teams. A one-way analysis of variance (ANOVA) showed that the effect of mutable teams was significant, with $p = 0.00008$ and $F(1, 196) = 16.14$. Fig. 6.19 plots the final schedules as points, randomly dispersed along the X axis for clarity, overlaid with the mean and error bars (at one standard deviation).

Fig. 6.20(a) plots the best schedule found as a function of the number of optimization iterations performed. Our approach to optimization (Algorithm 3.4) caches the shortest conflict-free schedule achieved as optimization progresses, and reloads it at the end of optimization. As a result, Fig. 6.20(a) plots the average makespan that would be achieved if optimization were to cease after a given number of iterations. Each point on each curve is an average of the minimum observed makespan across all 100 runs after the given number of iterations of optimization. The clear asymptoting of the curves suggests that while additional improvement should be possible with more iterations of optimization, we have reached the point of diminishing returns. Further, Fig. 6.20(b) plots the average advantage of the mutable team approach by subtracting the two curves in Fig. 6.20(a). We can see that, while both approaches improve throughout the 500 iterations, the mutable teams condition does not make significant gains relative to immutable teams after approximately 100 iterations of optimization. The initial drop, between approximately 5 and 30 iterations, is likely meaningless. The standard deviation of the data for low



(a) The minimum makespan achieved on average, as a function of the number of optimization iterations. Mutable teams consistently outperform immutable, with their advantage increasing through approximately 100 iterations.



(b) The average advantage of mutable teams, as a function of the number of optimization iterations performed. This is the difference between the curves plotted in Fig. 6.20(a).

Figure 6.20: Average makespan achieved for plans involving mutable and immutable teams, as a function of the number of optimization iterations.

numbers of iterations is quite high, as it is heavily dependent upon the state of the schedule after the initial round of repair completes.

From Fig. 6.19 and 6.20, we can conclude that mutable teams provide a significant improvement in the makespan of the final schedule and that, in this scenario, 100 iterations of optimization are sufficient to achieve the maximal separation between the mutable and immutable team conditions.

One danger of increasing the flexibility of a planner is that the increased complexity will require additional computation, more iterations of repair or optimization, or both. This is exacerbated in the case of mutable teams, as duration prediction becomes significantly more computationally complex. In the case of mutable teams, each iteration of repair and optimization is individually slower, but far fewer iterations of repair are necessary. This is a result of the expanded set of repair techniques utilizing mutable teams that the planner may employ. In general, they have a localized effect on the schedule and create few, if any, additional conflicts when applied. In contrast, other repair methods affect large portions of the schedule, potentially creating new, or worsening, existing conflicts.

Fig. 6.21 plots several metrics of interest from this experiment. All reported p and F values are the result of a one-way ANOVA on the metric in question. Fig. 6.21(a) and (b) evaluate the cost of constructing the initial, unoptimized, schedule. Recall that each experimental run begins with an empty schedule and

a list of tasks that must be performed. The planner then repairs the schedule, without optimizing, until it is conflict-free. Fig. 6.21(a) and (b) plot the number of repair iterations and time needed to do so, respectively. We can see that when mutable teams are available, the planner requires on average 320.75 (57.5%) fewer iterations of repair to construct a valid schedule. The effect of mutable teams on this measure is statistically significant ($p < 0.00001$, $F(1, 198) = 36.06$). However, each iteration of repair is more time-consuming, resulting in a planner that utilizes mutable teams requiring nearly the same amount of time to construct the initial schedule as without them. In fact, the difference in time is statistically insignificant (Fig. 6.21(b)). The additional repair methods available when mutable teams are enabled are not inherently more expensive than the core methods. Rather, the increase in expense largely is a result of the computational complexity of constructing a duration prediction for a mutable team. This implies that if our duration prediction approach were to be optimized, smaller training sets were used, or fewer particles were used during prediction, mutable teams would provide significant reductions in both the number of repair iterations and the total time required to repair.

The remainder of Fig. 6.21 explores related metrics. Fig. 6.21(c) and (d) are analogous to (a) and (b), but instead plot the total number of repairs (and time taken to repair) for the entire run, including both the formation of the initial schedule and the repair of conflicts created during optimization. These results further support the conclusions we drew from Fig. 6.21(a) and (b).

Fig. 6.21(e) graphs the end-to-end time needed to move from an empty schedule to a valid, optimized one. Again, the planner is able to make use of mutable teams with no overall computational penalty, as the reduced number of repair iterations compensate for their individually increased cost, as well as the greater cost of optimization.

Finally, Fig. 6.21(f) plots the time taken to perform the 500 iterations of optimization, excluding any repair needed to resolve conflicts caused by the optimization methods. As expected, the planner requires more time to perform the same number of optimization iterations when mutable times are available, again due to the increased complexity of duration prediction.

Our approach to optimization backtracks if progress has not been made for a given number of iterations, as detailed in Algorithm 3.4. One measure of how effective the optimization methods are is to evaluate the number of times this backtracking occurs. As can be seen in Fig. 6.22, when mutable teams are available, the planner backtracks an average of 7.07 times. In contrast, 14.5 backtracks occur during the average run with immutable teams. The effect of mutable teams on the number of backtracks is significant, with $p < 0.00001$ and $F(1, 198) = 83.60$, implying that optimizations performed with mutable teams are on average more effective. This is the case because mutable teams allow the planner to more finely

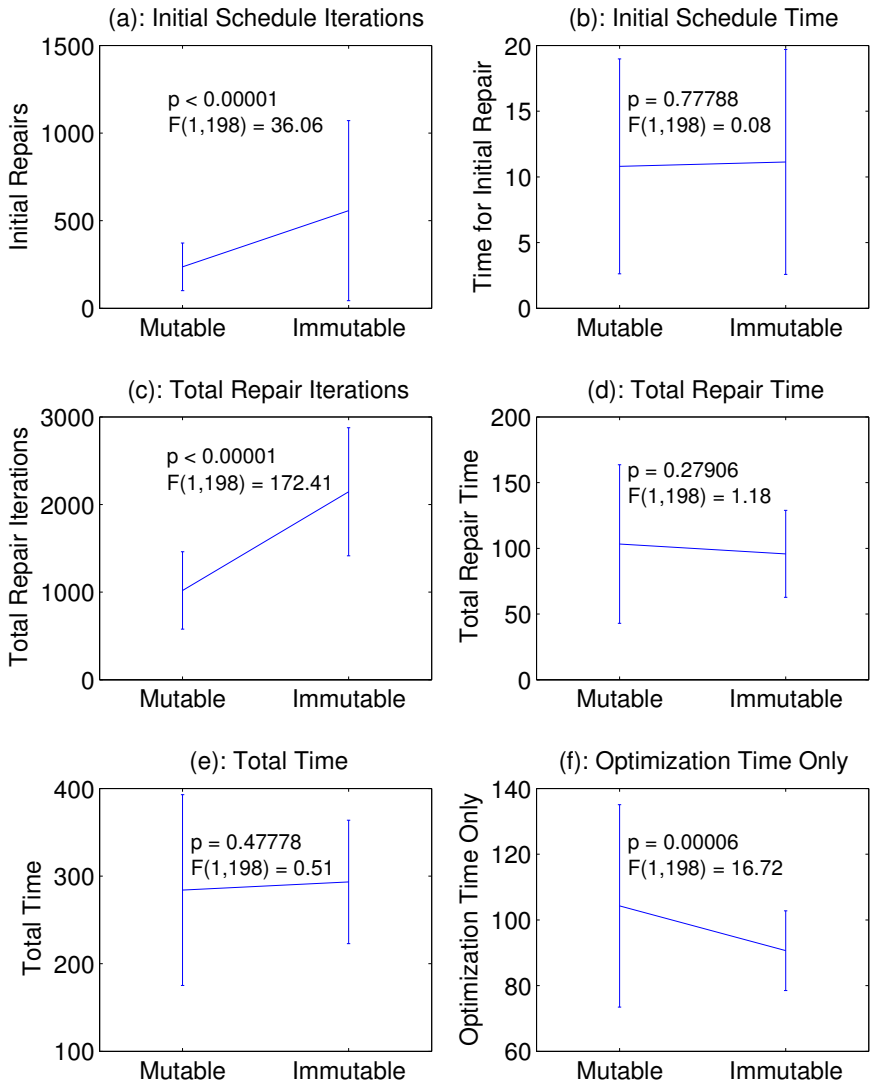


Figure 6.21: Mutable teams consistently allow the planner to repair the schedule with fewer iterations (a, c), with no computational cost (b, d, e). (a) plots the number of iterations of repair needed to construct an initial, unoptimized schedule, while (b) graphs the time needed to do so. (c) charts the number of iterations of repair needed to construct the initial schedule and repair all conflicts introduced by the optimization process, while (d) depicts the time taken. (e) graphs the total time taken to construct and optimize the schedule, while (f) plots the time required by optimization, not including any resulting repairs.

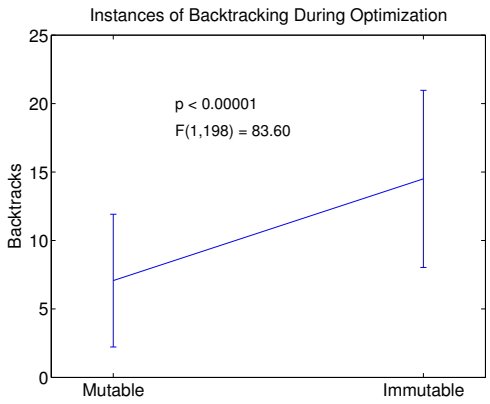


Figure 6.22: The optimization procedure (Algorithm 3.4) backtracks to the best schedule observed to date if no significant progress is made after a given number of optimization iterations. A planner utilizing mutable teams does so significantly fewer times, indicating that optimizations are on average more useful.

allocate available agents: if an agent is idle for a relatively brief period of time, it temporarily may assist a task with an open optional role, while a planner utilizing immutable teams would be unable to make use of the idle agent without rearranging the schedule to allow its participation in the entire task. By using mutable teams, the planner is able to maximize the utility of the available resources.

This experiment has demonstrated that mutable teams provide significant improvements in the generated schedule, while increasing the effectiveness of plan repair sufficiently to counteract the increased cost of forming duration predictions. Given further improvements to the efficiency of our approach to duration prediction, mutable teams could well improve all aspects of plan repair and optimization.

6.7 Summary

Mutable teams are those to which agents may be added, or from which agents may be removed, during the execution of a task, enabling the more fluid and efficient use of available agents. Mutable teams require a more complex task representation in the planner in order to encode and track required and optional roles, as well as a flexible executive. However, the addition of mutable teams greatly increases the planner's available menu of options during repair and optimization. Schedules for our experimental scenario built with mutable teams are significantly shorter on average than those built without mutable teams, and can be constructed with the same computational effort. While individual iterations of repair last longer

as a result of the increased cost of duration prediction, fewer cycles of repair are necessary, due to the usefulness of mutable teams in repairing conflicts. These factors offset to yield a system capable of constructing more efficient schedules in the same amount of time.

Chapter 7

Live Task Modification

7.1 Overview

Live task modification is the act of changing some aspect of an executing task, generally in an attempt to take advantage of an opportunity or alleviate a problem detected during execution. In this thesis, we address the adjustment of the team profile of an executing task by the planner. Recall that the team profile is the schedule of when agents will join and leave the team: by modifying an executing team's profile, the planner is able to adjust the arrival and departure times of agents, as well as add or remove agents from the team. This greatly expands the set of techniques the planner may utilize when reacting to events as they occur, and results in a more flexible and robust system.

In contrast, existing planning and execution systems do not allow the planner to modify tasks once they have been dispatched to the executive. This yields a clean separation between the domains of the planner and the executive. By instead granting the planner control over agent allocation at all times, a proactive replanning system is able to more effectively execute schedules in dynamic, uncertain environments.

To support live task modification, the planner must be able to reason rapidly about whether to modify executing tasks. If the planner is too slow to respond to events, the continuing execution of the task may either render the state upon which the planner is basing its decisions invalid or proceed past the point at which useful changes could have been applied. A proactive replanner must be able to evaluate rapidly whether repairs or optimizations should be applied to executing tasks.

A live task modification-enabled executive must accept changes to the team profile of an executing task. In order to do so, it must be possible for the executive to retract or modify tasks and constraints that have been dispatched to it by the

planner, a capability that some existing execution systems lack. In addition, the executive must not make any assumptions about the future availability of agents: the planner may remove any agent from an executing task at any time.

While live duration prediction and mutable teams may be evaluated in isolation, our formulation of live task modification assumes the existence of mutable teams. Immutable teams do not support the mid-task addition or removal of agents, which make this approach to live task modification inapplicable. As a result, we have not experimentally validated live task modification independently of the other components of proactive replanning. Instead, it is evaluated in concert with various combinations in the experiments reported in Chapter 8.

We frame our discussion of live task modification in terms of changes to the team profile, but the general concept is much broader, encompassing any change to an executing task made by the planner. This is an open area of research that encompasses classes of modifications such as changing the parameters or goals of an executing task, or abandoning tasks already underway.

Live task modification is a relatively simple concept on the surface: transfer a modicum of control over executing tasks from the executive to the planner. The challenge is in the details: supporting live task modification requires an agile planner and a flexible executive. However, the resulting system is able to respond fluidly to unanticipated events that occur during execution, rapidly adapting the schedule to accommodate the realities of execution.

7.2 Applicability

Live task modification is applicable to any scenario that supports mutable teams, although it is most effective when combined with live duration prediction. When live duration prediction is available, the planner is provided with advance warning of problems or opportunities resulting from execution anomalies. Without prediction, the planner will often not detect an anomaly early enough for live task modification to be applied. Live task modification allows a proactive replanning system to respond directly to execution-time events by reallocating agents between tasks, rather than being restricted to reworking the portion of the schedule that has not yet begun execution. This makes a variety of plan repair and optimization techniques feasible, such as load balancing and arrival time adjustment.

7.2.1 Load Balancing

Load balancing is a natural application of live task modification that allows a proactive replanning system to ameliorate the effects of execution-time problems and

take advantage of opportunities as they are presented. Load balancing is the act of transferring agents between multiple executing tasks in order to approximately equalize their durations, resulting in a schedule with a shorter makespan. Fig. 7.1 demonstrates this application of live task modification. In this example, task A has a single role with agent bounds of $[1, 3]$, while task B's role has bounds of $[1, 2]$.

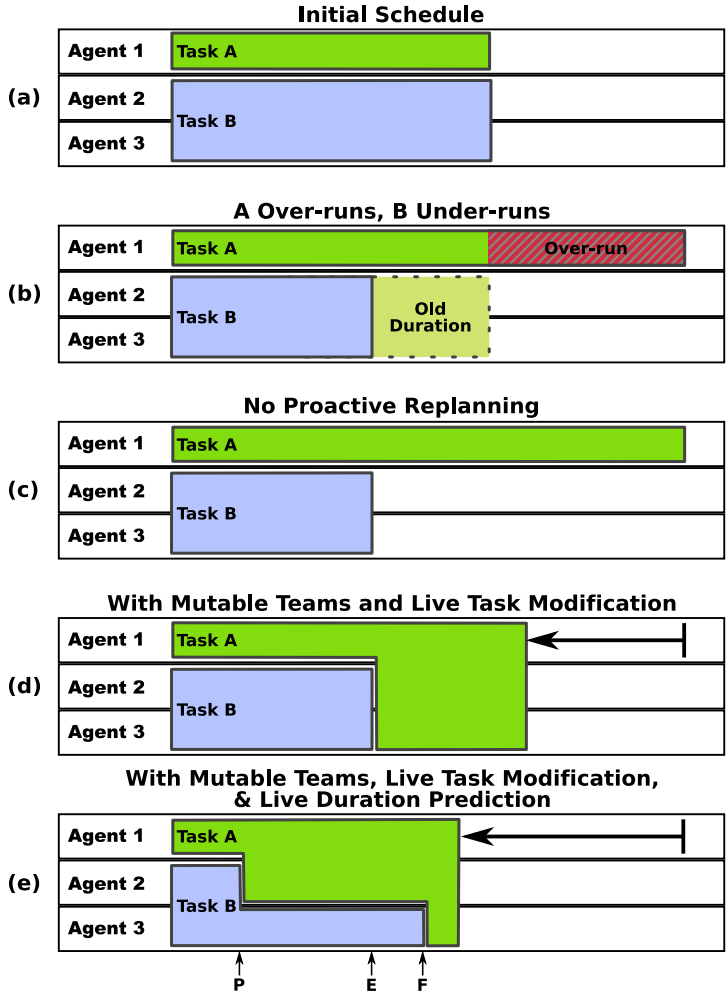


Figure 7.1: When used in concert with mutable teams, live task modification allows the more efficient execution of a schedule (d) than otherwise possible (c). If live duration prediction is also supported, further efficiencies are possible (e).

In the initial schedule (Fig. 7.1(a)), the planner expects the tasks to complete at the same time, with two agents allocated to B and one to A. Fig. 7.1(b) depicts

one possible result of execution: task A over-runs, while task B under-runs. In the absence of any elements of proactive replanning, the makespan of the resulting schedule is less than efficient (Fig. 7.1(c)). However, if mutable teams and live task modification are available, the agents assigned to task B may be added to task A once B has completed, partially counteracting the effects of the over-run (Fig. 7.1(d)). If the system also supports live duration prediction, an even more efficient execution of the schedule is possible (Fig. 7.1(e)). In this case, if the over- and under-runs are predicted at point P, the planner may react immediately by transferring agent 2 from its optional role in task B to an optional role in task A. This increases the duration of task B from point E to point F, but reduces the schedule's overall makespan. Agent 3 may then be added to task A's final optional role slot once B completes, further reducing the makespan by a slight but measurable amount.

7.2.2 Arrival Time Adjustment

Arrival time adjustment is an application of live task modification used to change a team profile in response to events affecting other tasks. If a task preceding an agent's participation in a cooperative task over- or under-runs, the planner may utilize live task modification to adjust when the agent joins the cooperative task. This allows the planner to resolve classes of conflicts that are unsolvable without live task modification.

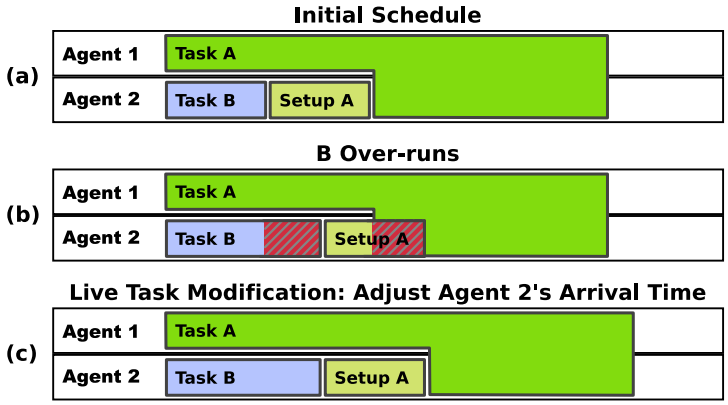


Figure 7.2: Live task modification is necessary to resolve the conflict that occurs when a task over-runs, making its agent unable to meet its scheduled arrival time for a cooperative task.

Fig. 7.2 illustrates a situation in which live task modification is necessary to resolve a conflict arising from the realities of execution. The initial schedule

(Fig. 7.2(a)) involves agent 2 performing task B in parallel with the beginning of task A, then joining task A once B and a setup action are complete. If task B over-runs (Fig. 7.2(b)), agent 2's setup task will overlap its commitment to participate in task A. If live task modification is not available, the planner is unable to resolve this conflict. When task B over-runs, both task A and B have been committed to the executive, along with their team profiles. Without live task modification, the planner can neither adjust the team profile of task A, nor abandon task B to give agent 2's setup task the time it needs. In such a situation, the planner would be forced to rely on the executive to resolve the problem. This requires the executive to have the reasoning power to resolve this type of conflict, a function that is properly in the planner's domain. However, if live task modification is available, the planner is able to adjust task A's team profile by delaying agent 2's arrival (Fig. 7.2(c)) and accepting a slightly longer task A as the price of resolving the conflict.

7.2.3 Summary

Our formulation of live task modification is applicable to any domain in which mutable teams are available. In general, live task modification is useful whenever it is possible to adjust executing tasks in a meaningful fashion. Its addition expands the planner's domain to include all agent assignments, rather than only assignments to tasks that have not yet begun execution. This increased power allows the proactive replanner to respond to execution-time events by directly adjusting the affected tasks, rather than being limited to compensating for the effects of the events on the remainder of the schedule. In addition, live task modification is required to resolve a class of conflicts that may arise if mutable teams are utilized. While live task modification is of little utility when used in isolation, it magnifies the proactive replanner's flexibility, robustness, and capability when combined with live duration prediction and mutable teams.

7.3 Task Modification and the Planner

The effective use of live task modification requires a planner that is capable of rapid, reactive plan repair and optimization. When used as part of a comprehensive proactive replanning system, state updates will continuously arrive from the executive, resulting in frequent updates to the duration predictions for executing tasks that must be incorporated into the planner's schedule. As problems and opportunities appear, the planner must be able to recognize them, formulate a response, and communicate the desired changes to the executive before the state changes upon which the planner based its reasoning. The time available for reasoning will vary

widely between domains, from a few seconds to minutes or even hours, depending on how rapidly tasks evolve. Multi-agent domains usually will require responses within minutes, which often is insufficient time for a planner to build an entirely new multi-agent plan in a complex domain. As a result, an effective proactive re-planner must be capable of quickly updating and repairing the near-term portion of the schedule.

7.3.1 Required Planner Capabilities

First, a live task modification-enabled planner must be capable of supporting live duration prediction: without ongoing predictions, the planner will not have the information necessary to formulate an effective response to any execution-time anomalies. This requires that the planner be able to receive high-frequency[1] state updates from the executive, re-predict the duration distributions for executing tasks, and update its schedule to reflect the new information. If the planner takes too long to repair or optimize the plan, it may not receive a new update indicating a problem or opportunity until it is too late to address. One solution to this problem may be to interleave state updates and iterations of repair or optimization.

In addition to running repair and optimization routines as time allows, it is desirable for them to be triggered immediately in response to updates. This would reduce the lag between the time when an update is generated and the point at which task modifications are sent to the executive in reply. While a planner may support live task modification if it only supports periodic revisiting of the schedule, it will be unable to make full use of live task modification's potential.

Once the planner has applied a set of updates from the executive, it must rapidly repair and optimize the near-term schedule, potentially at the cost of a less-optimal long-term schedule. A variety of approaches to such near-term repair have been proposed in the literature; we discuss two in the following section.

Finally, in order to make use of the opportunities presented by live task modification, the planner must be able to reason about the team profiles of executing cooperative tasks. In many existing planning/execution systems, a task passes beyond the purview of the planner once it has been committed to the executive. For a planner to support live task modification, it must be possible to relax this boundary, and instead delineate the planner's domain as the allocation of agents to all tasks, present or future. Although this is not difficult to do within the planner, supporting live task modification places significant constraints upon the executive, as will be

[1] As with the required response time of the planner, the frequency at which state updates arrive will vary between domains. In order to reduce the lag between when events occur and when the planner is apprised of them, the frequency of state updates should be rapid enough to ensure that several updates will arrive within the planner's minimum response time.

discussed in Section 7.4.

7.3.2 Representation

In this section, we discuss the representation in ASPEN and CASPER of the four capabilities that were identified in the previous section as necessary for live task modification:

1. Accept task state updates from the executive.
2. Calculate and apply changes to predicted durations.
3. Rapidly repair and optimize the near-term schedule.
4. Reason about team profiles of executing tasks.

The first is supported by nearly all planning systems applicable to multi-agent coordinated teams: any planner that is unable to receive information from its executive is poorly equipped to operate in a dynamic, uncertain world. In the CASPER architecture, the executive updates the planner's task parameters as state updates become available. The task parameters are part of ASPEN's Parameter Constraint Network (PCN), which encodes dependencies between parameters, and ensures that changes to a parameter are propagated throughout the network. The PCN is extremely powerful, and allows the specification of parameter dependencies on both an inter-task and intra-task basis. By using the state parameters as inputs to our duration prediction function, we ensure that the prediction is updated whenever a new state estimate arrives.

For example, Listing 7.1 contains a simplified form of ASPEN's representation of the *Transport* task. Here, the task has a single transporter role, with agent bounds of $[1, 3]$ (line 2). This version of *Transport* has two other state variables: the distance remaining to the goal (line 3) and the amount of work remaining to recover from a temporary failure, such as becoming mired in the sand (line 4). As execution proceeds, the executive updates these three state variables, as well as setting $curTime$ to the current time. The dependency network within *Transport* (lines 10-18) ensures that the duration prediction is recalculated whenever a state variable is updated ($pred\_dur$, line 14)[2]. The network then combines the new prediction with the time already spent on the task to update the task's overall duration.

Any conflicts introduced by new duration predictions, as well as any opportunities, must be addressed rapidly if the planner is to adjust the profiles of active

[2] $pred\_dur$ returns the mean of the predicted duration distribution, as ASPEN utilizes fixed-duration tasks.

Listing 7.1: A portion of ASPEN's representation of the Transport task.

```
activity Transport {
  real numTransporters = [1.0, 3.0, 1.0]; // Agent bounds are [1,3]
  real distanceRemaining = [0.0, 10000.0, 50.0];
  real glitchRecovery = [0.0, 10000.0, 0.0];
  int curTime = [0, 90000];
  int elapsedTime;
  int notDone;
  int remDur; // Remaining duration

  dependencies =
    notDone <− gt(distanceRemaining, 0.0),
    elapsedTime <− sub(curTime, start_time ),
    remDur <− mul(notDone,
                  pred_dur("Transport",
                           "NumTransporters", numTransporters,
                           "DistanceRemaining", distanceRemaining,
                           "GlitchRecovery", glitchRecovery )),
    duration <− sum(elapsedTime, remDur);
};
```

teams to good effect. The third and fourth capabilities thus are intertwined: it must be possible to repair and optimize the near-term schedule quickly, which will necessarily involve reasoning about changes to the team profiles of executing tasks.

In our experiments, we take a synchronous approach to simulation, allowing an arbitrary amount of time to repair the schedule (and perform a fixed number of optimization iterations)[3]. However, in a real-time domain, it is desirable to focus repair and optimization on the near-term portion of the schedule, as planning time is limited. ASPEN addresses this with the concept of *commitment* and *repair* windows. The commitment window specifies which tasks may not be modified by the planner, while the repair window specifies the set of tasks whose conflicts should be repaired. In an unmodified CASPER system, the commitment window is used to ensure the planner does not modify tasks about to start execution, while the length of the repair window scales the time needed to repair and the foresight of the planner.

[3]The problem of optimally balancing the amount of time spent planning against the lag in the planner's response loop introduced by a long plan cycle is a complex tradeoff, and a thorough investigation is beyond the scope of this thesis.

When live task modification is available, we dispense with the commitment window, as the planner is able to modify the team profile of executing tasks. It is able to do so by adjusting the duration of committed join tasks and the existence of uncommitted join tasks associated with committed cooperative tasks. In a real-time scenario, the responsiveness of the proactive replanner may be adjusted by changing the length of the repair window, varying the number of repair or optimization iterations performed per cycle, or varying the maximum time that repair or optimization may consume. Reducing any or all of these variables will result in a more responsive planner, although the resulting schedules often will be less efficient.

If live task modification is not enabled, committed tasks may not be modified in any way (beyond updates to their predicted durations), and all join tasks are committed when their associated cooperative task is committed.

CASPER and ASPEN's representation of task parameters allows task state updates to flow smoothly from the executive to the planner, where they trigger a recalculation of the affected task's duration prediction. Our representation of mutable teams dovetails with ASPEN's commitment and repair window concepts to allow our heuristics to repair any resulting conflicts easily or to seize new opportunities by modifying the team profiles of executing tasks, among other methods. With the proper selection of repair window size, ASPEN is a *fast* and *reactive* planner, which allows the proactive replanning system to make full use of the advantages afforded to it by live task modification.

7.4 Task Modification and the Executive

Live task modification places significant constraints on the executive of a proactive replanning system. In order to support live task modification, the executive must be capable of tracking the state of executing tasks; providing frequent state updates to the planner; accepting changes to the team profiles of executing tasks; and supporting mutable teams, as discussed in Section 6.4. In this work, we make use of a heavily modified version of the single-layer CASPER executive.

7.4.1 Required Executive Capabilities

The executive must be capable of tracking arbitrary state variables for each task instance, such as the distance remaining to the goal, motor temperature, or any other state that is relevant to predicting the remaining duration of the task. This implies that some level of the system architecture is able to measure the relevant state and transmit the measurements to the executive, if the executive is not collecting the

observations directly. In the three-layer planner/executive/behavioral architecture we utilize, the behavioral layer tracks the task's current state and relays it to the executive.

In addition to tracking the state of executing tasks, the executive must be able to relay updated state to the planner. The ideal update rate will be dependent on the speed of the planner, the rapidity with which the task state evolves, and the frequency at which state measurements become available. In order to accomplish this, the channel of communication between executive and planner must be more expressive than is commonly the case. Instead of providing relatively simple and infrequent task completion or failure notices, the executive must be able to transmit repeated, detailed messages that encapsulate an arbitrary set of state information.

The final requirement placed upon the executive by live task modification is the ability to accept changes to the team profiles of executing tasks. This is distinct from requiring that the executive be able to support team profiles (i.e. agents joining and leaving a team), which is discussed in Section 6.4. This is potentially the most problematic requirement, depending on the nature of the interaction between planner and executive. If the planner incrementally commits the team profile as execution reaches each agent's scheduled start time (e.g. the start time of a join task, in our representation), then the executive only needs to modify the end-point of an executing task, along with any constraints in which the end-point is involved. However, if the planner commits the entire team profile when the cooperative task starts, the executive must be able to retract or modify portions of the cooperative task: an agent that had been scheduled to participate may be removed from the team before it joins in order to address problems with another team.

By supporting live task modification, we remove some potential for the executive to locally optimize the schedule. When live task modification is not enabled, the entire team profile (e.g. the cooperative task and all join tasks) is committed to the executive as a unit. The executive may be able to make use of such firm knowledge of when additional agents are scheduled to arrive. For instance, it may be possible to arrange a set of transporter agents around the load so as to attenuate the effects of an agent joining the team, but this is only possible if the executive is informed of the scheduled addition. While we demonstrate the effectiveness of live task modification in Chapter 8, we have not evaluated the cost of removing such potential for optimization from the executive.

Supporting live task modification requires little additional complexity in the executive, beyond that required for mutable teams. The executive must be able to track executing tasks' state, provide rapid state updates to the planner, and accept changes to executing tasks. In exchange for a slight increase in complexity, a proactive replanning system that supports live task modification is significantly more flexible and robust than one that relies solely upon live duration prediction

and mutable teams.

7.4.2 Implementation

In this thesis, we make use of a heavily modified version of the single-layer, single-threaded CASPER executive, as well as a simulator of our design. In this instantiation of the CASPER architecture, the planner, executive, and simulator run in the same process. The planner performs a repair and optimization cycle, then commits tasks scheduled to start to the executive. The executive updates its task list, triggers one time step of simulation, updates its task state in response, and propagates the new state to the planner. This loop repeats until the scenario is complete. CASPER also supports a multi-threaded configuration, in which the executive and simulator run in a separate thread, allowing planning to occur in parallel. However, this thesis is focused on evaluating the potential utility of proactive replanning, and we chose to eliminate the variability inherent in a multi-threaded approach. Properly handling the issues of when to commit tasks and how to ensure that the planner is not reasoning about tasks that will have completed by the time any changes are sent to the executive is an active area of research, as discussed in Section 7.3.2.

The CASPER executive tracks the scheduled start and end times of tasks, monitors the values of the resource and state timelines, verifies that an agent is only executing a single task at a time, and ensures that a task is not considered complete until the simulation has reached a terminal state. We have implemented a simulator that stochastically simulates the execution of each task via a modified form of Augmented Transition Networks (Woods, 1970) (Woods, 1973), and have integrated it into the CASPER framework (see Section 8.1.3 for a detailed discussion of the simulator). The simulator explicitly tracks a variety of state variables, and makes a subset of them available to the executive. The CASPER executive is able to track the state of executing tasks by querying the simulator after each step of simulation.

In this architecture, providing the planner with updated task state values is simply a matter of setting the appropriate task parameters to the newly observed values. As discussed in Section 7.3.2, ASPEN's Parameter Constraint Network (PCN) then propagates the values as appropriate, triggering any needed recalculations. CASPER supports the setting of task parameters by the executive in both its single- and multi-threaded modes of operation.

Our representation of mutable teams (Section 6.3.2) requires the executive only to support the modification of the scheduled end point of a join task, rather than the more complex removal of a committed task discussed in Section 7.4.1. Since the executive maintains a flat schedule of task start and end points, this is easily supported.

7.5 Summary

The addition of live task modification to a proactive replanning framework greatly increases the responsiveness and capabilities of the system. Live task modification allows the planner to adjust executing tasks by transferring agents dynamically between them, in response to unexpected events that occur during execution. This enables the optimization and repair of the schedule in ways that are impossible without live task modification. While it requires a slightly more complex executive and a fast, reactive planner, the resulting system is capable of responding fluidly to the realities of execution in a manner not otherwise possible.

Chapter 8

Proactive Replanning

Proactive replanning is the prediction of problems (or opportunities) and their early resolution (or exploitation) by adjusting both the pending schedule and the tasks currently underway. It implies a much tighter connection between the planner and executive than has become the norm in planning and execution systems, and results in a system that is more flexible, reliable, and efficient than is otherwise possible. We have investigated three aspects of proactive replanning: live duration prediction, mutable teams, and live task modification. Each has been discussed in isolation in the preceding chapters; here, we examine how they form a complete proactive replanning system, and how we have incorporated them into the ASPEN/-CASPER framework to form a well-integrated, flexible, and efficient planning and execution system.

To collect the quantity of experimental data necessary to validate proactive replanning, we have developed a metric simulator built upon CASPER's simulation layer. The simulator, named ROBINSON[1], tracks the precise location of all agents, as well as the start and end points of all tasks. While the planner operates with a simple, discrete model of position, ROBINSON maintains a much more complex, continuous world model: while the planner's view of the world may be approximately accurate, there will necessarily be discrepancies that must be resolved during the process of execution, just as in the real world. Our modified CASPER executive ensures that agents are properly positioned before they are allowed to join a team.

Using our simulator, we have evaluated the effects of each of the feasible combinations of live duration prediction, mutable teams, and live task modification, to determine their individual efficacy and the manner in which they interact. These

[1]The Robinson map projection is based on tables of coordinates, akin to the tables of agent and task locations that our simulator tracks.

experiments have been performed in the same multi-agent, highly stochastic, lunar outpost construction scenario utilized in the evaluation of mutable teams in Chapter 6. The majority of tasks in this scenario include optional roles, with an overall objective of minimizing the makespan of the executed schedule. Schedules executed using proactive replanning are on average 11.5% shorter than those built and executed without proactive replanning. All three aspects of proactive replanning contribute to these gains: mutable teams provide a 5% reduction, while live task modification yields the remaining 6.5%, but only if live duration prediction is also enabled.

In this chapter, we will first examine the CASPER architecture and the extensions necessary to properly support proactive replanning. The integration of the various aspects of proactive replanning into ASPEN/CASPER is then discussed, and a summary of the heuristics used by ASPEN when performing proactive replanning is presented. In addition, we provide a detailed description of the use of several of our heuristics. Finally, we present the results of our experiments and discuss their implications.

8.1 Architecture

We have implemented proactive replanning by extending the CASPER execution system, which incorporates the ASPEN planner. The details of ASPEN and CASPER are discussed in Chapter 3. This section revisits the aspects of the CASPER architecture relevant to the integration of live duration prediction, mutable teams, and live task modification, as well as discusses the extensions to ASPEN and CASPER necessary to support proactive replanning. CASPER consists of planning, executive, and simulator/hardware interface layers (Fig. 8.1), each of which may communicate with the neighboring layer(s).

During the execution of a schedule, each of the three CASPER layers communicates with its neighbors, as diagrammed in Fig. 8.1. Prior to execution, the planner constructs and optimizes the initial plan. Once execution begins, tasks are dispatched from the planner to the executive as their start times are reached. The executive, in turn, creates appropriate task models within the simulator, which tracks the positions and state of all agents, teams, and goals. As the simulator's state advances, the executive provides task state updates to the planner, which is then able to repair or optimize the schedule as warranted, before proceeding to the next step of execution. Algorithm 8.1 details the high-level procedure followed during simulated execution.

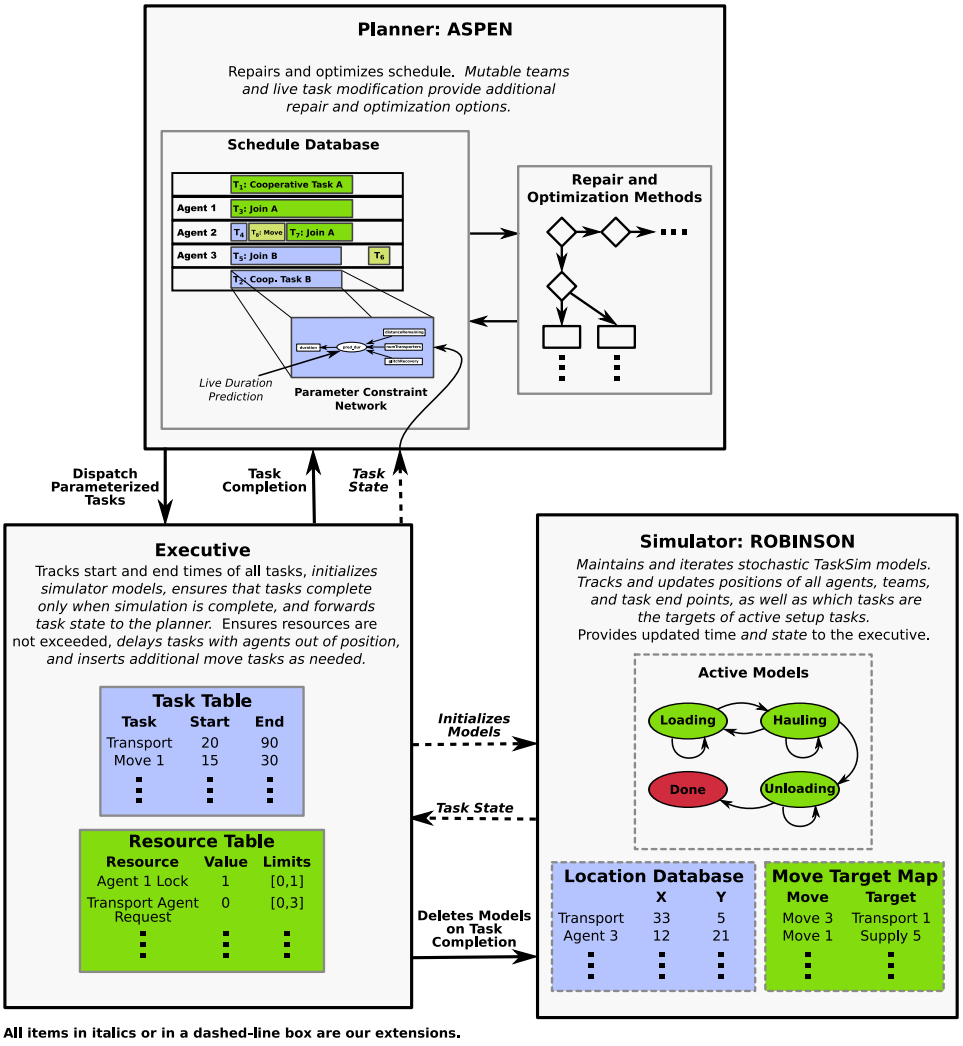


Figure 8.1: The structure and information flow of the CASPER planning and execution system, as extended to support proactive replanning. Our extensions are indicated in italics or dashed lines and boxes.

8.1.1 Planner: ASPEN

CASPER's planning layer is ASPEN: an iterative repair-oriented planner, which utilizes user-supplied heuristics to inform a set of repair and optimization methods. ASPEN and our extensions to it are described in detail in Section 3.1.

We have developed a suite of heuristics (summarized in Table 8.2) and methods designed to guide ASPEN's plan repair and optimization process so as to take advantage of the opportunities provided by proactive replanning. We examine our transfer method and the heuristics it utilizes in detail in Section 8.3.

8.1.2 Executive

As discussed in Section 3.2, CASPER provides a relatively simple embedded executive that we have extended to support our approach to proactive replanning. In the original executive, tasks deterministically complete when scheduled to do so, and the planner is informed only of task completion and any resource violations. We have extended the executive to provide updates to the planner as task state changes in the simulator, as well as to ensure that tasks do not complete until their simulation completes. In contrast to the basic executive, our extended version monitors the state of the our simulator (ROBINSON) for each task, ensuring that the planner does not believe the task to have completed until its simulation completes. Since simulation is stochastic, the lengths of executed tasks may differ significantly from their expected durations.

We have also extended the executive to support live task modification, by accepting changes in the duration of committed join tasks, which are not subject to live duration prediction[2]. When executing join tasks, the executive ensures that they complete at the specified time, although they may start late, due to the agent starting out of position. If live task modification is enabled, the executive updates the scheduled completion times of committed join tasks if and when the planner adjusts them.

Since ROBINSON maintains a much higher-fidelity model of the world than does the planner, the planner may commit tasks to the executive while the agents involved are somewhat out of position. To allow the executive to compensate for such problems, it is empowered to queue any tasks that have been committed for execution, but are not yet executable, as well as to insert additional repositioning tasks as necessary. Multi-agent tasks are queued until all agents assigned to the task at its start time are in position: that is, if a *Transport* task has two transporters assigned for its duration, with a third joining 10 time units into execution, the

[2] A join task that is scheduled to last until the end of its cooperative task will be adjusted appropriately as the expected duration of the cooperative task changes.

executive will queue the task until the first two agents are in place. If the third agent is delayed, execution of the task will continue with two agents until the third connects with the team. No movement task inserted by the executive is reported to the planner. Instead, the planner indirectly observes them through the lack of change in the state variables of the queued task(s).

8.1.3 Simulator: TaskSim and ROBINSON

The simulation layer used in this thesis consists of two distinct components: our TaskSim task modeling and simulation library, and ROBINSON, a heavily modified version of the CASPER simulator, which makes use of TaskSim to stochastically model task execution.

TaskSim

Task execution is stochastically modeled within ROBINSON using our TaskSim library, which is derived from Augmented Transition Networks (Woods, 1970) (Woods, 1973). The TaskSim models introduce a degree of uncertainty akin to that found in real-world robotic teams. They model the high- to mid-level details of a task, including non-terminal failures, and provide task-level state to the ROBINSON for relay to the executive. For instance, while individual components of agents (such as manipulators or sensors) are not simulated, events such as a portion of a load breaking loose are represented.

An Augmented Transition Network (ATN) is a finite state machine in which arcs have associated tests and effects. A single start and one or more stop states are specified, controlling where simulation begins and when it completes. An arc is followed only if the test evaluates to true, at which point the effects are applied. By convention, in our models exactly one edge's test condition(s) must evaluate to true in any given state and time[3], ensuring that the model is only in a single state at a time, unlike Petri Nets (Murata, 1989). Effects may modify the value of variables or invoke a submachine. This enables recursion, as well as making models significantly more modular. Our implementation supports the standard ATN semantics and extends ATNs by supporting random variables from a number of distributions (e.g. uniform and normal), as well as variable references. The values of random variables and variable references are reevaluated at every time step. These extensions to the ATN formalism allow us to introduce stochastic elements into our model and represent, to an extent, the uncertainty inherent in execution.

[3]In general, ATNs support effect-less self-transitions; that is, if no arc is true, the state does not change. In our models, the time variable must be updated as the model is stepped, to allow ROBINSON to determine when to cease stepping. As a result, we always define self-transitions.

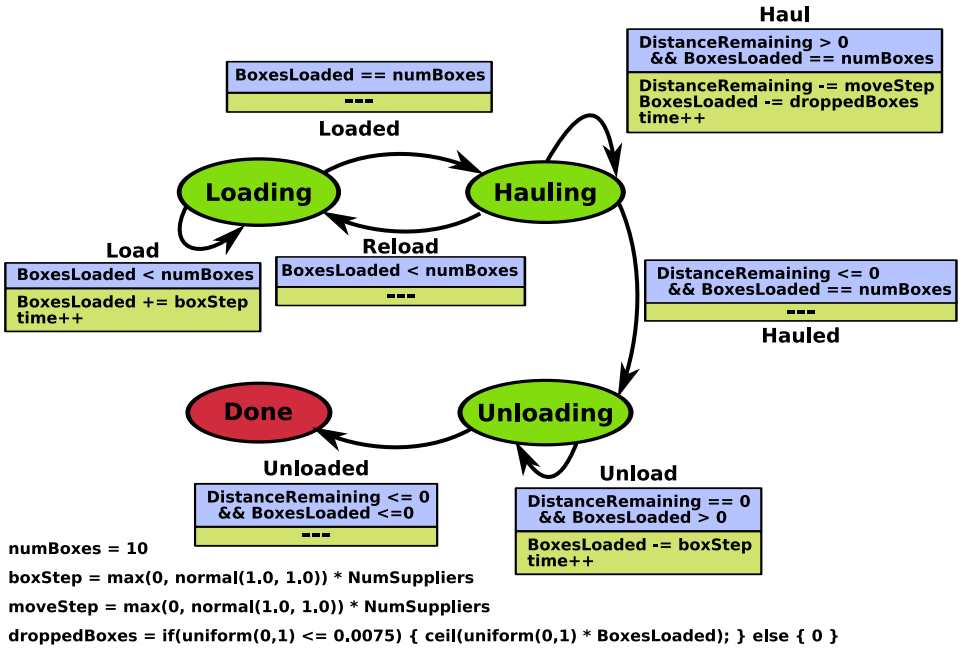


Figure 8.2: A graphical representation of the TaskSim model for *Supply Habitat*. The values of `moveStep` and `boxStep` are dependent on the number of agents assigned to the team. *normal(μ, σ)* returns a sample from a normal (Gaussian) distribution with mean μ and standard deviation σ. *uniform(a, b)* returns a random value uniformly distributed across the range $[a, b]$.

We will illustrate the capabilities of TaskSim by examining the model for the *Supply Habitat* task. A graphical representation of the model is presented in Fig. 8.2. The model definition for *Supply Habitat* and all other tasks are provided in Appendix B.2. The task consists of 1 - 3 agents loading a set of boxes at the lander site, transporting them to the vicinity of the habitat, and unloading them. During transit, it is possible for boxes to be dropped; if this occurs, the team must stop and reload them. The model has three states, corresponding to the loading, transit, and unloading actions. Filling the two optional roles speeds progress during all phases of the task. There are three state variables provided as arguments to the model: the number of agents assigned (`NumSuppliers`), the distance from the team's current location to the habitat (`DistanceRemaining`), and the number of boxes currently loaded (`BoxesLoaded`). As their values evolve, the executive updates the associated task parameters in the planner.

Variables internal to the task also may be defined, as well as when the variables

may be updated. The values of local variables are only updated by ROBINSON or by the effects of traversed edges, while the values of reference variables are updated at each step. Some references, such as `moveStep` and `boxStep` in Fig. 8.2, make use of the *uniform* or *normal* functions, which provide a new sample from a distribution of the appropriate type at each step. This functionality allows TaskSim models to execute in a stochastic fashion. In this example, the number of boxes loaded or meters moved during a given execution step (`boxStep` and `moveStep`, respectively) are functions of the number of assigned agents, with random deviations occurring.

Another reference is `droppedBoxes`, which contains the number of boxes that were dropped during the previous execution step. There is a 0.75% chance of a drop occurring on a given execution step, with a random number of boxes ranging from 1 to all the loaded boxes falling off if a drop occurs. Non-terminal failures such as this result in multi-modal duration distributions since the team needs to stop while re-collecting the dropped boxes, with each mode corresponding to a different total number of dropped boxes observed during an execution.

The task begins in the *Loading* state, where it remains until all boxes have been loaded onto the robot(s). At each step of execution, the tests associated with all arcs out of the current state are evaluated. The *effect* clause of the arc whose test evaluates to true is applied, and the state is updated to the value of the arc's *target* clause. For instance, the *Load* arc from the *Loading* state loops back to *Loading* until all the boxes are loaded. Note the special t_g variable: this represents (global) time. When the ROBINSON is executing an active TaskSim model, it transitions along arcs until t_g is equal to or greater than the current simulation time. This allows arcs to occur instantaneously (e.g. the *Loaded* arc instantly transitions from *Loading* to *Hauling*), or for a single arc to represent a significant passage of time.

Once loading is complete, the model transitions to the *Hauling* state, where the team begins moving to the habitat. $BoxesLoaded$ is updated at each step with the results of the $droppedBoxes$ calculations, and the model transitions back to the *Loading* state via the *Reload* arc if any boxes are dropped. While executing *Hauling*, the model updates its $DistanceRemaining$ parameter based on the progress made by the team on each time step. ROBINSON compares the previous and resulting values of $DistanceRemaining$ to determine how far the team has moved, updating the team's position in its world model as appropriate.

Once the team has arrived at the habitat, the boxes are unloaded in the same fashion as they were loaded. The model transitions to the *Done* state once unloading is complete (i.e. `BoxesLoaded = 0`), signaling the successful completion of the task.

Note that TaskSim models are location-agnostic: they have no representation of the metric location of agents or sites, operating strictly from their parameters. This

avoids much unnecessary complexity in the models, allowing them to be focused upon the specifics of the task, while the ROBINSON maintains a detailed world model. ROBINSON is responsible for providing inputs appropriate to the current state of the world. For instance, it will calculate the actual distance between the team and its goal to provide an accurate initial value for the *DistanceRemaining* parameter.

TaskSim models also have no representation of mutable teams, instead representing only the current state of the team. As agents arrive and depart from the team, the executive will update the appropriate parameters of the model (*NumSuppliers* in the case of *Supply Habitat*).

ROBINSON: An Extended CASPER Simulator

CASPER's default simulator is quite simple, supporting deterministic tasks and no explicit world model. It is intended as a starting point for a domain-specific simulator; we have extended it to provide one for our experimental scenarios, which we refer to as ROBINSON. ROBINSON maintains a TaskSim model for each executing task. When a step of execution occurs, arcs are traversed in each model until the model's time variable equals or exceeds the new global time. The resulting state is then provided to the executive, for relay to the planner.

In addition to stochastically modeling the execution of each task, the simulator maintains a detailed world model, which is used by the executive to determine when tasks must be delayed or when additional repositioning tasks will need to be inserted. The world model tracks:

- Fixed locations and sub-locations
- Agent locations
- Team locations
- Role locations for each team
- Start and end locations for each active cooperative task
- A mapping from active setup tasks (e.g. moves) to the cooperative tasks they are rendezvousing with.

All locations are represented as Cartesian coordinates. The three fixed locations correspond to the three discrete positions that the planner reasons about (*Lander*, *Habitat*, and *Comm*), while the sub-locations indicate the specific points near each discrete location at which different tasks take place. For instance, the four

Transport tasks all move from the *Lander* to the *Comm* site but, since all items being transported are not collocated, each originates and terminates in a different sub-location.

ROBINSON tracks the position of each agent, as well as the center of each team. The agent and team locations are updated as execution proceeds by determining the distance travelled during the last step, then moving the associated team or agent along a vector towards their goal. While doing so, the simulator caps agent velocity at a maximum of 2 meters per second, as a precaution against modeling errors.

Since robots occupy space, all members of a team cannot be collocated. Instead, each type of task defines a formation, with the locations of each role specified relative to the center of the team. When an agent joins a team or begins a setup (move) task prior to joining, the executive assigns it to a specific position. The agent is added to the team only when it reaches its assigned place. Existing agents do not need to relocate to accommodate new arrivals.

While calculating the movement vector of tasks with well-defined start and goal locations is simple, this is not the case with *Move* tasks intended to bring an agent into position to join a moving cooperative task after it has begun execution. ROBINSON takes the place of a low-level motion controller, and guides the moving agent to the correct position by assuming the target task will proceed at a constant velocity, then finding the earliest point in (x, y, t) space where the moving agent may meet the team. The simulator calculates the target task's expected velocity by querying the executive for the task's expected end time, then dividing by the distance between the team's current position and the location of its goal. ROBINSON recalculates the movement vector during each step of simulation, allowing it to compensate for inaccurate duration predictions.

8.1.4 Flow of Execution

We are utilizing the single-process variant of CASPER in the experiments reported here, which ensures synchronous execution of the steps outlined in Algorithm 8.1. By taking this route, we are able to ensure that the planner has sufficient time for plan repair and optimization after every step of execution. This approach factors out the difficult problem of determining how to ensure that the state the planner is reasoning about has not become excessively out of date. This is an area ripe for further work, although we note that we are able to construct, repair, optimize, and execute simulated schedules approximately 10 hours in length in less than 15 minutes. If the speed of execution is slow enough with respect to the speed of planning and optimization, the planner is easily kept up to date. The true challenge arises when planning operations and execution are occurring at equivalent speeds.

Algorithm 8.1 The flow of execution in the proactive replanning-enhanced ASPEN/CASPER system, while performing simulation and planning in the same thread.

1: **while** A task has not completed execution **do**
2: **Planner:** Dispatch tasks scheduled to begin execution at the current time step to the executive.
3: **Executive:** Initialize TaskSim models within the simulator for any new tasks that are executable and any queued tasks that have become executable. Queue new tasks whose agents are out of position, and create the requisite move tasks within the executive.
4: **Executive:** Update task parameters, such as the number of assigned agents, in the simulator from the planner.
5: **Simulator:** Step the simulation of the TaskSim models associated with all active tasks.
6: **Simulator:** Update the world model based on the changes in the TaskSim models.
7: **Executive:** Transmit updated task state from the simulator to the planner, updating parameters within each task's Parameter Constraint Network.
8: **Planner:** If live duration prediction is enabled, update the duration predictions for all executing tasks whose state has changed.
9: **Planner:** Repair any conflicts in the schedule. The enabling of mutable teams and/or live task modification affect the available set of choices during repair.
10: **Planner:** Perform 5 iterations of optimization, repairing any resulting conflicts after each iteration. Again, the enabling of mutable teams and/or live task modification will affect the available menu of choices.
11: **end while**

8.2 Heuristics

In addition to extending ASPEN and CASPER to support the various components of proactive replanning, we developed a large suite of heuristics that guide repair and optimization. These heuristics allow the planner to efficiently take advantage of the possibilities afforded by mutable teams, optional roles, and live task modification without being overcome by the combinatorics of this complex problem. There are necessarily slight variations in the optimization heuristics between the scenario presented in Section 5.7.2, where the objective is to maximize reward within a fixed planning horizon, and the scenario discussed in Section 6.6.2, where the objective is to minimize the makespan of a specified set of tasks. However, the

heuristics used in all of our experiments remained substantially unchanged.

There are five themes that are incorporated into the majority of the heuristics in our suite:

- Use stochasticity.
- Utilize predicates to recognize common configurations.
- Make simplifying assumptions and choices.
- Minimize the impact of schedule changes.
- Utilize post-processing.

While none of these techniques are unique to this work, their collective use allowed our proactive replanner to efficiently operate in a combinatoric search space.

8.2.1 Stochasticity

The use of stochasticity is prevalent in ASPEN: many of ASPEN's default heuristics make extensive use of weighted randomized selection when choosing among a set of options. We continued and expanded this use of stochasticity throughout our heuristic suite. Nearly all decisions made by the planner involve a random element, with weights assigned as appropriate to the scenario. For instance, when selecting an initial start time for a cooperative task, we weight candidate intervals of time by the number of idle agents available. This increases the likelihood that the planner will be able to fill the task's required roles quickly, with minimal disruption to the remainder of the schedule. Similarly, when determining which agent should be added to a team, our heuristics incorporate each agent's travel time and temporally proximate commitments into the weights used to randomly select an agent.

By incorporating a random component into our heuristics, we are able to include many possible approaches in our decision making, even if a number of them are only rarely applicable. By assigning lower weights to the less general approaches, they are only sporadically used. However, if the planner reaches a local minimum or a particularly difficult portion of the search space, it will eventually select the applicable specialized (or excessively general) approach. For example, when selecting the start time for a new task, our system randomly selects a time from the entire planning horizon 1% of the time.

8.2.2 Predicates

Such an approach is clearly inefficient, as much computation will be wasted in situations where a specialized, low-probability tactic is needed. We are able to alleviate much of this inefficiency by using predicates to recognize common configurations and by adjusting our weighting schemes to bias the search towards methods known to be effective in similar states, or appropriate to the planner's current goals. A set of predicate functions are invoked at a number of points throughout our heuristic suite, and are used to further focus the planner's search.

For instance, when optimizing a schedule during execution, opportunities that occur earlier in the schedule receive greater weight when the planner is selecting the next optimization method. This biases the limited amount of optimization available towards the portion of the schedule that is affected the most by the effects of execution and has the least time remaining in which to be optimized.

In some cases, the matching of a predicate drives a particular course of action in a non-discretionary fashion. During makespan optimization, if there is any slack in the critical path, the current iteration of optimization is used to pack the schedule (Section 3.1.3), since the makespan is guaranteed to be reduced. Table 8.1 summarizes the more common predicates used throughout the heuristic suite.

8.2.3 Simplifying Assumptions

The third theme common to our heuristics is their use of simplifying assumptions and choices. Due to the free placement of tasks on the schedule and the effectively unlimited number of join tasks that may be associated with a given cooperative task, the complete tree of potential schedules is vast. In order to address these combinatorics, our heuristics make a variety of simplifying assumptions that restrict the set of options considered, relying on further repair or optimization to tune the schedule. For example, when evaluating how much time to allocate to a setup task that must intercept a moving team, we assume constant velocities for all participants, rather than trying to model the moving team's execution.

8.2.4 Minimize Impact

We have endeavored to *minimize the impact of schedule changes*. During our initial experiments, we found that a single poor decision during schedule repair often resulted in a cascading series of conflicts that affected large portions of the schedule. This expanded the initial, localized repair attempt into a recalculation of large portions of the schedule. Such cascades were particularly common when repairing conflicts in an optimized schedule, where little slack remained. Our iterative deepening approach to repair and our use of limited backtracking (Section 3.1.3) were

Table 8.1: A selection of the predicates utilized in the heuristic suite, and a summary of what they examine.

Predicate	**Summary**
Calculate Slack	Calculate the critical path's slack. If the slack is non-zero during optimization, a schedule pack is performed (Section 3.1.3).
Find Useless Moves	Searches for movement tasks that have no effect. They are then deleted to simplify the schedule.
Repairing / Optimizing	Some general heuristics are biased differently based on whether repair or optimization is underway.
Executing	If execution is underway, many heuristics are biased towards operating upon tasks that will soon begin execution.
Task Type	The type of a task affects the operations that may be performed, and which may be most effective. Types include: join, cooperative, solo, and move.
Conflict Type	A variety of predicates evaluate details of the current conflict, including whether it is an agent oversubscription, an empty required role, a violated ordering constraint, a short conflict, a setup task overlapping its target, or overlapping state reservations.
Preference Type	A variety of predicates evaluate details of the current optimization preferences, such as whether the planner is optimizing for minimal makespan or maximal reward.
Timeline Usage	Calculates the percentage of a timeline that contains activities. This is used to guide new tasks towards agents that are underutilized.
Missed Chances	Calculates the sum of the length of open optional roles on critical-path tasks during which there are agents idle or filling optional roles in non-critical-path tasks. Used to bias optimization towards or away from transferring agents into critical-path tasks.
Agent Flexibility	Calculates the longest interval that an agent is free within a specified interval. Includes any adjacent free intervals, and serves as a measure of how much the use of a given agent will constrain a cooperative task. Used to weight agents when filling a role slot.

designed to limit conflict cascades and to partially localize the effects of repair and optimization.

The theme of minimizing impact reoccurs throughout many of our heuristics. For example, when moving a cooperative task to resolve a conflict, our heuristics prefer to move it by no longer than needed for an agent to move to the task's location, and no more than the length of the task. This ensures that the task is moved far enough that an agent may be transferred in, if necessary, but not so far that it will create many temporal or resource conflicts. When placing new cooperative tasks, the planner prefers portions of the schedule where more agents are idle. As mentioned above, this increases the likelihood that the task's required roles may be filled without affecting other tasks.

8.2.5 Post-Processing

In addition to the heuristics guiding the planner's search, we *utilize post-processing* extensively. A variety of post-processing routines are run periodically, in order to apply simplifying or optimizing operations to the schedule. These operations are able to consider a more global context, and operate upon larger combinations of tasks than is computationally feasible in the heuristics that are executed more frequently. A simple example is the removal of useless move tasks: any moves that either have the same start and end locations, or have no tasks dependent upon the move task's effects, are deleted. This reduces the complexity of the schedule, and increases the efficiency of the planner.

A more complex instance is the minimization of the number of join tasks. Many of our heuristics add additional join tasks, or modify existing ones. As repair or optimization proceeds, their number steadily increases, making further operations more complex. To counteract this growth, we periodically attempt to minimize the number of join tasks, while not creating any new conflicts. We make use of four strategies to do so: merging, subsumption, extension, and the removal of "cross-fades". If a given agent has multiple join tasks for a role in a given cooperative task, and the agent has no intervening commitments, the join tasks are merged into a single join. If a join may be extended to subsume a second join, the second join is deleted and the first extended. For example, suppose agent 1 is filling a role for the first half of the task, while agent 2 fills the role for the second half (Fig. 8.3(a)). If agent 1 is available during the entire cooperative task, agent 2's join task will removed, and agent 1's will be extended to cover the entire cooperative task (Fig. 8.3(b)). The third strategy is to extend all joins as far as possible, subject to task and agent constraints, in order to maximize the amount of time optional roles are filled. Finally, "cross-fades" are removed. A cross-fade is the assignment of two agents to two different roles within a cooperative task,

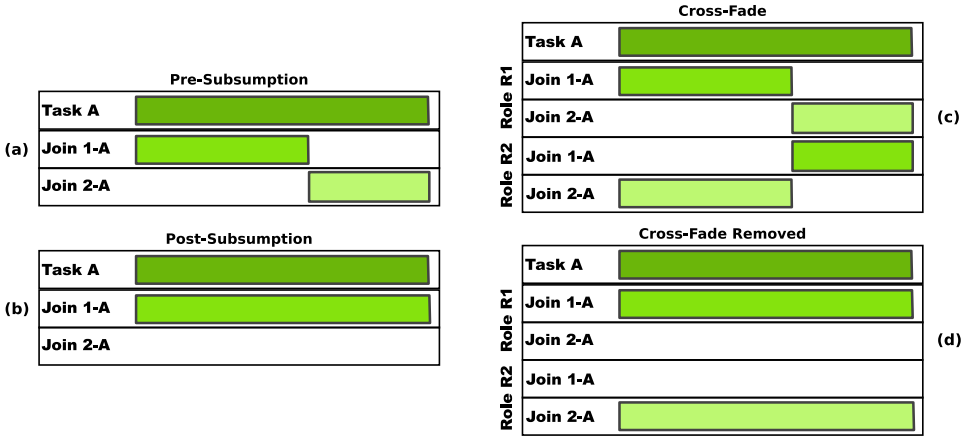


Figure 8.3: Two simplifying operations that may be carried out upon cooperative tasks. In (a) and (b), a join is extended to subsume another, while in (c) and (d) roles are rearranged to reduce the total number of join tasks.

then swapping the assigned roles (Fig. 8.3(c)). The schedule may be simplified by assigning each to a single role (Fig. 8.3(d)).

These five themes have informed the design of our heuristics, and are the underpinnings of the entire suite, which is summarized in Table 8.2. The summaries are necessarily terse, and do not address many special cases: the heuristic suite is comprised of over 6,500 lines of code, with 13,000 lines of supporting libraries. Repair and optimization methods included in the generic version of ASPEN that did not require any proactive replanning-specific heuristics have been omitted.

8.3 Transferring Agents

To further illustrate our heuristic suite, we will detail our *Transfer* repair and optimization method, and discuss the use of heuristics throughout. The complete suite is summarized in Table 8.2.

The *Transfer* method reassigns an agent from one team to another, while inserting any necessary setup tasks to reposition the agent appropriately. It is of use during both schedule repair and optimization. Transferring frequently will reduce the duration of the task receiving an additional agent, which can resolve certain temporal or resource conflicts during repair operations. During optimization, agents may be transferred into tasks on the critical path that have open optional roles, thus shrinking the critical path and the schedule's makespan. Note that the *Transfer* method exclusively addresses the movement of an agent from one team

Table 8.2: Summaries of the heuristic suite used during schedule repair and optimization. Note that choice points are listed chronologically as they occur in each method.

Method	Choice Point	Heuristic Summary
Repair: Move	Culprit To Move	This is highly dependent on the specific conflict being resolved. For instance, if resolving an agent oversubscription conflict, prefer moving non-join tasks, as they will have a smaller impact on the remainder of the schedule.
	Duration	For non-join tasks, return the predicted duration. For join tasks, return the longest interval during which the role is available and the agent is free.
	Valid Interval	Select the longest interval during which the role is available and the agent is free.
	Start Time	Usually return the earliest time in the valid start time interval; infrequently, return a random time from the interval.
Repair: Add	Task Schema to Add	Build a list of tasks that could affect the current conflict, which will vary with the type of conflict. For instance, if an agent is out of position, we prefer adding a move task or a join task that relocates the agent, while agent request conflicts are best solved with a join for the appropriate role. Randomly select a task from the list according to the flexibility of the agent.
	Culprit to Delete	If the task being added will make another redundant, delete the redundant task. This may happen when a join for a moving task is added.
	Duration	For non-join tasks, return the predicted duration. For join tasks, return the longest interval during which the role is available and the agent is free.

Continued on Next Page...

Table 8.2 – Continued

Method	Choice Point	Heuristic Summary
	Valid Interval	This is dependent upon the conflict being repaired. For out-of-position conflicts, the move task will be inserted just before the task in need of a different position, while during repair of an overlap near the end of a cooperative task, the longest interval for an optional join is preferred.
	Start Time	When adding moves, prefer the latest start time, to minimize the delay between the end of the move and the start of the task that needs the resulting position. For most other situations, prefer earlier start times to maximize the length of the join. If a new cooperative task is being added, weight the available start times by the number of agents idle (and thus available to the task) at each time.
Repair: Delete	Culprit to Delete	If one of the contributing tasks is a useless move (e.g. it moves nowhere), delete it. Otherwise, weight the tasks by the likelihood that their removal will resolve the conflict without creating additional conflicts. Randomly select a task according to the assigned weights.
Repair: Connect	Constraint	Randomly select an open constraint to connect. This is used to re-establish broken links between join and cooperative tasks.
	Culprit to Connect	Given a join, select a cooperative task that overlaps the join. If no such task with an appropriate open role slot exists, select the nearest cooperative task with an appropriate slot open.
Repair: Move and Connect	—	This method concatenates the Connect and Move methods, ensuring that the same task is moved as is connected to.

Continued on Next Page...

Table 8.2 – Continued

Method	Choice Point	Heuristic Summary
Repair: Transfer	Task Schema to Add	Select a cooperative task from the tasks contributing to the conflict, preferring tasks that start prior to the conflict and have open optional roles.
	Culprit to Change Duration	Select a donor join that overlaps the recipient task, preferring agents that have few adjoining commitments.
	Task Schema to Add	Determine if a setup task is necessary, by examining the transferring agent's position timeline. If so, select the setup task, as well as its start and goal positions.
	Culprit to Move	If a setup task with the correct goal is available during the relevant timespan, move it into position, rather than adding a new setup task.
	Culprit to Delete	Find any move tasks that overlap or are adjacent to the new join, and have the same final position. They are now redundant, and may be deleted safely.
	Duration	For setup tasks, return the predicted duration for moving from the agent's current position to that of the recipient task. For the new join task, return the length of the interval during which the optional role is open immediately following the setup task, less any overlap by the setup task.
	Valid Interval	For setup tasks, select the earliest interval such that the setup task completes as close to the start of the longest interval available for the new join task. For join tasks, return the interval immediately following the setup task.
	Start Time	Return the earliest time from the valid interval.

Continued on Next Page...

Table 8.2 – Continued

Method	Choice Point	Heuristic Summary
Repair: Add and Setup	—	This method and associated heuristics are identical to the *Transfer* repair method, with the omission of the donor task, and hence the *Culprit to Change Duration* choice point.
Repair: Right Shift	Culprit To Move	Select the latest task involved in the current conflict. In addition, calculate the set of successor activities to shift by following inter-task constraints into the future, ignoring any constraint with a slack greater than the length of the conflict.
	Start Time	Select a start time late enough to move the selected task past the end of the current conflict.
Optimize: Move	Culprit To Move	For 50% of move optimizations, randomly select a task on the critical path with non-zero slack to pack, weighting the tasks by the amount of slack. For the remainder of optimizations, attempt to extend an optional join for a cooperative task on the critical path. Weight the options by the amount that each optional join may be expanded.
	Valid Interval	Return the interval of valid start times that includes the task's current start time.
	Start Time	Select the earliest start time in the valid interval.
Optimize: Delete	Culprit to Delete	Select a useless move for deletion. A move is useful if and only if: (1) it changes the value of its associated position timeline, (2) a task follows the move that makes use of resulting position, and (3) the move is not immediately preceded and succeeded by joins for the same cooperative task.

Continued on Next Page...

Table 8.2 – Continued

Method	Choice Point	Heuristic Summary
Optimize: Change Duration	Culprit to Change Duration	Select an optional join on the critical path whose end time may be extended: that is, the role is open following the end of the join, and the agent is free. Randomly select from the options, weighting by the degree by which each may be extended.
	Duration	Select the maximum duration that will not introduce a conflict.
Optimize: Pack	—	There are no choice points when packing a schedule; it is a deterministic operation, as described in Section 3.1.3.
Optimize: Transfer	Task Schema to Add	Randomly select a cooperative task from the critical path, weighting by the inverse of each task's slack.
	Culprit to Change Duration	Select a donor join that overlaps the recipient task, preferring agents that have few adjoining commitments.
	Task Schema to Add	Determine if a setup task is necessary, by examining the transferring agent's position timeline. If so, select the setup task, as well as its start and goal positions.
	Culprit to Move	If a setup task with the correct goal is available during the relevant timespan, move it into position, rather than adding a new setup task.
	Culprit to Delete	Find any move tasks that overlap or are adjacent to the new join, and have the same final position. They are now redundant, and may be deleted safely.

Continued on Next Page...

Table 8.2 – Continued

Method	Choice Point	Heuristic Summary
	Duration	For setup tasks, return the predicted duration for moving from the agent's current position to that of the recipient task. For the new join task, return the length of the interval during which the optional role is open immediately adjacent to the setup task, less any overlap by the setup task.
	Valid Interval	For setup tasks, select the earliest interval such that the setup task completes as close to the start of the longest interval available for the new join task. For join tasks, return the interval immediately following the setup task.
	Start Time	Return the earliest time from the valid interval.
Optimize: Add and Setup	—	This method and associated heuristics are identical to the *Transfer* optimization method, with the omission of the donor task, and hence the *Culprit to Change Duration* choice point.
Optimize: Swap	Culprit to Delete	Select a join that is at least partially responsible for the associated cooperative task having zero slack. If the cooperative task has zero slack due to temporal constraints on the cooperative task, do not consider any joins associated with it. Randomly select one of the joins, weighting by the likelihood that assigning the join to another agent will result in the cooperative task having non-zero slack.
	Task Schema to Add	Randomly select an agent to assume control of the join, weighting by the amount of time the agent is free prior to the start of the join.

Continued on Next Page...

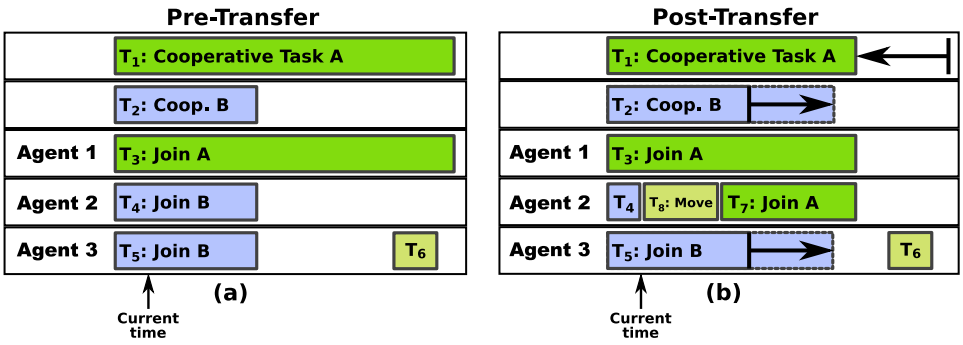


Figure 8.4: Transferring agent 2 from its optional role in T_2 to an optional role in T_1 reduces the schedule's makespan. Doing so requires shortening the donor join task T_4, inserting a setup task T_8, and inserting the new join task T_7.

Table 8.2 – Continued

Method	Choice Point	Heuristic Summary
	Culprit to Delete	If the selected agent is currently performing a join for a different cooperative task, return it: the agent performing the first join will assume control of the second.
	Task Schema to Add	Return the task type needed for the agent from the first join to replace the second join.

to another, while *Add and Setup* enables the addition of an otherwise idle agent to a team, along with any necessary setup tasks. This somewhat artificial distinction simplifies our repair and optimization method selection heuristics.

The *Transfer* method is outlined in Algorithm 8.2; we will illustrate its use and discuss the heuristics involved by examining the example transfer depicted in Fig. 8.4. In Fig. 8.4(a), two cooperative tasks are executing in parallel: T_1 and T_2. In this example, an agent will be transferred from T_2 to T_1 in order to reduce the schedule's makespan. If at any point in the process a selection cannot be made (e.g. no donor tasks are available, or all optional role openings are too short), the method fails, and the schedule is reverted to its state prior to the start of the transfer method.

Algorithm 8.2 A simplified form of the *Transfer* repair and optimization method. The complete method also determines if an existing setup task may be moved, rather than adding a new one, but this logic is omitted for clarity. Heuristics are invoked on each line that begins with "select".

```
1:  function transferAgent() do
2:      If any step fails, revert to the status quo and return.
3:      Select the recipient team.
4:      Select the donor join task (and thus the donor team).
5:      Select a setup task to add, if needed.
6:      Select a task to delete, if any tasks will become redundant.
7:      Lift any redundant tasks from the schedule.
8:      Lift the donor join task from the schedule.
9:      If needed, add a setup task, selecting its duration and start time.
10:     Add the appropriate join task to the recipient team, selecting its duration and
        start time.
11:     Shrink and replace, or delete, the donor join task.
12:     Delete any redundant tasks.
13:     Propagate the Parameter Constraint Network.
14: end function
```

The first step of transferring is to select the team to which an agent will be added; we refer to this team/task as the *recipient*. To do so, the planner invokes the *Task Schema to Add* choice point, which in turn executes one of our heuristics. During optimization, this heuristic examines one of the schedule's critical paths (selected randomly from the set of all critical paths), recording all cooperative tasks that are either directly on the critical path, or which have a join task on the critical path. Each cooperative task is weighted by the longest optional role slot available and the inverse of its slack: we prefer to add agents to tasks that have little slack otherwise. A cooperative task then is selected randomly from the list, according to the calculated weights. This stochastic approach is used throughout our suite of heuristics, allowing the search to be guided towards promising possibilities, while enabling the occasional selection of less-promising options. This helps to ensure that the space of possible solutions is explored over time. In Fig. 8.4(a), the critical path consists of a single cooperative task (T_1), greatly simplifying the selection of the recipient task. During repair, a cooperative task is selected from among those tasks contributing to the current conflict, with preference given to tasks that start prior to the conflict and have open optional roles. Transferring agents to such tasks may reduce the task's duration sufficiently to resolve the conflict.

Once the recipient is identified, the planner must find another cooperative task that may donate an agent, invoking the *Culprit to Change Duration* choice point, and another of our heuristics. Here, we are searching for a join task that may be shortened (or deleted) to allow its agent to transfer to the recipient. We consider all optional join tasks in the schedule that overlap the recipient task, but are not part of a cooperative task that is on the critical path. Each potential donor join task is weighted by the agent's *flexibility*. For these purposes, we define flexibility as the length of an interval containing the potential donor join task that is otherwise free of commitments. A donor join task then is selected randomly from the list, according to the calculated weights. By weighting according to flexibility, we prefer agents that will introduce the fewest additional constraints on the recipient task: this eases further optimization or repair. In Fig. 8.4(a), if T_4 and T_5 were both filling optional roles, T_4 would receive a heavier weight, as T_6 reduces the flexibility of agent 3.

Once a donor join task (and thus agent) and recipient task have been identified, the planner is able to determine the type of join task that will add the transferring agent to the recipient task in the recipient's open optional role. If the recipient has more than one open optional role, a role is selected randomly by weighting them according to the length of time that each role is unfilled, less any existing commitments for the transferring agent[4]. When transferring, we would prefer to add the new agent for a significant length of time, rather than in a small gap between two other join tasks. If a role is available for less than 5% of the duration of the cooperative task, it is ignored: on average, such small join tasks result in greater constraints on the cooperative task than the small reduction in duration warrants.

With the donor and recipient identified, the planner is able to evaluate whether a setup task will need to be added to the schedule, via another invocation of the *Task Schema to Add* choice point and associated heuristics. In our domains, the only required setup actions are those of movement that serve to position the agent properly. The schedule contains a position timeline for each agent, which tracks the current position of the agent throughout the planning horizon (see Section 3.1.3). By comparing the position requirements of the new join task with the expected position of the agent during the recipient task, the planner determines if a setup *Move* task must be added and, if so, its start and goal positions[5]. In Fig. 8.4(b), a *Move* task (T_8) proved to be necessary.

Once all the components of the transfer have been identified, the planner eval-

[4]Note that in the domains presented here, no cooperative task has more than one type of optional role. If multiple role types are available within a team, and they have differing effects on the cooperative task's duration, the roles should instead be weighted by the effect on the task's duration of adding an agent to the role for the available length of time.

[5]The complete *Transfer* method also considers whether an existing *Move* may be repurposed, or if an additional task must be added. These calculations are omitted for brevity and clarity.

uates if any tasks will become redundant, invoking another of our heuristics via the *Culprit to Delete* choice point. Tasks may become redundant when the recipient task is moving from one location to another; joining the recipient may fortuitously move the agent into position for its next task, eliminating the need for an existing *Move*. Any moves that overlap or are adjacent to the new join and have the same final position as the new join will be deleted.

At this point, all of the actors in the transfer have been identified. In Fig. 8.4, the recipient task is T_1, the donor join task is T_4, the donor team is T_2, a setup task will need to be added (T_8), and the new join is of a type appropriate to add agent 2 to an optional role in a cooperative task of type A. In preparation for the addition of the setup and join task, the planner temporarily removes all reservations by the donor join task and any redundant setup tasks from the schedule.

If a new setup task is necessary, it is added now (T_8 in Fig. 8.4(b)). To do this, the planner must select its duration and start time, utilizing the *Duration*, *Valid Interval*, and *Start Time* choice points. The duration of *Move* tasks is not adjustable by the planner: instead, it is controlled by our duration prediction algorithms, which take into account the agent's current position and the position needed to join the recipient task. The setup task is placed such that it completes just prior to the longest interval in which the new join task may be placed and starts no earlier than the beginning of the donor join task (or the current time). If insufficient time is available between the beginning of the donor join task and the start of the new join task, the setup task is placed as early as possible, maximizing the length of time the transferring agent may participate in the recipient task. These calculations become somewhat complex if the recipient task is moving, as the length of the setup task affects the location of the recipient team when the setup completes. In Fig. 8.4(b), the setup task T_8 is scheduled to begin at the current time.

Once the setup task, if any, is in place, the planner is free to add the task representing the transferring agent joining the recipient task (T_7 in Fig. 8.4(b)). The planner again invokes the *Duration*, *Valid Interval*, and *Start Time* choice points (and thus the relevant domain-specific heuristics) to determine the new join task's duration and start time. These heuristics set the new join task's duration to be equal to the length of the role slot available in the recipient task, less any overlap by the setup task and other prior commitments of the agent. The start time is selected in a similar fashion. In this instance, the selection of the task's duration and start time is a straightforward calculation that is performed within our heuristic functions as a matter of convenience. The choice of duration and start time is significantly more complicated within other repair and optimization methods.

Finally, the donor join task's duration is reduced to make room for the setup task (or the new join, if no setup was required), and it is placed back onto the schedule. In Fig. 8.4(b), T_4 is shortened to end at the current time. If the setup

completely overlaps the donor join task, the donor is deleted entirely. Any redundant setup tasks are deleted at this point, and the Parameter Constraint Network is propagated, resulting in the calculation of new duration predictions for the affected cooperative tasks. The cooperative task (T_2) associated with the donor join task (T_4) will increase in duration, while the recipient (T_1) will shrink. The increase in duration of the donating cooperative task may result in conflicts, but they normally will not involve the critical path, allowing the planner to resolve them while maintaining the new, shorter makespan.

Note that it is theoretically possible for the planner to oscillate: if the increase in T_2's duration due to the removal of agent 2 were sufficient to place it on the critical path, the planner may decide during the next iteration of optimization to transfer the agent back to T_2, since ASPEN's iterative approach is memoryless. In practice, we have found that the stochastic nature of our heuristics, and ASPEN's stochastic approach to optimization as a whole, is sufficient to avoid such looping.

8.4 Experimental Results

We have performed an experiment in simulation to evaluate the effects of live duration prediction, mutable teams, and live task modification, both in isolation and in different combinations, to gain insight into how the different aspects of proactive replanning interact. We have evaluated six different combinations of the three components of proactive replanning as to their effect on the executed schedules' makespan, as well as several other metrics. Through the use of proactive replanning, we are able to execute schedules on average 11.5% (88.7 minutes) shorter than otherwise possible, with mutable teams and live task modification contributing equally to the improvements. While live duration prediction did not directly reduce makespan in this domain, it enabled live task modification's effect.

8.4.1 Scenario

These experiments were performed in the context of the same scenario described in Section 6.6.2. This domain represents a series of activities that might occur shortly after a lunar landing, as part of the preparations for establishing a base. The tasks include the construction of four communications arrays, laying cable from the array farm to the habitat site, and transporting supplies to the habitat, where they are stowed. Five homogeneous agents are available, three distinct locations are utilized, and eight classes of tasks must be performed. Some groupings of tasks must be performed serially (e.g. the components for the communications arrays must be transported from the lander before they can be assembled), while

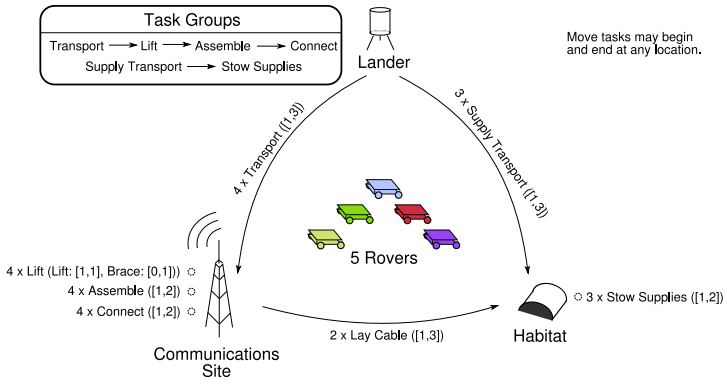


Figure 8.5: The CommTower scenario. The repeat count for each task precedes it, while the agent bounds for each role follow. The evaluation and optimization metric for this scenario is the makespan of the schedule. This duplicates Fig. 6.18.

others may be performed at any time (e.g. laying cable from the communications site to the habitat). All cooperative tasks include at least one optional role. Most tasks must be performed multiple times, with a total of 24 cooperative tasks to be scheduled. Due to repositioning tasks and our representation of mutable teams, the average number of tasks on the executed schedule varies from 83.7 to 92.1 between experimental conditions. Each agent may participate in only one task at a time. The tasks vary with respect to their starting and ending locations, duration, available roles, and the effect of filling optional roles. See Fig. 8.5 and Section 6.6.2 (especially Table 6.2) for further details about the tasks and the planner's representation of location.

The objective of the scenario is to minimize the executed schedule's makespan, while accomplishing the specified set of 24 cooperative tasks. The planner may add additional *Move* tasks as necessary, and is free to schedule the tasks in any order, subject to inter-task temporal constraints. The scheduling horizon is long enough so as to be effectively infinite: the nominal lengths of the tasks vary from 20 minutes to three hours, with a 72-hour scheduling horizon. Agents may move no faster than 2 meters per second when moving unburdened. The three sites are separated by 40 - 50 meters and are arranged in an approximately equilateral triangle.

8.4.2 Experimental Design

Common Elements

The high-level flow of a single experimental run, starting with a blank schedule and a set of cooperative tasks that must be performed, is to:

1. Iteratively repair the schedule until it is valid (that is, all specified cooperative tasks have been placed on the schedule, sufficient agents have been assigned to each task, and no conflicts exist).

2. Perform 300 iterations of optimization, repairing any conflicts via iterative deepening repair (Algorithm 3.5) after each iteration of optimization.

3. Execute the resulting optimized schedule. After each step of execution:

   (a) Repair any conflicts via iterative deepening repair.

   (b) Perform 5 iterations of optimization, repairing any conflicts after each iteration via iterative deepening repair.

In all experimental conditions, duration prediction is performed for all unexecuting tasks, regardless of whether *live* duration prediction is in effect. This is necessary to provide principled durations for upcoming tasks, especially those involving mutable teams: there is no valid way to provide fixed durations for tasks utilizing mutable teams, as the team profile may differ significantly between task instances. As a result, even the baseline (no live duration prediction, mutable teams, or live task modification) will incur some computational costs due to prediction of durations for unexecuting tasks.

Note that it is not possible to experimentally evaluate the effect of adding optional roles to tasks without artificially handicapping the baseline (no-optional-roles) condition. Clearly, if the baseline were to eliminate all optional roles, it would be possible for the optional role condition to execute the tasks faster. If the baseline instead required that all (or a specified subset) of the optional roles were filled, the tasks would require more resources than strictly necessary, reducing the degree to which tasks could be executed in parallel. As a result, all of the experimental conditions below incorporate optional roles. If mutable teams are enabled, agents may be committed to a portion of the optional (and required) roles, rather than the entirety of the role.

Varying Live Duration Prediction

When live duration prediction is enabled, duration predictions are performed as tasks execute and their state is updated, as described in Section 5.5. When live duration prediction is disabled, the prediction function instead returns the time between the task's current end point and the current time. When input into the Parameter Constraint Network, this is combined with the time elapsed to return the task's current duration. If the task completes prior to its expected end time, the executive sets a completion parameter, instantly truncating the task's duration.

If the task overruns, the executive increments the task's duration by three time units, ensuring that the planner does not believe the task to have finished. The overrunning task's duration will be repeatedly incremented until it completes.

Varying Mutable Teams

When mutable teams are enabled, the planner is allowed to assign agents to roles for only a portion of the task's duration. Required roles still must be completely filled, although multiple agents may perform different segments of a single required role. See Section 6.3.2 for complete details on the representation and integration of mutable teams. Note that when mutable teams are enabled, the computational cost of duration prediction increases significantly, as our particle projection approach to predicting the duration of a mutable team (Section 6.5.1) is computationally expensive.

When mutable teams are disabled, the planner must either fill a role in its entirety with a single agent, or leave the role empty. Optional roles are still available, but are constrained in this fashion. The planner has a limited ability to trade resources (agents) for time (task duration), but in a much more discrete and less flexible manner than is possible with mutable teams.

Varying Live Task Modification

When live task modification is enabled, the planner is allowed to adjust the duration of executing join tasks, adjust the start times and durations for as-yet unexecuting join tasks of executing cooperative tasks, and add or remove unexecuting join tasks from executing cooperative tasks. The details are discussed in Chapter 7. Note that these operations all presuppose the existence of mutable teams: as formulated here, live task modification is meaningless if mutable teams are not available.

If live task modification is disabled, no join tasks associated with an executing cooperative task may be adjusted in any fashion by the planner. Note that it is possible for unresolvable conflicts to arise in this case: if an agent's task overruns and comes into conflict with the agent joining the end of an already-executing cooperative task, the planner has no options available to resolve the conflict. In these experiments, such conflicts are allowed to remain; the executive will delay the start of the join task internally until the agent becomes available.

Metrics

While the primary evaluation metric for this experiment was the makespan of the executed schedule, we also examined several other measures to gain insight into

how and why proactive replanning improves the makespan. Makespan was the only metric directly optimized by our approach.

In order to verify the mechanism by which live task modification reduces the makespan, we evaluate a measure we term missed opportunity. *Missed opportunity* measures the optional roles on the schedule's critical path that were not filled. To determine the missed opportunity in a schedule, we first find the set of tasks that lie on the schedule's critical path. At each point in time, we calculate the minimum of two values: the number of open optional roles in tasks on the critical path and the sum of the number of filled optional roles in non-critical tasks and the number of idle agents. This function is thus an approximate measure of when, and how many, agents could have been added to the critical path, under the assumption that agents may move from task to task instantaneously. The missed opportunity for a schedule is the integral of this function. Due to the instantaneous transfer assumption, this measure of missed opportunity is an upper bound only. A schedule properly optimized with respect to makespan should minimize missed opportunity, as in general assigning agents to non-critical tasks will not affect the makespan. Live task modification is expected to further reduce missed opportunities, as the planner may fill the optional roles of executing tasks as needed.

We also evaluate the potential costs of proactive replanning by examining the amount of plan repair required in each condition, the time needed to construct, optimize, and execute the schedules, and the amount of duration prediction that is performed. In Chapter 6, we found that the use of mutable teams simultaneously reduced the number of required repair iterations and increased the time required for each iteration, resulting in no significant change in the time needed to construct and optimize a schedule. In these experiments, we examine the effects of the three components of proactive replanning on the number of repair iterations and various decompositions of the time consumed by the planner. Live duration prediction and live task modification are expected to increase both metrics, due to the cost of predictions, conflicts introduced by fluctuations in the predictions, and the expansion of the set of tasks that may be repaired.

Experimental Conditions

Due to the dependence of live task modification on the availability of mutable teams, all eight combinations of the three components of proactive replanning need not be evaluated. Instead, a total of six experimental conditions were evaluated, enumerated in Table 8.3. 100 runs were performed for each condition, with each run building, optimizing, and executing a schedule, starting from a blank state. Given the scale of the experiment, the 600 runs were distributed across 23 computers of varying power. Analysis showed that inter-computer variation only had a

Table 8.3: The six experimental conditions evaluated in this experiment.

Live Duration Prediction	Mutable Teams	Live Task Modification	Notes
No	No	DNA	Baseline.
Yes	No	DNA	Live duration prediction alone has little direct effect.
No	Yes	No	Mutable teams alone account for half the total improvement.
No	Yes	Yes	Without live duration prediction, the effects of live task modification are limited.
Yes	Yes	No	Live duration prediction provides no significant advantages without live task modification.
Yes	Yes	Yes	Complete proactive replanning system.

significant effect on measures of computation time (e.g. time spent repairing). All timing results are reported from the subset of 452 runs performed on a cluster of 9 identical computers. The runs in this subset are distributed approximately evenly across the six conditions.

8.4.3 Analysis Procedure and Definitions

In the following section, a variety of metrics are evaluated and analyzed. To clarify the discussion, we define several terms to unambiguously reference different portions of an experimental run. A *run* consists of the formation of an initial schedule from a list of available agents and required tasks, its optimization, and its subsequent execution. The actions taken by the planner during a run may be decomposed in several ways. When decomposing according to *phase*, we segregate the data according to the temporal portion of the run in which it occurred. We define two phases, and two sub-phases, within a run:

1. *Build Initial*: The construction and optimization of the initial schedule. This phase lasts from the beginning of the run until just prior to the start of execution. It is composed of two sub-phases:

(a) *Construct Initial*: The construction of a complete and legal (but unoptimized) schedule. This is relatively short, and involves only repair operations.

(b) *Optimize Initial*: The optimization of the initial schedule, consisting of both optimization and repair operations.

2. *Execute*: The execution of the schedule, including repair due to the effects of execution, optimization, and repair due to the effects of optimization.

In addition, a run may be analyzed with respect to the function being performed: repair, optimization, or prediction. Throughout the analyses below, we examine different portions of the experimental runs, in order to evaluate specific hypotheses or to further explain noteworthy results. We use the term *planning time* to refer to the total time used by the planning system during a given phase of the schedule.

When performing each analysis, we utilize a 2x3 two-way ANOVA design. The presence or absence of live duration prediction (labeled *LDP*) serves as the two-level variable (LDP). The combination of mutable teams (abbreviated *MT*) and live task modification (abbreviated *LTM*) provides the three-level variable $(MT - LTM)$, with the values $(\sim MT)$, $(MT, \sim LTM)$, and (MT, LTM). We combine mutable teams and live task modification in this fashion in order to properly capture the dependence of live task modification on the presence of mutable teams.

The ANOVA analysis yields a *p*-value, as well as an *F-ratio*, for the effect of (LDP) and $(MT - LTM)$ on the response variable in question, as well as any interaction between (LDP) and $(MT - LTM)$. If the *p*-value is less than 0.05, the given variable had a statistically significant effect on the response variable. If the effect of $(MT - LTM)$ is significant, we perform a post-hoc Student's t-test, which examines the three levels of $(MT - LTM)$, and determines which levels are significantly different. While this provides useful information, it also may mask interactions with (LDP): if the addition of LDP affects two of the $(MT - LTM)$ levels in opposite directions, a significant result may appear to be insignificant. If the interaction between (LDP) and $(MT - LTM)$ is significant, we perform a Student's t-test on the 6 combinations of variable levels, again yielding groups of levels that are not significantly different.

In many of the analyses below, live duration prediction (LDP) and live task modification (LTM) must be simultaneously present for either to have a significant effect. We refer to this mutually dependent condition with the shorthand *LDP+LTM*.

For each response variable, a figure is provided depicting the mean and standard error of the data for each of the 6 experimental conditions. Note that the

standard error provides a measure of the accuracy to which the mean can be determined, and is not a measure of the spread of the underlying data. In addition, a table of *p* and *F* values is provided for the (LDP) and $(MT - LTM)$ variables, as well as their interaction. Statistically significant *p*-values are set in bold type.

8.4.4 Data and Analysis

Overall, the complete proactive replanning system is able to decrease the average makespan by 11.5% and reduce the missed opportunity metric by 60.4%, as compared with the baseline system, all while consuming little more than one second of computation time per timestep during execution. Mutable teams are responsible for 5% of the drop in makespan, as well as a 15.5% reduction in repair time and a 58.6% reduction in the amount of repair needed during optimization. Mutable teams result in a slight increase in overall planning time, as well as an increased number of duration predictions, due to their complexity. LDP+LTM provides the remaining 6.5% decrease in average makespan, while simultaneously increasing the number of predictions, due to the greater frequency of state updates. Finally, the use of live duration prediction significantly increases overall planning time, due to its indirect influence on our optimization strategy.

Makespan

The metric of primary interest is the makespan of the final schedule, and any effects on it by live duration prediction, mutable teams, and live task modification. Fig. 8.6 plots the mean makespans and standard errors for the six experimental conditions.

The full proactive replanning condition ((LDP, MT, LTM), right of Fig. 8.6) reduces the average makespan by 11.5% (88.7 minutes), as compared to the baseline condition ($(\sim LDP, \sim MT)$, left of Fig. 8.6). MT has a significant effect on the makespan, while LDP does not have an individual effect. However, LDP acts as an enabler for LTM: LTM results in a significant change in the makespan only if LDP is also enabled. We believe from the structure of the underlying system that LDP catalyzes LTM: the planner cannot effectively modify executing tasks without updated predictions of their expected durations. However, we note that the statistical analysis does not allow us to prove or disprove this theory: from a statistical perspective, it is equally likely that LTM is catalyzing LDP.

Analysis of the makespan data reveals that the $(MT-LTM)$ variable has a significant effect on the final makespan (Table 8.4). A post-hoc Student's t-test within the $(MT-LTM)$ variable determined that a significant difference existed between all three levels. From this, we are able to state that MT and LTM individually have statistically significant effects on the schedule's makespan. Note that the signifi-

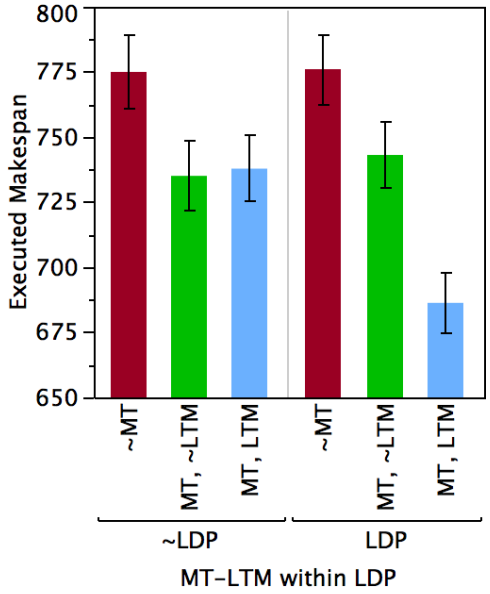


Figure 8.6: The means and standard errors of the final schedule's makespan for each of the six experimental conditions. See Table 8.4 for ANOVA analysis.

Table 8.4: ANOVA analysis results for the makespan of the executed schedule.

Metric	(LDP) $F(1, 599)$	$(MT - LTM)$ $F(2, 598)$	Interaction $F(2, 598)$
Executed Makespan *Fig. 8.6*	$p = 0.1855$ $F = 1.7571$	$p <$ **0.0001** $F = 11.8500$	$p =$ **0.0458** $F = 3.1004$

cance of LTM is due solely to the large effect of the (LDP, MT, LTM) condition, indicating that while LTM has a significant effect, it is only exhibited when in the presence of LDP. Fig. 8.6 indicates that MT and LDP+LTM contribute approximately equally to the improvement from the baseline to the (LDP, MT, LTM) condition (MT reduces makespan by 5%, while adding LDP and LTM yields the additional 6.5%).

While (LDP) had no significant effect individually, we can see from Fig. 8.6 that it and LTM must be presently simultaneously for either to affect the makespan. In the absence of LDP, insufficient warning of task overruns is provided for LTM to transfer in additional agents. If LTM is not present, the planner has a limited

Table 8.5: ANOVA analysis results for the missed opportunity measure.

Metric	(LDP) $F(1, 599)$	$(MT - LTM)$ $F(2, 598)$	Interaction $F(2, 598)$
Missed Opportunity *Fig. 8.7*	$p <$ **0.0001** $F = 49.9094$	$p <$ **0.0001** $F = 20.4852$	$p =$ **0.0002** $F = 858332.9000$

ability to respond to the information provided by LDP. This mutually enabling effect is captured in the analysis by the detection of a significant interaction effect between the (LDP) and $(MT - LTM)$ conditions.

A post-hoc Student's t-test on the interaction effect indicated that the following sets of conditions were mutually significantly different:

- (LDP, MT, LTM)
- $((LDP, \sim MT), (\sim LDP, \sim MT))$
- $((\sim LDP, MT, LTM), (\sim LDP, MT, \sim LTM))$

This confirms the trends plotted in Fig. 8.6: while LDP does not have an individually significant effect, its interaction with LTM is significant. This test shows that the slight difference between $(\sim LDP, MT, LTM)$ and $(\sim LDP, MT, \sim LTM)$ is not significant, confirming that LDP and LTM must both be present for either to be effective.

Note that LDP had a significant effect in Chapter 5 due to the characteristics of that domain. In Chapter 5, the objective was to maximize reward, and a variety of short, low-reward tasks were available. As a result, LDP allows the planner to schedule short tasks as gaps in the schedule are predicted. This results in LDP having a direct effect on the reward earned. In contrast, when minimizing makespan, LDP primarily provides advance warning of over- or under-runs. Without LTM, the planner is not able to directly address these problems and opportunities. LDP alone would have a more significant effect in a real-time scenario, as the advance warning would result in not only opportunities, but additional time to repair or optimize the schedule. The increased repair and optimization time is not a factor in these experiments, as we allow sufficient time between steps of execution to repair any conflicts and perform 5 optimizations per timestep.

Missed Opportunities

Missed opportunity is another metric of interest, and provides a conservative and approximate measure of how often the planner failed to fill optional roles on the

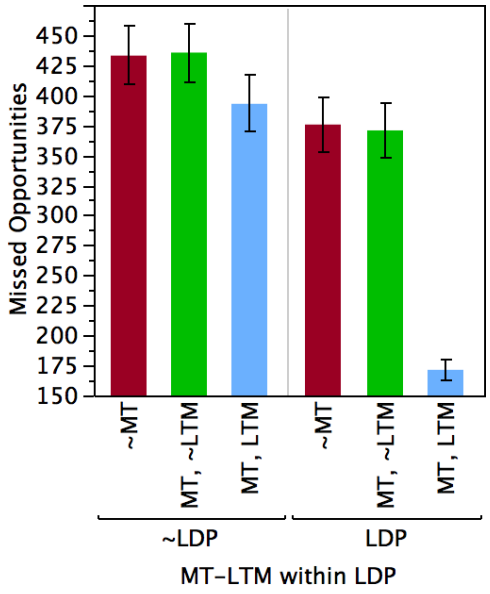


Figure 8.7: The means and standard errors of the missed opportunity metric for each of the six experimental conditions. See Table 8.5 for ANOVA analysis.

critical path. Fig. 8.7 plots the mean missed opportunity for each of the six experimental conditions. (LDP) had a significant effect on missed opportunity, as did the $(MT-LTM)$ variable (Table 8.5). LDP's effectiveness is due to the planner's increased ability to predict the actual critical path, and allocate agents accordingly. A post-hoc test on the $(MT-LTM)$ variable revealed a significant difference between the (MT, LTM) level and the $((\sim MT), (MT, \sim LTM))$ levels, indicating that LTM is the source of the $(MT-LTM)$ effect on missed opportunity, although much of the improvement is due to the (LDP, MT, LTM) condition. The addition of LDP+LTM allows the planner to opportunistically fill optional roles on the critical path as agents become available.

Unsurprisingly, there was a significant interaction between the (LDP) and $(MT-LTM)$ conditions. As can be seen in Fig. 8.7, missed opportunity is minimized when LDP, MT, and LTM are all available. This is due to the synergy between these components of proactive replanning: LDP provides early warning of execution-time events, MT provides the means to respond, and LTM provides the opportunity to do so directly. By granting the planner the ability to more accurately predict the critical path and to transfer agents onto active portions of the critical path, proactive replanning yields schedules that focus the efforts of the available agents on the critical portions of the scenario. The complete proactive

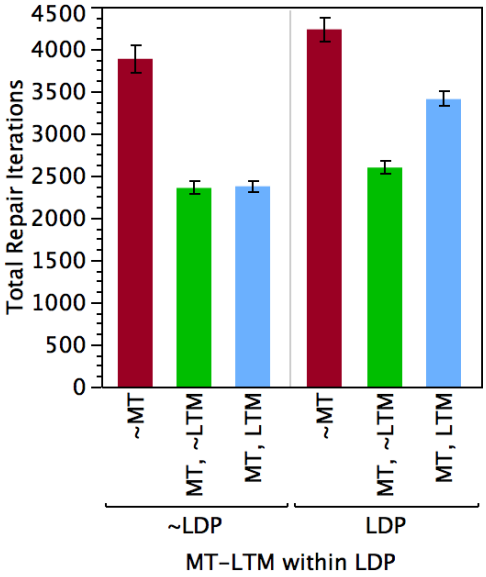


Figure 8.8: The means and standard errors of the number of repair iterations performed during the construction, optimization, and execution of each schedule for each of the six experimental conditions. See Table 8.6 for ANOVA analysis.

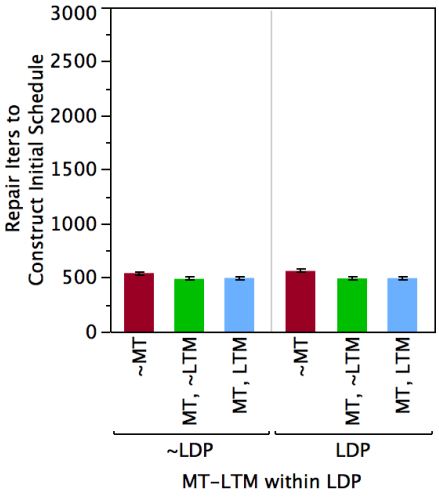
replanning system reduces missed opportunities by 60.4%, as compared with the baseline system.

While proactive replanning significantly reduces this metric, it does not approach zero. This is due to the metric's nature: the calculation assumes that agents may be instantly repositioned, when in reality a significant amount of setup time is often required. The missed opportunity remaining in the (LDP, MT, LTM) condition is largely due to open optional roles on the critical path that are distant from the available agents, making transfer inefficient.

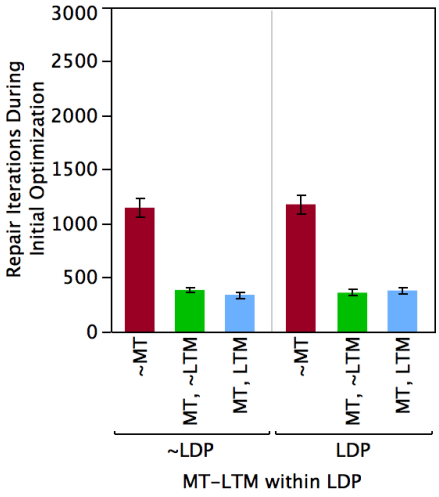
Iterations of Repair

While proactive replanning provides significant benefits, it adds complexity to the planning process, with potential efficiency impacts. Fig. 8.8 plots the number of repair iterations performed under each condition[6]. This data includes repair performed during all phases of the runs. Fig. 8.9 decomposes the amount of repair according to the phase of the run in which it occurred. The results from Chapter 6 are mirrored here, with mutable teams significantly reducing the amount of repair

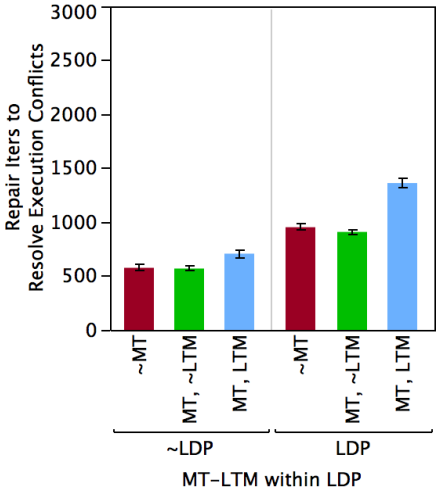
[6]The number of optimizations performed is solely a function of the schedule's final makespan: 5 iterations of optimization are performed after each step of execution.



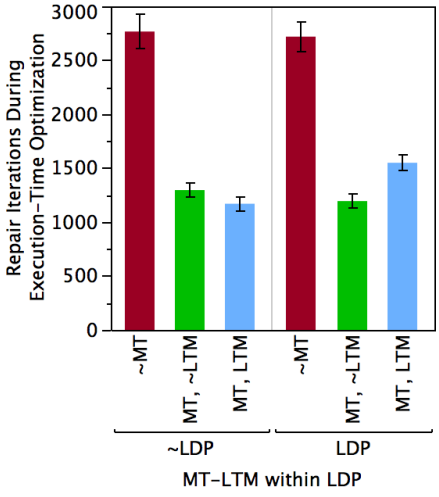
(a) Iterations of repair required to construct the initial, unoptimized schedule.



(b) Iterations of repair needed to resolve conflicts induced during the optimization of the initial schedule.



(c) Repair iterations expended in repairing conflicts created by the effects of execution.



(d) The number of repair iterations required to resolve conflicts created during optimization performed while executing.

Figure 8.9: The means and standard errors of the number of repair iterations required in the four contexts of repair. See Table 8.6 for ANOVA analysis.

Table 8.6: ANOVA analysis results for iterations of repair, as decomposed by context.

Metric	(LDP) $F(1, 599)$	$(MT - LTM)$ $F(2, 598)$	Interaction $F(2, 598)$
Total *Fig. 8.8*	$p <$ **0.0001** $F = 36.6261$	$p <$ **0.0001** $F = 112.3869$	$p =$ **0.0005** $F = 7.6777$
Build Initial Schedule *Fig. 8.9(a)*	$p = 0.5271$ $F = 0.4004$	$p <$ **0.0001** $F = 10.6360$	$p = 0.7505$ $F = 0.2871$
Optimize Initial Schedule *Fig. 8.9(b)*	$p = 0.6942$ $F = 0.1547$	$p <$ **0.0001** $F = 143.6052$	$p = 0.7863$ $F = 0.2406$
Execution *Fig. 8.9(c)*	$p <$ **0.0001** $F = 332.2610$	$p <$ **0.0001** $F = 56.9743$	$p <$ **0.0001** $F = 15.7644$
Optimize During Execution *Fig. 8.9(d)*	$p = 0.3588$ $F = 0.8435$	$p <$ **0.0001** $F = 132.1709$	$p =$ **0.0372** $F = 3.3091$

required, due to the additional flexibility and lower-impact options available to a mutable teams-enabled planner. Live duration prediction increases the amount of repair, as advance warning of over- and under-runs is available. In addition, the conflicts resulting from fluctuations in predicted task durations must be resolved. When live duration prediction is enabled, live task modification further increases the quantity of repairs, as the set of tasks that may be repaired is expanded.

While we have taken measures to reduce the impact of the fluctuations (or jitter) introduced by live duration prediction (Section 5.5), there may be more that can be done to reduce the negative impact of live duration prediction in this domain. However, a balance must be maintained between delaying updates to reduce repair time and ensuring that significant duration changes are presented promptly, allowing the planner to address them.

The (LDP) and $(MT - LTM)$ conditions both have statistically significant

effects on the overall amount of repair (Fig. 8.8), as does their interaction (Table 8.6). A post-hoc Student's t-test on the $(MT - LTM)$ condition revealed that all three levels are significantly different from one another. This indicates that mutable teams and live task modification are exerting independent effects upon the amount of repair. In fact, mutable teams reduce the number of repair iterations by an average of 1583 iterations, or 39% of the repair required without mutable teams. This parallels our results from Chapter 6. However, live task modification slightly offsets this, on average increasing the amount of repair by 415 iterations. This is a result of the expanded scope that live task modification affords the planner: it is able to optimize and repair a wider range of tasks, thus requiring more effort.

To gain further insight into the effects of proactive replanning on repair, we have decomposed the repair iterations according to the phase of the run in which they occurred (Fig. 8.9). During the construction and optimization of the initial schedule, only the $(MT - LTM)$ condition had a significant effect (Fig. 8.9(a) and 8.9(b), Table 8.6). In both cases, a post-hoc Student's t-test on the $(MT - LTM)$ condition revealed that the $(\sim MT)$ level differed significantly from $((MT, LTM)$ and $(MT, \sim LTM))$. This indicates that the effect is due solely to mutable teams, with the most significant effect occurring during the optimization phase. Mutable teams are more effective in a tightly packed schedule, as they allow more surgical repair of conflicts, minimizing the repair's effect on the remainder of the schedule. While mutable teams are effective during the initial repair stage (Fig. 8.9(a)), the improvement is less marked, as the schedule is temporally dispersed in this phase.

As expected, (LDP) only has a significant effect on the number of repairs due to the effects of execution (Fig. 8.9(c), Table 8.6). With frequent updates to the predicted duration of executing tasks, the planner will be aware of more conflicts during a run, resulting in more repair. This allows the planner to adjust to unexpected events early enough in order to ameliorate or take advantage of them.

A post-hoc Student's t-test of the data presented in Fig. 8.9(c) reveals that the (MT, LTM) level of the $(MT - LTM)$ condition differs significantly from $((\sim MT)$ and $(MT, \sim LTM))$. This indicates that MT is not of significant use when repairing execution-induced conflicts. This may appear counterintuitive. However, the majority of conflicts that arise due to the effects of execution are relatively short overlaps between tasks requiring the same resources. These are generally resolved through a simple right-shift of a portion of the schedule (Section 3.1.3), which makes little use of mutable teams.

The amount of repair required to resolve conflicts created during mid-execution optimization (Fig. 8.9(d)) reveals a pattern similar to that observed during the building of the initial schedule. The $(MT - LTM)$ condition has a statistically significant effect (Table 8.6), and a post-hoc Student's t-test reveals that the $(\sim MT)$ level differs significantly from the (MT, LTM) and $(MT, \sim LTM)$ levels. This again

indicates that mutable teams are the source of the significance: as we observed during the optimization of the initial schedule, the effectiveness of mutable teams is most marked in a tightly-packed schedule. A Student's t-test on the interaction between the (LDP) and $(MT - LTM)$ variables revealed that the members of the following groups of conditions do not significantly differ from one another:

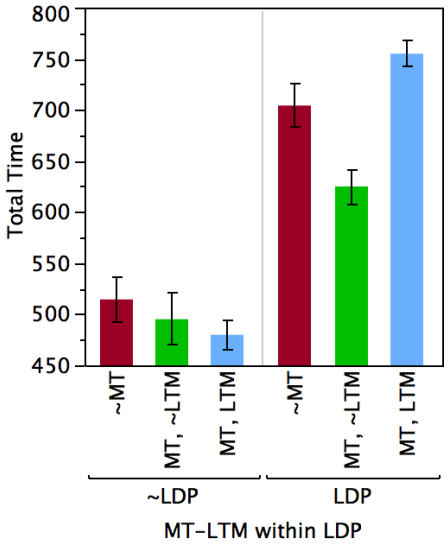
1. $(\sim LDP, \sim MT)$ and $(LDP, \sim MT)$
2. (LDP, MT, LTM) and $(\sim LDP, MT, \sim LTM)$
3. $(\sim LDP, MT, \sim LTM)$, $(LDP, MT, \sim LTM)$, and $(\sim LDP, MT, LTM)$

In summary, mutable teams reduce the number of repair iterations required when used during optimization by 58.6% on average, as the increased flexibility and decreased impact of repairs allows the repair of conflicts with minimal disturbance to the remainder of the schedule. This indicates that the repair advantages of mutable teams observed in Chapter 6 extend into execution. Live duration prediction mildly increases the required amount of repair due to executing tasks, as a result of the early warning of over- and under-runs that it provides to the planner. When used in combination with live task modification, the amount of repair required during execution increases further, as more tasks may be modified.

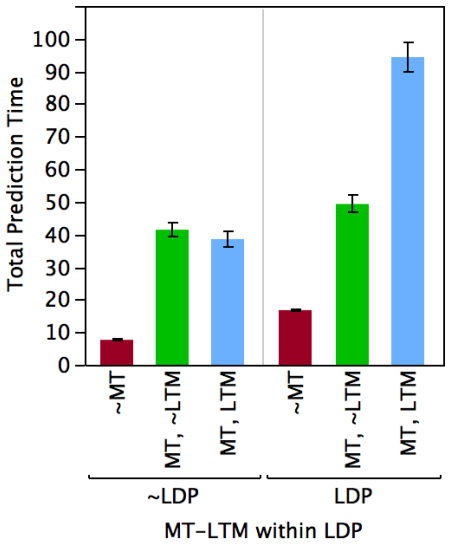
Time By Function

In Chapter 6, the decrease in repair iterations due to mutable teams was offset by an increase in the time required for each repair iteration, resulting in no net change in the time required. Fig. 8.10(a) plots the total time needed to construct, optimize, and execute the schedules in this experiment. The reduction in repair iterations due to the effect of mutable teams (Fig. 8.8) is largely offset by the increased cost of each iteration of repair. Measurements are in seconds of user time, as reported by `rusage()`. Fig. 8.10 also depicts the division of time between repair (Fig. 8.10(c)) and optimization (Fig. 8.10(d)), as well as the time spent forming duration predictions (Fig. 8.10(b)). The prediction time is measured across the entire run, and is included in the repair and optimization timing data. The time required to optimize, even without including the resulting repairs, dominates, indicating that our optimization methods may be a fruitful area for future efficiency improvements (Fig. 8.10(d)).

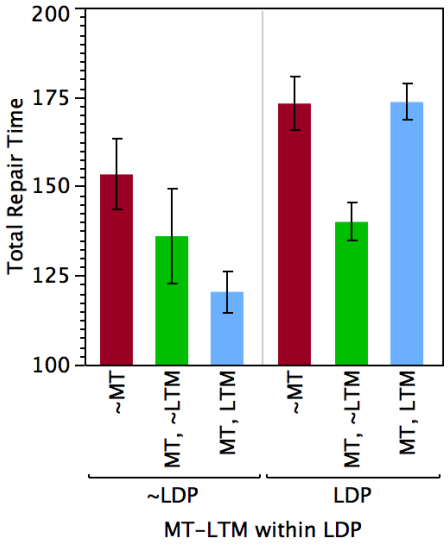
As can be seen from Fig. 8.10(b) and Table 8.7, the addition of mutable teams increases the overall cost of prediction: we must make use of particle projection prediction (Section 6.5.1), which is significantly more complex than prediction for immutable teams. In addition, more predictions are required when mutable



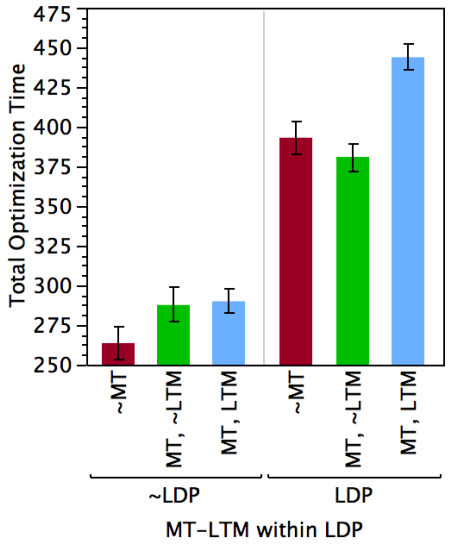
(a) Total time, from blank schedule to end of execution.



(b) Time spent predicting duration distributions during optimization and repair.



(c) Time spent repairing the schedule in response to the effects of execution and optimization.



(d) Time spent optimizing the schedule, not including repair induced by optimization.

Figure 8.10: The means and standard errors of the time spent on different operations. For timing analysis, we use the subset of 452 runs performed on a cluster of 9 identical computers. Measurements are in seconds of user time, as reported by `rusage()`. See Table 8.7 for ANOVA analysis.

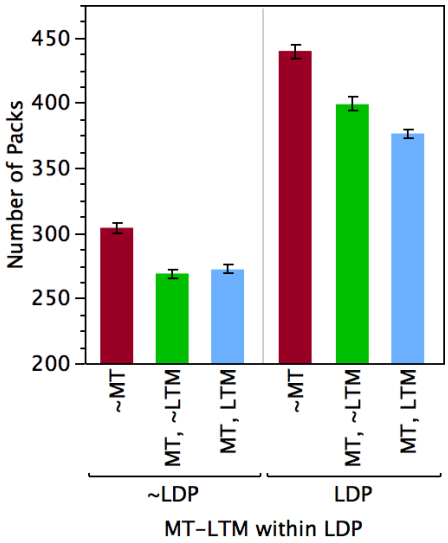


Figure 8.11: More expensive packing optimizations occur with LDP active, resulting in a more time-consuming optimization process. See Table 8.7 for ANOVA analysis.

Table 8.7: ANOVA analysis results for time metrics, as decomposed by function.

Metric	(LDP) $F(1, 451)$	$(MT - LTM)$ $F(2, 450)$	Interaction $F(2, 450)$
Total time *Fig. 8.10(a)*	$p <$ **0.0001** $F = 155.1040$	$p =$ **0.006** $F = 5.1720$	$p =$ **0.001** $F = 7.0291$
Prediction *Fig. 8.10(b)*	$p <$ **0.0001** $F = 146.4133$	$p <$ **0.0001** $F = 247.7283$	$p <$ **0.0001** $F = 62.1106$
Repair *Fig. 8.10(c)*	$p =$ **0.0002** $F = 13.8521$	$p =$ **0.0102** $F = 4.6378$	$p =$ **0.0133** $F = 4.3651$
Optimization *Fig. 8.10(d)*	$p <$ **0.0001** $F = 260.5608$	$p <$ **0.0001** $F = 9.4664$	$p =$ **0.0059** $F = 5.1887$
Pack Count *Fig. 8.11*	$p <$ **0.0001** $F =$1348.720	$p <$ **0.0001** $F = 75.1185$	$p =$ **0.0003** $F = 8.2337$

teams are available (Fig. 8.13), due largely to the need to evaluate potential agent transfers. The effect of the $(MT - LTM)$ condition on the time spent predicting is significant, while a post-hoc Student's t-test on the $(MT - LTM)$ variable indicates that all three levels are significantly different. This shows that both MT and LDP+LTM have significant effects. The increase due to LDP+LTM is a result of the increased number of predictions made during execution, as updated task state arrives and the planner evaluates potential transfers (Fig. 8.13(b)).

While predicting the duration of tasks involving mutable teams is more expensive, it does not outweigh the overall reduction in repair iterations (Fig. 8.8) when considering the overall time needed for repairs (Fig. 8.10(c)). Mutable teams appear to have a mild effect on the time needed to perform optimization (Fig. 8.10(d)), due to the increased complexity in reasoning about mutable teams.[7] However, the difference is not statistically significant. The $(MT - LTM)$ condition has a significant effect, but a post-hoc Student's t-test shows that the (MT, LTM) level is significantly different from the $(MT, \sim LTM)$ and $(\sim MT)$ levels, indicating that mutable teams alone do not increase optimization time significantly.

Live duration prediction alone significantly increases the planning time needed (Fig. 8.10(a)). In addition to the cost of the additional repair needed to resolve conflicts resulting from changes in predicted durations (Fig. 8.8), live duration prediction incurs the cost of at least one prediction per executing task per timestep[8], amounting to approximately 60,000 additional duration predictions per run[9]. These costs could be ameliorated by reducing the frequency of prediction updates. Doing so allows the system designer to decrease planning time at the cost of increased latency and an increased possibility of detecting a conflict or opportunity too late to adjust the schedule appropriately.

The increase in time due to live duration prediction is due in part to an increase in prediction time (Fig. 8.10(b)) and repair time (Fig. 8.10(c)). The increase in prediction time is due to an increased number of predictions and an increased cache miss rate: our prediction system caches duration predictions, to avoid unnecessary recalculations. When live duration prediction is active, more predictions are needed, from a broader span of states, resulting in not only more prediction requests, but more prediction calculations. The increase in repair time is due to the increased number of repair iterations (Fig. 8.8).

More surprising is the large increase in optimization time when live duration

[7] Note that this is only the time spent reasoning about optimization, and does not include the time spent repairing any resulting conflicts. As a result, there is no direct connection between Fig. 8.10(d) and the number of repair iterations in Figs. 8.8 and 8.9.

[8] More accurately, an additional prediction is required per executing task per state update. In these experiments, a state update arrives for every executing task at every timestep.

[9] Due to caching, far fewer predictions are required in our system.

prediction is active. This is at first counterintuitive: while updating the planner's duration predictions may result in conflicts, it does not seem that it should materially affect the complexity of optimization. It does not do so, but the use of live duration prediction does influence the choice of optimization heuristics. When LDP is active, more repair occurs, which has a tendency to slightly disperse the schedule, introducing slack into the critical path. This results in a packing of the schedule during the next round of optimization (Section 8.2.2). Packing is an expensive procedure, as it touches most, if not all, tasks on the schedule, and is iterative, in order to address constraint loops due to mutable teams (Section 3.1.3). As can be seen in Fig. 8.11 and Table 8.7, significantly more packing operations occur when live duration prediction is available.

While mutable teams decrease the time needed, live task modification offsets this gain when live duration prediction is also available, mirroring the number of repair iterations required (Fig. 8.10(a)).

Applying all three components of proactive replanning results in a 46.7% increase in planning time. While this is significant, we note that we were able to construct, optimize, and execute (including repair and optimization during execution) a nominally 10-hour plan in an average of 12.6 minutes. The additional planning time may or may not affect a given system, depending on the relationship between execution and planning speed. If time is a limiting factor, the frequency of duration prediction updates, the number of per-timestep optimizations, or the number of particles used in particle projection prediction may be reduced (see Section 8.5.2).

In the interest of completeness, we summarize below the results of the relevant post-hoc Student's t-tests on the four by-phase views of the timing data sets.

In the total time data (Fig. 8.10(a)), the (LDP) and $(MT - LTM)$ conditions have statistically significant effects, as well as a significant interaction (Table 8.7). A post-hoc Student's t-test on the $(MT - LTM)$ variable shows a significant difference between the $(MT, \sim LTM)$ level and the $((\sim MT), (MT, LTM))$ levels, indicating that the reduction due to mutable teams is offset by live task modification.

An analysis of the time spent computing duration predictions (Fig. 8.10(b)) reveals that again the (LDP) and $(MT - LTM)$ conditions have statistically significant individual effects and a significant interaction (Table 8.7). A post-hoc Student's t-test on the $(MT - LTM)$ variable indicates that all three levels are significantly different, while the same analysis applied to the interaction between (LDP) and $(MT - LTM)$ shows that the only pair of conditions that are not significantly different are $(\sim LDP, MT, \sim LTM)$ and $(\sim LDP, MT, LTM)$.

The portion of time spent repairing the schedule is plotted in Fig. 8.10(c), and includes the creation of the initial schedule, as well as the repair of conflicts caused

Table 8.8: ANOVA analysis results for time metrics, as decomposed by phase of operation.

Metric	(LDP) $F(1, 451)$	$(MT - LTM)$ $F(2, 450)$	Interaction $F(2, 450)$
Construct Initial *Fig. 8.12(a)*	$p = 0.9088$ $F = 0.0131$	$p <$ **0.0001** $F = 83.4520$	$p = 0.9820$ $F = 0.0181$
Optimize Initial *Fig. 8.12(b)*	$p = 0.9613$ $F = 0.0024$	$p <$ **0.0001** $F = 36.2360$	$p = 0.6326$ $F = 0.4584$
Execute *Fig. 8.12(c)*	$p <$ **0.0001** $F = 171.3615$	$p =$ **0.0047** $F = 5.4334$	$p =$ **0.0011** $F = 6.9417$
Time Per Timestep *Fig. 8.12(d)*	$p <$ **0.0001** $F = 367.2523$	$p <$ **0.0001** $F = 36.3050$	$p <$ **0.0001** $F = 31.0443$

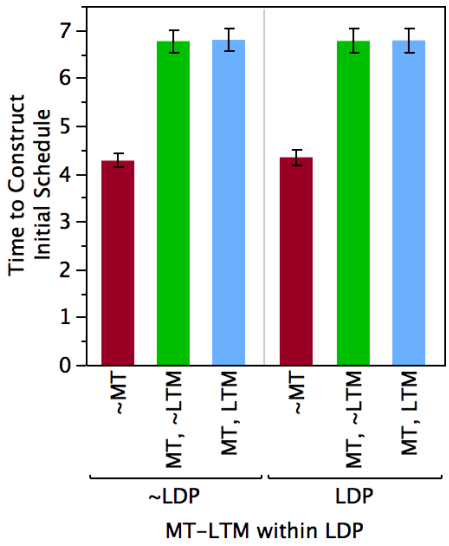
by execution-time events and optimization. A 2x3 ANOVA analysis reveals that the effect of both the (LDP) and $(MT - LTM)$ conditions, as well as their interaction, is statistically significant (Table 8.7).

As in the previous three timing data sets, (LDP), $(MT - LTM)$, and their interaction all have significant effects on the time spent optimizing the schedule (Fig. 8.10(d), Table 8.7). A post-hoc Student's t-test on the $(MT - LTM)$ condition shows that the (MT, LTM) level is significantly different from the $(MT, \sim LTM)$ and $(\sim MT)$ levels.

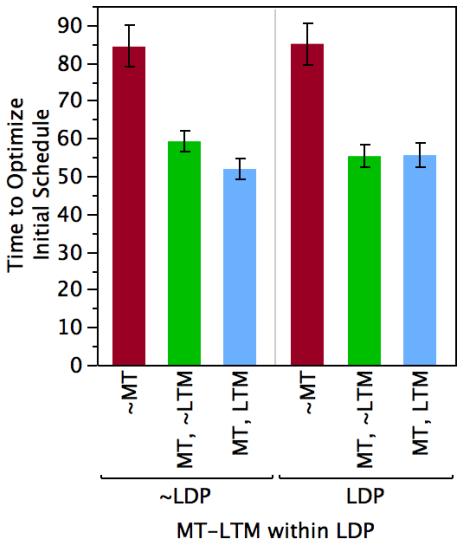
In examining the number of packing optimizations, a Student's t-test reveals that all three levels of the $(MT - LTM)$ variable differ to a statistically significant degree.

Time By Phase

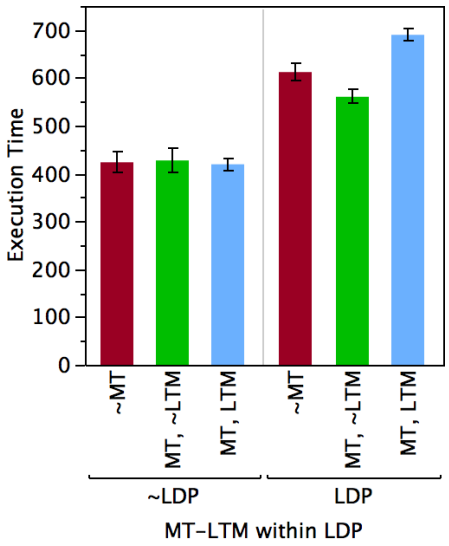
In addition to decomposing the planning time according to the operations being performed, we have also evaluated our system's efficiency during the three phases of a run (Fig. 8.12, Table 8.8). As expected, mutable teams are the only component of proactive replanning that has an effect during the construction and optimization of the initial schedule (Figs. 8.12(a) and 8.12(b), respectively). In both cases, the $(MT - LTM)$ condition has a significant effect, while (LDP) does not. In addition, a post-hoc Student's t-test shows that the $(\sim MT)$ level differs significantly from the (MT, LTM) and $(MT, \sim LTM)$ levels in both phases of the sched-



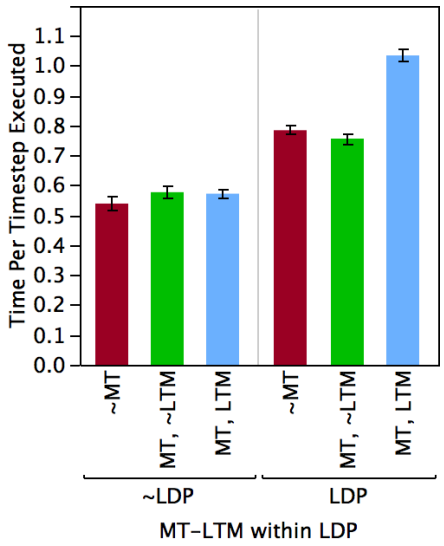
(a) Time spent constructing the initial, unoptimized schedule.



(b) Time spent optimizing the initial schedule, including repairs induced by optimization attempts.



(c) Time spent executing the schedule, including optimization and repair.



(d) Average time needed to execute a single step of the schedule, including optimization and repair.

Figure 8.12: The means and standard errors of the time spent on the three primary phases of a run, as well as the average time needed to execute a single step of the schedule. Measurements are in seconds of user time, as reported by `rusage()`. See Table 8.8 for ANOVA analysis.

ule build. This indicates that the difference between $(\sim LDP, MT, \sim LTM)$ and $(\sim LDP, MT, LTM)$ in Fig. 8.12(b) likely is not statistically significant. These results confirm those from the previous sections: mutable teams result in greater repair times (e.g. the repair-only initial schedule construction phase, Fig. 8.12(a)), but shorter optimization times, even when the resulting repair is included (Fig. 8.12(b)). The increased number of predictions necessary when operating with mutable teams (Fig. 8.13(a)) dominates under pure repair, but the flexibility afforded by them improves overall efficiency during optimization-induced repairs (Fig. 8.9(b), Fig. 8.12(b)).

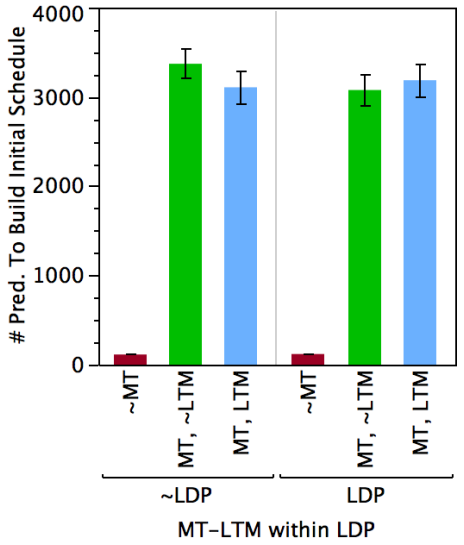
During execution, neither mutable teams nor live task modification have an appreciable effect on planning time if live duration prediction is not available (Fig. 8.12(c)). However, if live duration prediction is active, mutable teams significantly reduce the time needed, while the addition of live task modification significantly increases it. A post-hoc Student's t-test on the interaction of the (LDP) and $(MT - LTM)$ variables reveals that the three conditions without LDP are statistically indistinguishable, while the three that include LDP are mutually distinct. This parallels the results of our examination of efficiency with respect to overall repair and optimization (Figs. 8.10(c) and 8.10(d)), as does the statistically significant increase in time induced by the addition of LDP.

Fig. 8.12(d) plots the time required to execute one timestep of the schedule, including all repair and optimization time. This factors out the effect of the schedule's makespan on the overall planning time. It provides no additional insights into the interaction of the three components of proactive replanning[10], but does show that even the complete proactive replanning system takes little more than one second of computation per timestep. Depending on the size of the domain's timestep, this may yield true real-time performance.

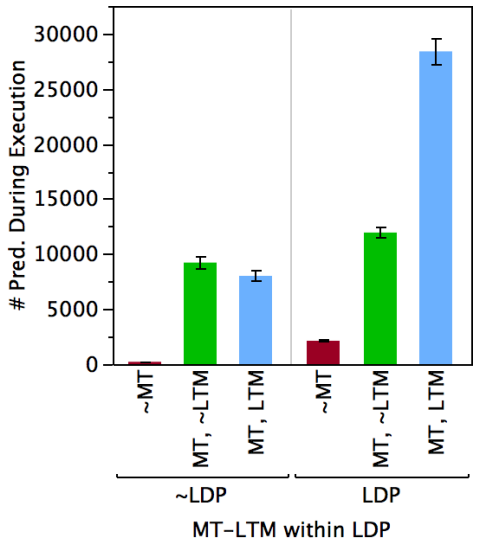
Duration Prediction

We have also analyzed the effects of the various components of proactive replanning on the number of duration predictions performed, as well as the time spent forming them. Fig. 8.13 charts the number of predictions made during the building of the initial schedule (Fig. 8.13(a)) and its execution (Fig. 8.13(b)). While it may appear that mutable teams cause a massive increase in the amount of prediction, a portion of the difference is due to the caching performed by our prediction framework. The data graphed in Fig. 8.13 represents cache misses, for which full duration predictions were computed. If a prediction for a task from a particular state had been recently computed, the cached result was immediately returned, and

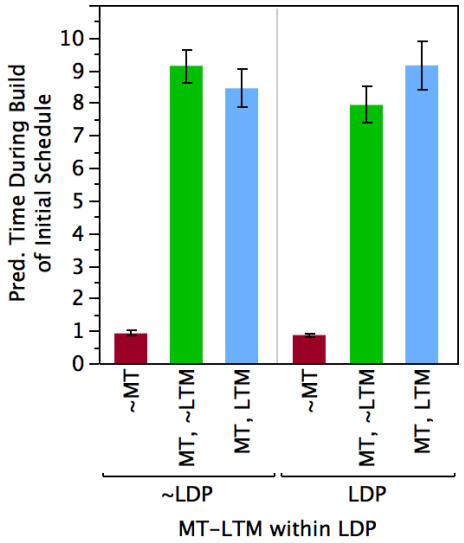
[10]The post-hoc Student's t-tests on $(MT - LTM)$ and the interaction of (LDP) and $(MT - LTM)$ yield identical results to those applied to the execution time data, with the exception that $(LDP, \sim MT)$ and $(LDP, MT, \sim LTM)$ do not differ significantly.



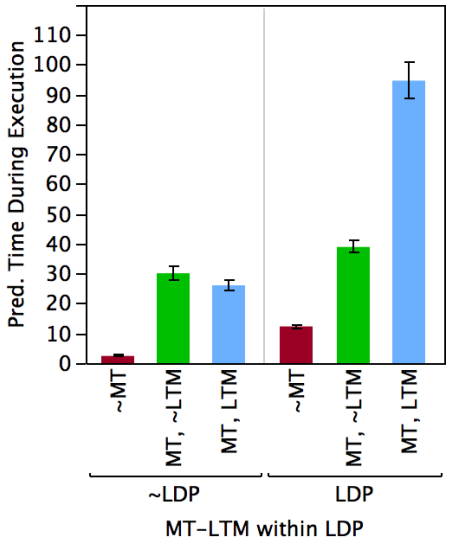
(a) Number of predictions performed during the construction and optimization of the initial schedule.



(b) Number of predictions performed during execution.



(c) Time spent predicting during construction and optimization of initial schedule.



(d) Time spent predicting during execution of the schedule.

Figure 8.13: The means and standard errors of the number of predictions performed during repair and optimization, and the time required, sectioned into those that occur prior to and during execution. See Table 8.9 for ANOVA analysis.

Table 8.9: ANOVA analysis results for the various measures of the number of predictions and time consumed.

Metric	(LDP) $F(1, 451)$	$(MT - LTM)$ $F(2, 450)$	Interaction $F(2, 450)$
Num. Predictions: Build Initial *Fig. 8.13(a)*	$p = 0.5555$ $F = 0.3479$	$p <$ **0.0001** $F = 298.4441$	$p = 0.3955$ $F = 0.9290$
Num. Predictions: Execute *Fig. 8.13(b)*	$p <$ **0.0001** $F = 297.3936$	$p <$ **0.0001** $F = 415.3714$	$p <$ **0.0001** $F = 155.2374$
Total Pred. Time During Build *Fig. 8.13(c)*	$p = 0.6463$ $F = 0.2109$	$p <$ **0.0001** $F = 168.9134$	$p = 0.1466$ $F = 1.9261$
Total Pred. Time During Execution *Fig. 8.13(d)*	$p <$ **0.0001** $F = 156.4994$	$p <$ **0.0001** $F = 173.2642$	$p <$ **0.0001** $F = 72.0913$

the query not included in this data. As the state space for mutable teams is much larger, but the cache size was held constant, a greater fraction of duration prediction requests resulted in cache misses, translating into a greater number of predictions, as well as time required.

In addition to the caching effects, fewer prediction requests overall are required when mutable teams are not available. Many fewer team profiles are possible, reducing the need for the evaluation of different combinations of agent commitments. A lack of mutable teams also eliminates the prediction-heavy process of reasoning about transferring agents, further contributing to the relative paucity of predictions in the $(\sim MT)$ conditions.

Mutable teams clearly increase the number of predictions made during execution, as do the addition of live duration prediction and live task modification (if LDP is also available) (Fig. 8.13(b), Table 8.9). While mutable teams expand the set of team profiles, swelling the prediction space, live duration prediction naturally results in more predictions throughout execution. As state updates arrive from the executive, new predictions are computed and integrated into the schedule. The ad-

dition of live task modification (when LDP is active) yields an additional jump in the quantity of predictions: much more reasoning about potential agent transfers to and from executing tasks is performed in a complete proactive replanning system.

A post-hoc Student's t-test performed on the data plotted in Fig. 8.13(a) clearly reveals that the $(\sim MT)$ level of $(MT{-}LTM)$ differs significantly from $((MT, LTM)$ and $(MT, \sim LTM))$. In Fig. 8.13(b), the differences between all three levels of $(MT - LTM)$ are statistically significant. A t-test applied to the interaction of the (LDP) and $(MT - LTM)$ variables demonstrates that the only levels not significantly different from the remainder are $(\sim LDP, MT, LTM)$ and $(\sim LDP, MT, \sim LTM)$.

The total time spent predicting during each phase (Figs. 8.13(c) and 8.13(d)) closely mirrors the number of predictions (Figs. 8.13(a) and 8.13(b)), as would be expected. Post-hoc Student's t-tests yield identical results to the corresponding number of predictions data.

Summary

This experiment has demonstrated that the utility of mutable teams first explored in Chapter 6 extends into the execution and ongoing repair and optimization of the schedule, with mutable teams resulting in significantly shorter schedules that are constructed and executed in less time. Live task modification provides additional makespan reductions, albeit only when both live duration prediction and mutable teams are available. This is to be expected: without prior warning and the ability to modify team profiles, there are few opportunities for live task modification to improve the schedule. We have shown that the different aspects of proactive replanning are both useful in isolation and combine to form an effective replanning system. The complete system is able to construct and execute schedules that are on average 11.5% shorter than the baseline system, with 60.4% fewer missed opportunities. The proactive replanner dynamically reallocates its resources to focus on the most critical tasks, all well consuming approximately one second of compute time per timestep executed.

Finally, we note that in the future live duration prediction and live task modification will likely prove invaluable in both real-world domains and those involving deadlines. The advance warning provided by live duration prediction will translate into additional plan repair and optimization time in real-world scenarios. This experiment was performed in synchronous simulation, eliminating the link between early warning and additional repair or optimization iterations. In scenarios involving hard deadlines, the utility of live duration prediction and live task modification will increase significantly. With these capabilities, the planner should have enough time to identify potential deadline violations in time to take preventative measures.

8.5 Domain Exploration

We have shown that proactive replanning is effective in the domain and scenario discussed above. We have also performed a series of experiments in which we varied elements of the domain, to explore how the domain's characteristics affect the usefulness of live duration prediction, mutable teams, and live task modification. In particular, we varied the number of agents available, the number of particles used when forming duration predictions, and the length of non-terminal failures. Our results show that the utility of proactive replanning is enhanced as more agents become available and as the length of failures increases, while even poor predictions enable a proactive replanner to perform well.

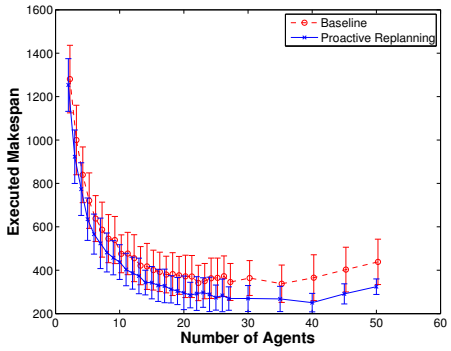
In each of the experiments below, we vary a parameter of the domain. At each value of the parameter, we perform an experiment identical in form to that reported in the previous section, with 100 runs per combination of proactive replanning components. For conciseness, we analyze only the baseline $(\sim LDP, \sim MT)$ and complete proactive replanning (LDP, MT, LTM) conditions. When varying the number of particles, we examine solely the (LDP, MT, LTM) condition.

Plots for each experiment are provided, depicting the mean and standard deviations of each response variable, as a function of the variable being controlled (e.g. number of agents). Note that this differs slightly from the plots in the previous section, in that we report standard deviation, as opposed to standard error. Standard deviation provides an estimate of the spread of the underlying data, and allows us to approximately compare the baseline and proactive replanning curves below. The data points for the baseline condition are slightly offset from their true position on the X axis to increase the visibility of the error bars. We also plot the percentage change in the response variable between the baseline and proactive replanning cases, where applicable.

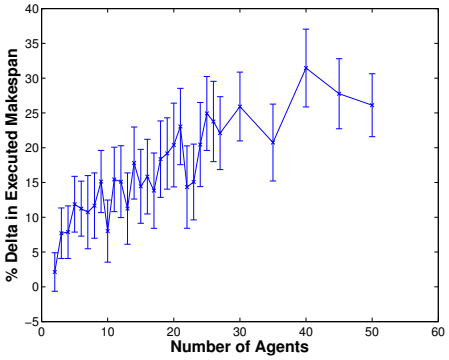
8.5.1 Effect of Agent Scarcity

One interesting aspect of multi-agent scheduling is the scarcity of resources (e.g. agents). If sufficient agents are available to execute all tasks in parallel, the scheduling problem is trivial. However, this is never the case in a realistic domain, with the quantity of tasks outnumbering the agents by a significant margin. In this experiment, we examined how the reduction in makespan provided by proactive replanning changes as more agents become available, varying the number of agents from 2 to 50. The results reported in Section 8.4 utilize 5 agents.

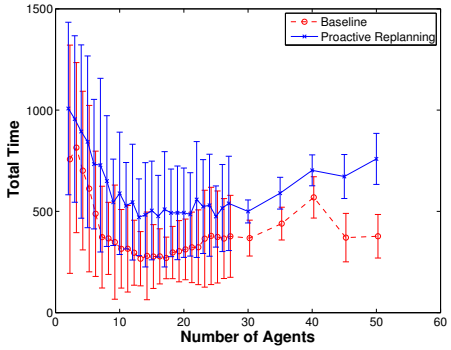
Fig. 8.14(a) plots the average makespan for the baseline and proactive replanning conditions. As expected, the makespan decreases as more agents become available, and asymptotes as agents saturate the domain. The increase in makespan



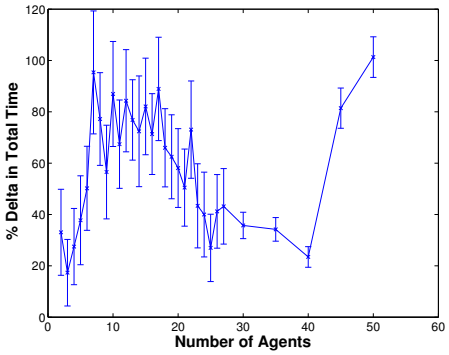
(a) The average makespan achieved for the baseline and complete proactive replanning conditions, in domains with a range of agents available. Error bars are the standard deviations of each sub-experiment.



(b) The percentage reduction in makespan between the baseline and complete proactive replanning conditions, as a function of the number of agents in the domain. Error bars are the standard deviations of the difference, scaled into a percentage.



(c) The average planning time required for the baseline and complete proactive replanning conditions, in domains with a range of agents available. Error bars are the standard deviations of each sub-experiment.



(d) The percentage increase in planning time required between the baseline and complete proactive replanning conditions, as a function of the number of agents in the domain. Error bars are the standard deviations of the difference, scaled into a percentage.

Figure 8.14: The effects of proactive replanning are magnified as more agents become available.

when more than 35 agents are available is likely due to the assumption on the part of some of our heuristics that agents are a scarce resource. Heuristics designed for use in an agent-poor environment may perform suboptimally in an agent-rich domain.

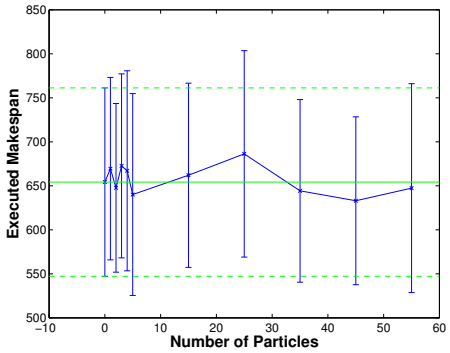
Fig. 8.14(b) plots the percentage that the application of proactive replanning reduces the makespan, as compared with the baseline. As additional agents become available, the utility of live task modification and mutable teams increases, as more agents are available to fill holes in the tasks' optional roles. The curve appears to begin asymptoting near the end, although significant noise is present. This asymptoting is to be expected: as sufficient agents become available to perform more and more tasks in parallel, some of the potential for gain via proactive replanning will be lost.

We also examined the effects of varying the number of available agents on the overall planning time required for all phases of an experimental run. Fig. 8.14(c) plots the total time required in the baseline and proactive replanning conditions, measured in the number of user seconds, according to `rusage`. Fig. 8.14(d) plots the percentage of additional time required by the proactive replanning system, as compared with the baseline. As the number of agents increased, planning time decreased, until approximately 15 agents were available. This decrease is due in part to the reduction in makespan, and in part to the ability of the planner to quickly find a free agent to resolve a conflict or perform an optimization. After this point, the complexity introduced by reasoning about ever-larger quantities of agents outweighs the ever-decreasing incremental reduction in makespan.

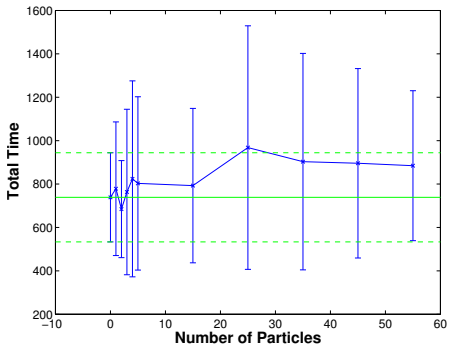
8.5.2 Effect of Number of Prediction Particles

When constructing a proactive replanning system, one variable is the number of particles used to form duration predictions (Section 6.5.1). In this experiment, we have evaluated the effect of varying the number of particles on both the makespan of the executed schedule and the total planning time required. Fig. 8.15(a) plots the average makespan achieved by a proactive replanning system as the size of the particle set is varied. The degree of variation is much less than the standard deviation, implying that the number of particles has little effect on the executed makespan. In fact, the proactive replanning system is able to achieve nearly identical results when using distribution transfer functions (horizontal lines in Fig. 8.15(a); Section 6.5.1), which are known to be highly inaccurate. This indicates that even very inaccurate duration predictions provide useful information, and that there is little utility in forming extremely precise predictions. Instead, the utility likely flows from the rapid updating of the predictions in response to the realities of execution.

Fig. 8.15(b) plots the planning time required to construct and execute the schedule, as a function of the number of particles. As expected, this trends upwards as particles are added. However, as additional particles provide little benefit, the system designer likely is best served by the use of a few particles, yielding a faster system with little, if any, reduction in the quality of the executed schedules.



(a) The average makespan achieved for the complete proactive replanning condition, as the number of particles used to form duration predictions varies.



(b) The average planning time required for the complete proactive replanning condition, as the number of particles used to construct duration predictions is varied.

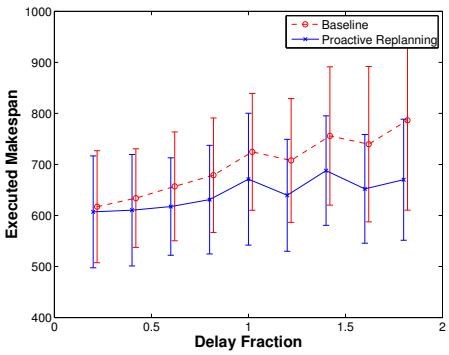
Figure 8.15: Increasing the number of particles used to construct duration predictions increases the planning time required, with no significant effect on the makespan. Error bars are the standard deviations of each sub-experiment. The solid horizontal line is the average when transfer functions are used, while the dashed horizontal lines represent the standard deviation of the transfer function sub-experiment.

8.5.3 Effect of Failure Length

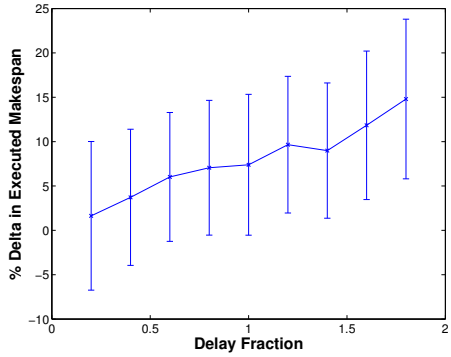
If live task modification is to be maximally useful, changes in the predicted duration of tasks must be long enough that there is time to reposition agents and add them to the affected task. Non-terminal failures are the primary cause of significant changes in predicted duration, as the team must take time to recover from the failure before the task may proceed. In a subset of this domain's tasks, optional agents significantly increase the rate at which the team may recover from failures. In these tasks, adding an agent during a failure recovery will reduce the task's duration significantly, even if the optional role is filled only for a brief time.

In this experiment, we examine the relationship between proactive replanning and the delay incurred by failures. The tasks were modified such that the delay resulting from a failure was a fraction of the task's duration when no failures occurred. We then varied this fraction from 0.2 to 1.8, and examined the average makespan and number of missed opportunities of the schedules produced and executed in the baseline and complete proactive replanning conditions.

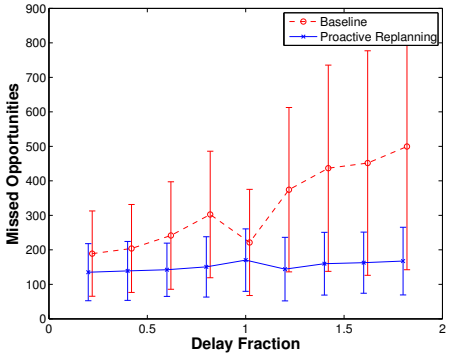
Fig. 8.16(a) plots the average makespan of the two conditions as a function of the delay fraction. As expected, the average length of a schedule increases as



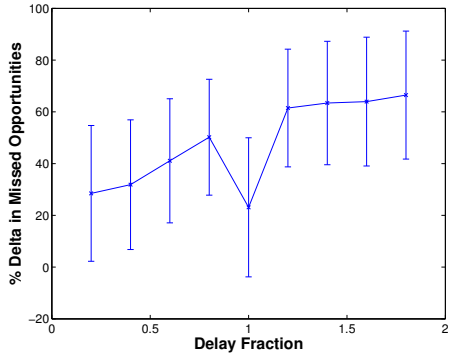
(a) The average makespan achieved for the baseline and proactive replanning conditions, as the effect of non-terminal failures is varied. Error bars are the standard deviations of each sub-experiment.



(b) The percentage reduction in makespan between the baseline and proactive replanning conditions, as a function of the effect of a non-terminal failure. Error bars are the standard deviations of the difference, scaled into a percentage.



(c) The average number of missed opportunities for the baseline and proactive replanning conditions, as the effect of a non-terminal failure changes. Error bars are the standard deviations of each sub-experiment.



(d) The percentage decrease in missed opportunities realized by the proactive replanning system, relative to the baseline, as a function of the effect of failures. Error bars are the standard deviations of the difference, scaled into a percentage.

Figure 8.16: The effects of proactive replanning are magnified as the impact of non-terminal failures increases.

failures have greater impact, but the rate of increase is greater in the baseline condition. This is illustrated by Fig. 8.16(b), where we plot the percentage reduction in makespan realized by moving to a proactive replanning system. As the length of each delay increases, the opportunity for live task modification to ameliorate its ef-

fects increases, allowing proactive replanning to generate schedules progressively more efficient than the baseline.

Missed opportunities is a metric that evaluates how well the planner was able to focus its resources upon the critical set of tasks. As Figs. 8.16(c) and 8.16(d) show, a proactive replanner is able to eliminate the growth of missed opportunities as the impact of non-terminal failure is increased. By transferring agents into and out of active teams, a proactive replanning system may react to failures as they occur, rather than being forced to either accept their affects or lavish resources upon all tasks. We do not have an explanation for the sudden dip in missed opportunities for the baseline case at a delay fraction of 1.0.

This experiment demonstrates that proactive replanning provides benefits even when the impact of an individual failure is low, and that those benefits scale linearly with the degree of impact.

8.6 Conclusions

Proactive replanning is conceptually simple, yet powerful: predict problems or opportunities, then adjust currently executing tasks and the remainder of the schedule to compensate. We have evaluated the effectiveness of three components of a proactive replanning and execution system: live duration prediction, mutable teams, and live task modification. As we have shown in previous chapters, live duration prediction and mutable teams are useful concepts in their own rights. Here, we have reported the results of experiments evaluating the interaction between the three components and their overall effectiveness, examining metrics of overall schedule makespan, missed opportunities, computational cost, and the number of repair and optimization iterations necessary. The utility of mutable teams was reaffirmed, yielding schedules 5% shorter than otherwise possible and reducing total repair time by 15.5%. When live duration prediction and live task modification were also available, our proactive replanning system executed schedules 11.5% (88.7 minutes) shorter than the baseline, demonstrating the efficacy of the proactive replanning approach. In addition, we explored several aspects of the domain and their effects on the components of proactive replanning. We found that as the number of agents or the impact of non-terminal failures increase, proactive replanning becomes progressively more advantageous. We have shown proactive replanning to be of worth in two domains while varying a number of their characteristics, with utility whether its components are used in isolation or in combination. We have also found that live duration prediction is beneficial, even when only very approximate predictions are available. When used together, live duration prediction, mutable teams, and live task modification result in a flexible, robust, and efficient

replanning system able to construct and execute schedules in a significantly more efficient fashion.

Chapter 9

Conclusions

The thesis of this work is that a proactive replanner will be able to construct and execute more efficient or value-laden schedules by leveraging live duration prediction, mutable teams, and live task modification. To support this claim, we have developed algorithms for each of the three components, integrated them into the CASPER execution framework, and evaluated various aspects of proactive replanning in stochastic simulation. The components of proactive replanning, used either in isolation or in concert, allow the planner to produce and execute schedules significantly more efficient or reward-laden than otherwise possible.

We developed a general approach to the prediction of a distribution across remaining durations for tasks, applicable both prior to and throughout execution. Our approach is nonparametric, allowing it to model accurately the multi-modal duration distributions characteristic of many construction tasks. We evaluated the effectiveness of duration prediction in a multi-agent construction scenario in the absence of optional roles, mutable teams, and live task modification. By employing live duration prediction, the planner was able to achieve 45% of the improvement possible if it had complete foreknowledge of the outcome of all tasks. Further, we discovered that statistically significant improvements were possible with as few as four runs of training data.

We formulated the concept of optional roles and mutable teams, which allow the description of coordinated tasks that may make use of a variable number of agents, and which allow agents to join or leave at will. We explored and evaluated a variety of approaches to representing these concepts within the ASPEN planner. Our final representation meshes with ASPEN's model of tasks to allow the planner to reason not only about the task as a whole, but also about the commitment of individual agents to the task. We extended the ASPEN core in several ways to make the use of mutable teams more efficient, and constructed a suite of heuristics

that are aware of mutable teams to guide plan repair and optimization appropriately. We evaluated the effect of mutable teams on the pre-execution generation and optimization of schedules. The addition of mutable teams enabled the planner to construct schedules a statistically significant 5.65% shorter than possible with optional roles alone.

We developed the concept of live task modification, and extended ASPEN and the CASPER executive to support the necessary close coordination. Live task modification allows the planner to address directly the cause of any conflicts or optimization opportunities arising from execution-time events by adjusting the team profile of executing tasks. This increases the planner's ability to ameliorate the problems, or take advantage of the opportunities, predicted by live duration prediction.

We performed a broad-ranging experiment to evaluate the effects of the three components of proactive replanning, both in isolation and in all possible combinations, in a makespan-oriented multi-agent construction scenario. The complete proactive replanning system was able to execute schedules that are a statistically significant 11.5% shorter than possible with none of the components of proactive replanning. All three aspects of proactive replanning contributed to this improvement, with mutable teams providing a 5% reduction and live task modification yielding the remaining 6.5%. Live task modification resulted in shorter schedules only if live duration prediction was also available. While the components of proactive replanning are individually useful, they show their true potential when used in concert.

9.1 Future Work

Proactive replanning has proved to a be a fruitful area of research. While we have explored duration prediction, mutable teams, and live task modification as applied to relatively large multi-agent schedules in simulation, a number of avenues of research remain open.

9.1.1 Evaluation on Real-World Hardware in Real Time

While we have evaluated proactive replanning in simulation, and ensured that the planner's model of the world diverged from that of the simulator, we have not addressed all of the challenges involved in deploying a proactive replanning system on a real-world robotic team. There are three difficulties in doing so: creating agents capable of performing in mutable teams, performing duration prediction in a potentially much noisier environment, and ensuring that the planner operates on

an up to date model of the world.

Creating a group of robots capable of performing in mutable teams is a far from trivial effort, requiring capable hardware, flexible low-level controllers, a robust executive, and an applicable domain. While the Trestle team has made strides in this direction, performing construction tasks with a team of three heterogeneous agents (Heger et al., 2005) (Sellner et al., 2006) (Simmons et al., 2007), insufficient agents are available to form more than one team.

Duration prediction likely will become more difficult in a real-world environment, as the available state measurements become more indirect and noisy. There is no guarantee that the most appropriate state values will be measurable, potentially forcing us to form predictions based upon a state vector only loosely related to the task's duration. Identifying and measuring, quickly and accurately, the state variables of greatest utility will be a significant challenge as duration prediction is applied to live robots. In addition, the quality of the predictions may fall if the available data is noisier than the tasks we have simulated. While we have added significant stochasticity to our simulated tasks, real-world execution invariably includes unforeseen events that may corrupt the available training set. However, we have shown in Section 5.7.1 that live duration prediction is useful even when training data is scarce; we would expect similar results for noisy or scarce data in a live environment.

Finally, ensuring that the planner operates with an up-to-date model of the world may prove difficult. The fundamental problem is to determine the length of time to be spent repairing or optimizing the plan: when execution is asynchronous, the state of the world will evolve as planning proceeds, potentially invalidating the plan being formed before it can be put into practice. We note that this is unlikely to be an issue if execution proceeds at a relatively slow pace: we were able to construct, optimize, and execute notional 10-hour schedules for five agents in under 15 minutes, using a modern, but not cutting-edge, 2.66 GHz processor.

However, for faster-paced domains, or when insufficient computational power is available, there are several potential strategies that may help to ameliorate this problem. The first, already available in the CASPER architecture, is to make use of near- and long-term planning windows, as well as commitment windows. These windows are specified portions of the schedule within which the planner's options are restricted. The commitment window extends from the current time a short distance along the schedule, and prevents the planner from operating on the tasks contained within it. The length of the window is chosen based on how long a typical plan repair or optimization cycle takes, ensuring that the portion of the schedule about to be executed is not brought into conflict in the course of optimization. The near-term window is used to constrain plan repair, and runs from the end of the commitment window a domain-dependent distance into the future. Repairing

and optimizing the schedule within this window is of paramount importance to the planner, as this is the portion of the schedule about to enter the commitment window. The near-term should be repaired and optimized even if conflicts remain in the long-term window, which encompasses the remainder of the schedule. The long-term portion of the schedule should be repaired or optimized only as time allows.

In addition to the use of windows, constraints may be placed on the time spent optimizing the plan. By caching a valid plan prior to beginning optimization, the planner always is able to revert to a workable schedule if optimization goes awry.

Finally, we note that the utility of live duration prediction and live task modification may increase in a real-world scenario, as live duration prediction provides advance warning of problems or opportunities. In our experiments in simulation, we perform plan repair and optimization synchronously with execution, ensuring that sufficient time is available, and eliminating a portion of the utility of live duration prediction and task modification. In an asynchronous domain, the forewarning translates into additional time to repair or optimize the schedule, allowing more iterations to be performed than would be possible with a non-proactive system.

9.1.2 Reducing Repredictions

We evaluated several approaches to reducing the computational cost of duration prediction, in exchange for reduced accuracy (Sections 5.7 and 8.4). A complementary approach would be to reduce the number of repredictions performed. When live duration prediction is available, we currently repredict the duration distribution of every executing task whenever updated state information becomes available (e.g. after every step of execution). This is computationally expensive, but allows the planner to recognize an anomalous execution event in the minimum amount of time.

The most direct approach would be to only construct new predictions every N steps of execution, reducing the overall expense of prediction in exchange for an increase in the average lag between the occurrence of an anomaly and the planner's recognition of it. However, it may be possible to decide when to repredict in a more principled manner. A consequence of our particle projection prediction method (Section 6.5.1) is that we are able to build an expected distribution across task state at arbitrary points in the task's future. Whenever we construct a duration prediction, it should be possible to cheaply construct a series of distributions across the task's state space ($S_1 ... S_M$) ranging from the current time to M steps into the future. The primary computational cost of duration prediction is querying for nearby points; projecting them through the training database is quite fast. The planner then would be able to evaluate the task's evolving state against S_t to deter-

mine if the cost of forming a new prediction is warranted: if the probability of the task's state at time t is less than a threshold in S_t, an anomaly has likely occurred.

A simplification of this approach would be to predict the average state M timesteps into the future and linearly interpolate between it and the current task state. As the task executes, its evolving state would be compared with the interpolation, rather than a series of distributions, and a reprediction performed if the current state deviated significantly or after M steps of execution.

9.1.3 Heterogeneous Agents

In the experiments presented in this thesis, the available agents were homogeneous: all agents were able to fill any role, with the same efficiency and reliability. As agents differentiate, duration prediction and mutable teams become simultaneously more difficult and potentially more rewarding. In the simplest form of heterogeneous agents, the agents differ only by *which* roles they may fill: if two agents are able to fill a role, they are equally capable of performing it. This is straightforward to represent in our current formulation of mutable teams: we simply omit the relevant join tasks from the database of potential tasks (recall that a join task is specific to the combination of cooperative task, role, and agent performing the role).

If agents instead have varying levels of capability, the problem becomes much more difficult, and more interesting. In such a domain, agents may vary with respect to the way in which filling a particular role affects the team's efficiency or reliability. This complication is unlikely to affect mutable teams directly, but would have a dramatic impact on duration prediction. One approach would be to include the agents' level of capability in the task's state space, but the resulting expansion in the size of the state space may prove unmanageable. Under certain simplifying assumptions, it may be possible to factor out the effects of capability, performing a two-stage prediction in which an initial duration distribution is constructed, assuming some baseline capability level, then is transformed according to the actual set of capabilities. It may be possible to adapt duration transfer functions (Section 6.5.1) to address this.

9.1.4 Mutable Teams with Durative Integration and Disengagement

Our current model of mutable teams assumes that when an agent and recipient team rendezvous, the agent is able to join the team instantly, with no deleterious effects on the team's progress. In reality, this will be true only for "non-contact" roles, such as scouts, computation platforms, or signal relays. When an agent joins a role in which it is cooperatively manipulating an object, or must otherwise closely coordinate with other agents, the recipient team likely will need to slow or stop

progress temporarily to allow the new agent to integrate into the team. This will serve to offset the usefulness of mutable teams to an extent, although the precise degree of offset of course will be highly domain-dependent.

If the planner does not take into account the cost of agents integrating or disengaging from teams, it may overuse mutable teams, to the detriment of the executed schedule. In Section 6.5.2, we discuss several potential approaches to modeling these effects within our duration prediction algorithm. Experimental evaluation of these possibilities, and the exploration of other potential solutions, remains an open research topic.

9.1.5 Mutable Teams and Semi-Terminal Failures

We have not yet explored optional roles that allow the team to recover from an otherwise unrecoverable failure, but mutable teams show significant promise in this realm. In their simplest incarnation, such "semi-terminal" failures cause the team's progress to halt until an otherwise optional role has been filled. In the absence of mutable teams and live task modification, the planner would be forced to treat such a role as required, in order to avoid a potentially infinite delay. In contrast, a proactive replanner is able to reason about the likelihood of such a failure and the average time needed to transfer an agent into the role. This allows the planner to make a principled decision about whether the role should be filled, and increases the flexibility of the system as a whole.

9.1.6 Applicability to Least-Commitment Planners

While we have developed our proactive replanning system using a most-commitment planner, there is no fundamental reason why proactive replanning concepts could not be applied to a constraint-based, least-commitment planner, such as IxTeT (Laborie and Ghallab, 1995). If the task start times and durations are modeled within the planner as distributions, duration prediction could prove particularly useful. If they are instead modeled as ranges, it would be straightforward to convert distributions into a range covering a specified percentage of the probability mass.

Modeling mutable teams may prove difficult, as it would be difficult to predict a likely duration distribution if the arrival and departure times of the participating agents are not known. It is theoretically possible to build duration distributions given a distribution across when an agent will arrive and depart, but it becomes quite computationally expensive, at least when using our current approach. Further research into how mutable teams could be efficiently represented in a least-commitment planner could prove fruitful.

Of the components of proactive replanning that we have investigated, live task modification is the most likely to be directly applicable. It primarily affects the interaction between the planner and the executive, an interface that is quite similar in both most- and least-commitment planners.

9.1.7 Predicting Resource Usage

It appears likely that our approach to duration prediction may be adapted to predict either total resource usage over the remainder of the task, or a profile of likely resource usage as a function of time. This possibility and a promising approach is discussed in Section 5.6, although investigating it further is beyond the scope of this thesis. If this proves to be feasible, it will provide the planner with a method to apply proactive replanning techniques to metrics such as minimizing resource consumption.

9.1.8 Human Interaction and Sliding Autonomy

An exciting future area of work in proactive replanning is its use to form integrated human-robot teams. With adequate sensing technology, it may be possible to build a duration prediction model for tasks performed by humans (either directly or via teleoperation), allowing the planner to predict the expected duration of a human's task throughout execution. With this information, the planner may be able to determine if the human is having difficulty with the task, and proactively offer to provide a robot either to assist with or complete the task. Being able to determine when another human is in need of assistance is an effective cooperative technique utilized by human teams. Bringing this technique to a mixed human-robot team could increase the efficiency and capability of the team as a whole.

Similarly, if humans are collocated with the robots or are available to perform teleoperation, the planner may consider a human as another possible agent to fill a role in a cooperative task. Of course, this is predicated upon the ability to model heterogeneous agents. Given knowledge about the human's skills, the planner would be able to request assistance when the human's abilities become particularly valuable. This would allow the planner to coordinate the sliding of autonomy for specific aspects of tasks between the autonomous agents and the collocated or remote human team members.

9.2 Summary

We have presented the concept of proactive replanning, and investigated three components: (live) duration prediction; optional roles and mutable teams; and live task

modification. We have integrated these methods into the ASPEN/CASPER system, extending it as necessary, and developed a suite of heuristics to guide ASPEN's iterative plan repair and optimization in their use. Live duration prediction and mutable teams were experimentally evaluated in isolation, proving to be useful techniques even when not used in concert with the remaining aspects of proactive replanning. We also examined the effects and interactions of the three components in various combinations on the ability of the system to build and carry out efficient schedules in the face of uncertain execution. Mutable teams and live task modification both produced significant reductions in schedule length, with live task modification's gains conditional upon the availability of live duration prediction. While much research remains, we have shown proactive replanning to be a useful addition to the arsenal of available planning and execution techniques.

Bibliography

Alami, R., Chatila, R., Fleury, S., Ghallab, M., and Ingrand, F. (1998). An architecture for autonomy. *International Journal of Robotics Research, Special Issue on Integrated Architectures for Robot Control and Programming*, 17(4).

Ambros-Ingerson, J. and Steel, S. (1988). Integrating planning, execution and monitoring. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*.

Bacchus, F. and Ady, M. (2001). Planning with resources and concurrency: A forward chaining approach. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*.

Beardah, C. C. (1997). Some archaeological applications of kernel density estimates. *Journal of Archaeological Science*, 24:347–354.

Beetz, M. and McDermott, D. V. (1994). Improving robot plans during their execution. In *Proceedings of the International Conference on AI Planning Systems (AIPS)*.

Belker, T., Hammel, M., and Hertzberg, J. (2003). Learning to optimize mobile robot navigation based on htn plans. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA '03)*, volume 3, pages 4136–4141.

Bernard, D., Dorais, G. A., Gamble, E., Kanefsky, B., Kurien, J., Millar, W., Muscettola, N., Nayak, P., Rouquette, N., Rajan, K., Smith, B., Taylor, W., and Tung, Y.-W. (2000). Final report on the remote agent experiment. In *Proceedings of NMP DS-1 Technology Validation Symposium*, Pasadena, CA.

Bertoli, P., Cimatti, A., Roverie, M., and Traverso, P. (2001). Planning in nondeterministic domains under partial observability via symbolic model checking. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*.

Boddy, M., Horling, B., Phelps, J., Goldman, R., Vincent, R., Long, A., and Kohout, B. (2005). C-taems language specification v. 1.06.

Bonasso, R., Firby, R., Gat, E., Kortenkamp, D., Miller, D., and Slack, M. (1997). Experiences with an architecture for intelligent, reactive agents. *Journal of Experimental and Theoretical Artificial Intelligence*, 9(2-3):237–256.

Bonet, B. and Geffner, H. (2000). Planning with incomplete information as heuristic search in belief space. In *Proceedings of the International Conference on AI Planning Systems (AIPS)*.

Bookstein, F. L. (1989). Principal warps: Thin plate splines and the decomposition of deformations. *IEEE Transations on Pattern Analysis and Machine Intelligence*, 11:567–585.

Breiman, L. (1993). *Classification and Regression Trees*. Chapman and Hall, Boca Raton.

Chatterjee, S. and Hadi, A. S. (1986). Influential observations, high leverage points, and outliers in linear regression. *Statistical Science*, pages 379–416.

Chien, S., Knight, R., Stechert, A., Sherwood, R., and Rabideau, G. (1999). Integrated planning and execution for autonomous spacecraft. In *Proceedings of the 1999 IEEE Aerospace Conference*, Aspen, CO.

Chien, S., Knight, R., Stechert, A., Sherwood, R., and Rabideau, G. (2000a). Using iterative repair to improve the responsiveness of planning and scheduling. In *Proceedings of the International Conference on AI Planning Systems (AIPS)*.

Chien, S., Rabideau, G., Knight, R., Sherwood, R., Engelhardt, B., Mutz, D., Estlin, T., Smith, B., Fisher, F., Barrett, T., Stebbins, G., and Tran, D. (2000b). Aspen – automated planning and scheduling for space mission operations. In *Space Ops*, Toulouse.

Cimatti, A. and Roveri, M. (2000). Conformant planning via symbolic model checking. *Journal of Artificial Intelligence Research*, 13:305–338.

Correll, N. and Martinoli, A. (2004). Collective inspection of regular structures using a swarm of miniature robots. In *Proceedings of the Ninth International Symposium on Experimenatl Robotis (ISER-04)*, number 6 (2005) in Springer Tracts in Advanced Robotics, Singapore.

Currie, K. and Tate, A. (1991). O-plan: the open planning architecture. *Artificial Intelligence*, 52.

Dias, M. B., Lemai, S., and Muscettola, N. (2003). A real-time rover executive based on model-based reactive planning. In *Proceedings of the International Conference on Robotics and Automation (ICRA)*.

Emery, R., Sikorski, K., and Balch, T. (2002). Protocols for collaboration, coordination and dynamic role assignment in a robot team. In *Proceedings of the IEEE International Conference on Robotics and Automation, 2002 (ICRA'02)*, volume 3, pages 3008–3015, Washington, DC. DOI: 10.1109/ROBOT.2002.1013689.

Estany, M. G. and Losilla, C. B. (1998). An application of the transformed kernel density estimation to labor earnings in spain. Working Papers in Economics 33, Universitat de Barcelona. Espai de Recerca en Economia.

Estlin, T., Rabideau, G., Mutz, D., and Chien, S. (2000). Using continuous planning techniques to coordinate multiple rovers. *Linkoping Electronic Articles in Computer and Information Science*, 5(16).

Estlin, T., Volpe, R., Nesnas, I., Mutz, D., Fisher, F., Engelhardt, B., and Chien, S. (2001). Decision-making in a robotic architecture for autonomy. In *Proceedings of the International Symposium on Artificial Intelligence, Robotics, and Automation in Space*.

Ferraris, P. and Giunchiglia, E. (2000). Planning as satisfiability in nondeterministic domains. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*.

Fikes, R. E., Hart, P. E., and Nilsson, N. J. (1972). Learning and executing generalized robot plans. *Artificial Intelligence*, 3.

Fikes, R. E. and Nilsson, N. J. (1971). Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2.

Firby, R. J. (1994). Task networks for controlling continuous processes. *Artificial Intelligence Planning Systems*.

Fox, M. and Long, D. (2002). Pddl 2.1: An extension to pddl for expressing temporal planning domains. Technical report, University of Durham, UK.

Frank, J. and Jónsson, A. (2003). Constraint-based attribute and interval planning. *Journal of Constraints, Special Issue on Constraints and Planning*, 8(4).

Friedman, J. H. (1991). Multivariate adaptive regression splines (with discussion). *Annals of Statistics*, 19:1–141.

Fukunaga, A. S., Rabideau, G., Chien, S., and Yan, D. (1997). ASPEN: A framework for automated planning and scheduling of spacecraft control and operations. In *Proceedings of the International Symposium on AI, Robotics and Automation in Space (i-SAIRAS)*, Tokyo, Japan.

Garvey, A. and Lesser, V. (1995). Design-to-time scheduling with uncertainty. Technical Report 95-03, University of Massachusetts.

Gat, E. (1992). Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*.

Gat, E. (1997). Esl: A language for supporting robust plan execution in embedded autonomous agents. In *Proceedings of the 1997 IEEE Aerospace Conference*.

Gerkey, B. P. and Mataric, M. J. (2003). A formal framework for study of task allocation in multi-robot systems. Technical Report CRES-03-13, University of Southern California.

Gordon, N. J., Salmond, D. J., and Smith, A. F. M. (1993). Novel approach to nonlinear/non-gaussian bayesian state estimation. *Proc. Inst. Elect. Eng. F*, 140:107–113.

Haigh, K. Z. and Veloso, M. M. (1998). Planning, execution and learning in a robotic agent. In *Proceedings of the International Conference on AI Planning Systems (AIPS)*.

Heger, F. W., Hiatt, L. M., Sellner, B., Simmons, R., and Singh, S. (September 5-8, 2005). Results in Sliding Autonomy for Multi-Robot Spatial Assembly. In *8th International Symposium on Artificial Intelligence, Robotics and Automation in Space (iSAIRAS)*, Munich, Germany.

Jennings, J. and Kirkwood-Watts, C. (1998). Distributed mobile robotics by the method of dynamic teams. In *???*

Karalic, A. (1992). Linear regression in regression tree leaves. In *Proceedings of ISSEK '92 (International School for Synthesis of Expert Knowledge)*.

Kay, S. M. (1993). *Fundamentals of Statistical Signal Processing: Estimation Theory*, chapter 7. Prentice Hall.

Kullback, S. and Leibler, R. A. (1951). On information and sufficiency. *Annals of Mathematical Statistics*, 22:79–86.

Laborie, P. and Ghallab, M. (1995). Planning with sharable resource constraints. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*.

Maheswaran, R. T. and Szekely, P. (2008). Criticality metrics for distributed plan and schedule management. In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduleing (ICAPS)*.

Mataric, M. J. (1992). Designing emergent behaviors: from local interactions to collective intelligence. In Meyer, J. A., Roitblat, H., and Wilson, S., editors, *Proceedings of the 2nd International Conference on Simulation of Adaptive Behavior (SAB-92)*, pages 432–441, Cambridge, MA. MIT Press.

Mataric, M. J., Sukhatme, G. S., and Ostergaard, E. H. (2003). Multi-robot task allocation in uncertain environments. *Autonomous Robots*, 14:255–263.

Meuleau, N., Dearden, R., and Washington, R. (2004). Scaling up decision theoretic planning to planetary rover problems. In *AAAI-04: Proceedings of the Workshop on Learning and Planning in Markov Processes Advances and Challenges*, volume Technical Report WS-04-08, pages 66–71, Menlo Park, CA. AAAI Press.

Murata, T. (1989). Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580.

Muscettola, N. (1994). *Intelligent Scheduling*, chapter HSTS: Integrating planning and scheduling. Morgan Kaufmann.

Muscettola, N., Dorais, G., Fry, C., Levinson, R., and Plaunt, C. (2002). Idea: Planning at the core of autonomous reactive agents. In *Proceedings of the 3rd International NASA Workshop Planning and Scheduling for Space*.

Muscettola, N., Nayak, P., Pell, B., and Williams, B. (1998). Remote agent: To boldly go where no ai system has gone before. *Artificial Intelligence*, 103(1-2).

Musliner, D. J., Durfee, E. H., hui Wu, J., Dolgov, D. A., Goldman, R. P., and Boddy, M. S. (2006). Coordinated plan management using multiagent mdps. In *Proceedings of the 2006 AAAI Spring Symposium on Distributed Plan and Schedule Management*. AAAI Press.

Nau, D., oz Avila, H. M., Cao, Y., Lotem, A., and Mitchell, S. (2001). Total-order planning with partially ordered subtasks. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*.

Nesnas, I., Wright, A., Bajracharya, M., Simmons, R., Estlin, T., and Kim, W. S. (2003). Claraty: An architecture for reusable robotic software. In *Proceedings of the SPIE Aerosense Conference*, Orlando, Florida.

Park, J. and Sandberg, I. W. (1991). Universal approximation using radial-basis-function networks. *Neural Computation*, 3(2):246–257. ISSN:0899-7667, published by MIT Press, Cambridge, MA, USA.

Parzen, E. (1962). On estimation of a probability density function and mode. *The Annals of Mathematical Statistics*, 33(3):1065–1076.

Quinlan, J. (1992). Learning with continuous classes. In *Proceedings of the 5th Australian Joint Conference on Artificial Intelligence*.

Reinsch, C. (1967). Smoothing by spline functions. *Numerical Mathematics*, 10:177–183.

Sellner, B., Heger, F. W., Hiatt, L. M., Simmons, R., and Singh, S. (2006). Coordinated multi-agent teams and sliding autonomy for large-scale assembly. *Special Issue of the Proceedings of the IEEE on Multi-Robot Systems*, 94(7).

Sellner, B., Simmons, R., and Singh, S. (2005). User Modelling for Principled Sliding Autonomy in Human-Robot Teams. In Parker, L. E., Schneider, F. E., and Schultz, A. C., editors, *Multi-Robot Systems: From Swarms to Intelligent Automata*, volume 3. Springer.

Silverman, B. W. (1986). *Density estimation for statistics and data analysis*. Chapman and Hall, London, UK.

Simmons, R. and Apfelbaum, D. (1998). A task description language for robot control. In *Proceedings of the Conference on Intelligent Robots and Systems (IROS)*, Victoria, Canada.

Simmons, R., Singh, S., Heger, F., Hiatt, L., Koterba, S., Melchior, N., and Sellner, B. (2007). Human-robot teams for large-scale assembly. In *Proceedings of the NASA Science Technology Conference 2007 (NSTC-07)*, Adelphi, MD.

Smith, D. and Weld, D. (1998). Conformant graphplan. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*.

Smith, D. and Weld, D. (1999). Temporal planning with mutual exclusion reasoning. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*.

Smith, S., Gallagher, A. T., Zimmerman, T. L., Barbulescu, L., , and Rubinstein, Z. (2006). Multi-agent management of joint schedules. In *Proceedings of the 2006 AAAI Spring Symposium on Distributed Plan and Schedule Management*.

Smith, S., Gallagher, A. T., Zimmerman, T. L., Barbulescu, L., , and Rubinstein, Z. (2007). Distributed management of flexible times schedules. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. AAAI Press.

Smith, S. F. (1988). Reactive scheduling systems. In *Expert Systems and Intelligent Manufacturing*, pages 43–56. Elsevier Science Publishing Co., Inc.

Smith, S. F. (1993). Integrating planning and scheduling: Towards effective coordination in complex, resource-constrained domains. In *Proceedings of the Italian Planning Workshop*, Rome, Italy.

Smith, S. F., Ow, P. S., Muscettola, N., Potvin, J.-Y., and Matthys, D. C. (1990). Opis: An opportunistic factory scheduling system. In *Proceedings of the Third International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE-90)*, Knoxville, TE.

Stone, P. and Veloso, M. (1998). Task decomposition and dynamic role assignment for real time strategic teamwork. In *Proceedings of 5th International Workshop on Intelligent Agents, Agent Theories, Architectures, and Languages (ATAL '98)*, Paris, France.

Strens, M. and Windelinckx, N. (2005). Combining planning with reinforcement learning for multi-robot task allocation. *Adaptive Agents and MAS II, LNAI 3394*, pages 269–274.

Tambe, M., Adibi, J., Al-Onaizan, Y., Erdem, A., Kaminka, G. A., Marsella, S. C., and Muslea, I. (1999). Building agent teams using an explicit teamwork model and learning. *Artificial Intelligence*, 110:215–239.

Vijayakumar, S. (2001). *Locally Weighted Projection Regression (LWPR) - a users manual*. Dept. of Computer Science and Neuroscience, University of Southern California, Los Angeles, CA 90089-2520.

Vijayakumar, S., D'Souza, A., and Schaal, S. (2005). Lwpr: A scalable method for incremental online learning in high dimensions. *Neural Computation*.

Vijayakumar, S. and Schaal, S. (2000). Locally weighted projection regression. In *Proceedings of the Seventeenth International Conference on Machine Learning (ICML 2000)*, volume 1, pages 288–293.

Volpe, R., Nesnas, I., Estlin, T., Mutz, D., Petras, R., and Das, H. (2000). Claraty: Coupled layer architecture for robotic autonomy. Technical report, Jet Propulsion Laboratory.

Wagner, T. and Lesser, V. (1999). Design-to-criteria scheduling: Real-time agent control. Technical Report 1999-58, University of Massachusetts.

Wasserman, L. (2003). *All of Statistics: A Concise Course in Statistical Inference*. Springer. ISBN 0387402721, 9780387402727.

Weld, D., Anderson, C., and Smith, D. (1998). Extending graphplan to handle uncertainty and sensing actions. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*.

Wilkins, D. E. (1988). *Practical Planning: Extending the Classical AI Planning Paradigm*. Morgan Kaufmann.

Woods, W. A. (1970). Transition network grammars for natual language analysis. *CACM 13*, 10:591–606.

Woods, W. A. (1973). *Natural Language Processing, ed. R. Rustin*, chapter An experimental parsing system for transition network grammars, pages 111–154. Algorithmics Press, New York.

Zweben, M., Davis, E., Daun, B., and Deale, M. (1992). Rescheduling with iterative repair. Technical Report N93-15288, AI Research Branch, NASA Ames Research Center, Moffet Field, CA 94025, USA.

Appendices

Appendix A

Algorithms for Duration Prediction with Mutable Teams

Algorithm A.1 The transfer function between two distributions is a mapping of a CDF value to a PDF and duration ratio, which may be applied to distributions similar to the source distribution in order to transform them into the same domain as the destination distribution.

function computeTransferFn(dist A, dist B) **do**
  $cdfVals = \emptyset$
  for $i = 0...1$ **do**
    {Place knot points where either CDF's second derivative is large.}
    if $\frac{d^2 cdf(A)}{dt^2} > eps \| \frac{d^2 cdf(B)}{dt^2} > eps$ **then**
      $cdfVals.add(i)$
    end if
  end for
  $map = \emptyset$
  for all $cv \in cdfVals$ **do**
    $\{dur(A, cv)$ returns the duration at which A's CDF has a value of $cv.\}$
    $\{pdf(A, d)$ returns the value of A's PDF at duration $d.\}$
    $dA = dur(A, cv)$
    $dB = dur(B, cv)$
    $dratio = \frac{dB}{dA}$
    $pratio = \frac{pdf(B,dB)}{pdf(A,dA)}$
    $map.add(\{cv, pratio, dratio\})$
  end for
  return map
end function
function applyTransferFn(dist A', map) **do**
  $outPDF = \emptyset$
  for all $m \in map$ **do**
    $\{cv, pratio, dratio\} = m$
    $dA' = dur(A', cv)$
    $d = dA' * dratio$
    $p = pdf(A', dA') * pratio$
    $outPDF.add(\{d, p\})$
  end for
  Interpolate the knot points of $outPDF$ to the resolution required and return the resulting distribution.
end function

Algorithm A.2 Distribution transfer functions utilize the assumption that the form of a particular team's duration distribution will not change during task execution to approximate the expected duration distribution, given future arrivals and departures of agents.

1: $\{t_i$ is the time at which the team profile change occurs, while p_i is the change in assigned agents. $t_0 = 0$ and p_0 is the null change.$\}$
2: $T = (t_0, p_0)...(t_n, p_n)$
3: $S =$ Initial state
4: $stateDists = \emptyset$
5: **for** $j = 0...n$ **do**
6: $\quad S = S + p_j$ {Apply the change in the team}
7: $\quad stateDists[j] = query(S)$
8: **end for**
9: $dist = stateDists[0]$
10: **for** $i = 1...n$ **do**
11: $\quad transferFn = computeTransferFn(stateDists[i-1], stateDists[i])$
12: $\quad dist = shift(dist, t_i)$ $\{shift(d, t)$ shifts the distribution d t units towards 0.$\}$
13: $\quad dist = applyTransferFn(dist, transferFn)$
14: **end for**
15: $dist = shift(dist, -t_n)$

Algorithm A.3 Particle projection prediction projects a set of particles through the training database to estimate the duration distribution of a task whose set of assigned agents is expected to change over time.

{t_i is the time at which the profile change occurs, while p_i is the change in assigned agents. $t_0 = 0$ and p_0 is the null change.}
$T = (t_0, p_0)...(t_n, p_n)$
$S =$ Query (initial) state
{$states$ is a vector of tuples, each of which consist of a point in the task's state space, a weight, and a duration offset}
$states = (S, 1, 0)$
for all T_j in T **do**
  $points = \emptyset$ {$points$ is of the same form as $states$}
  for all s_k in $states$ **do**
    $qpoints = query(S_k)$
    Multiply weights of $qpoints$ by weight of s_k {Duration offsets are copied from s_k}
    $points = points + qpoints$
  end for
  $points = resample(O, points)$
  Normalize the weights of $points$
  Aggregate any states in $point$ with identical states and duration offsets.
  if j == n **then**
    {If this was the last projection, build distribution and return}
    $dist$ = empty distribution
    for all pt_i in $points$ **do**
      $d = duration(pt_i) + offset(pt_i) + \sum_{i=1}^{n} t_i$
      Add kernel to $dist$ centered at d with weight $weight(pt_i)$
    end for
    return $dist$
  end if
  {Otherwise, apply the next projection}
  $states = \emptyset$
  for all pt_i in $points$ **do**
    $state(pt)$ = point closest to t_{j+1} time units further along pt_i's run
    $weight(pt) = weight(pt_i)$
    $offset(pt) = offset(pt_i) + (duration(pt_i) - duration(pt) - t_{j+1})$
    $state(pt) = state(pt) + p_{j+1}$ {Apply the change in the team}
    $states = states + pt$
  end for
end for

Appendix B

TaskSim Model Definitions

B.1 Outpost Scenario

The following listings are simplified forms of the TaskSim (Section 8.1.3) models used in the live duration prediction experiments (Chapter 5). In all models, t_g is the model's global time variable, which is used to synchronize the execution of the model with the rest of the simulator.

Listing B.1: A slightly simplified form of the *HabHaul* TaskSim model.

```
1  Machine HabHaul(Var DistanceTravelled = 0, Var GlitchRecovery = 0) {
2    Var targetDist = 50;
3    Var step = normal(0.5, 0.1);
4    Var glitch_r = uniform(0,1);
5    Var glitch_p = 0.01;
6    VarRef glitchCost = ( glitch_r <= glitch_p )*50;
7
8    State Hauling {
9      Start
10     Arcs => {
11       (Name => "Haul",
12        Test => ( targetDist - DistanceTravelled > 0 && glitchCost <= 0),
13        Effect => (DistanceTravelled = min( targetDist , DistanceTravelled + step );
14                   t_g ++;),
15        Target => "Hauling"),
16       (Name => "Glitch",
17        Test => (glitchCost > 0),
18        Effect => (GlitchRecovery = glitchCost ;),
19        Target => "Recovering"),
20       (Name => "Finished",
21        Test => ( targetDist - DistanceTravelled <= 0 && glitchCost <= 0),
```

```
22          Effect => (),
23          Target => "Done")
24      }
25    }
26    State Recovering {
27      Arcs => {
28        (Name => "Recover",
29          Test => (GlitchRecovery > 0),
30          Effect => (GlitchRecovery = max(0, GlitchRecovery - 1);
31                     t_g ++;),
32          Target => "Recovering"),
33        (Name => "Recovered",
34          Test => (GlitchRecovery <= 0),
35          Effect => (GlitchRecovery = 0;),
36          Target => "Hauling")
37      }
38    }
39    State Done {
40      Stop
41    }
42  };
```

Listing B.2: A slightly simplified form of the *Move* TaskSim model used in the Outpost scenario.

```
1  Machine Move(Var DistanceTravelled = 0, Var GlitchRecovery = 0) {
2    Var targetDist = 50;
3    Var step = normal(1, 0.25);
4    Var glitch_r = uniform (0,1);
5    Var glitch_p = 0.01;
6
7    VarRef glitchCost = ( glitch_r <= glitch_p )*50;
8
9    State Moving {
10     Start
11     Arcs => {
12       (Name => "Move",
13         Test => ( targetDist - DistanceTravelled > 0 && glitchCost <= 0),
14         Effect => (DistanceTravelled = min( targetDist , DistanceTravelled + step );
15                    t_g ++;),
16         Target => "Moving"),
17       (Name => "Glitch",
18         Test => (glitchCost > 0),
19         Effect => (GlitchRecovery = glitchCost ;),
20         Target => "Recovering"),
```

```
21        (Name => "Finished",
22         Test => ( targetDist - DistanceTravelled <= 0 && glitchCost <= 0),
23         Effect => (),
24         Target => "Done")
25      }
26    }
27    State Recovering {
28      Arcs => {
29        (Name => "Recover",
30         Test => (GlitchRecovery > 0),
31         Effect => (GlitchRecovery = max(0, GlitchRecovery - 1);
32                    t_g ++;),
33         Target => "Recovering"),
34        (Name => "Recovered",
35         Test => (GlitchRecovery <= 0),
36         Effect => (GlitchRecovery = 0;),
37         Target => "Moving")
38      }
39    }
40    State Done {
41      Stop
42    }
43  };
```

Listing B.3: A slightly simplified form of the *Cable* TaskSim model used in the Outpost scenario.

```
1  Machine Cable(Var DistanceTravelled = 0, Var GlitchRecovery = 0) {
2    Var targetDist = 50;
3    Var step = normal(0.3, 0.25);
4    Var glitch_r = uniform(0,1);
5    Var glitch_p = 0.02;
6
7    VarRef glitchCost = ( glitch_r <= glitch_p )*50;
8
9    State Cabling {
10     Start
11     Arcs => {
12       (Name => "Cable",
13        Test => ( targetDist - DistanceTravelled > 0 && glitchCost <= 0),
14        Effect => (DistanceTravelled = min( targetDist , DistanceTravelled + step );
15                   t_g ++;),
16        Target => "Cabling"),
17       (Name => "Glitch",
18        Test => (glitchCost > 0),
```

```
19          Effect => (GlitchRecovery = glitchCost ;),
20          Target => "Recovering"),
21        (Name => "Finished",
22          Test => ( targetDist - DistanceTravelled <= 0 && glitchCost <= 0),
23          Effect => (),
24          Target => "Done")
25      }
26    }
27    State Recovering {
28      Arcs => {
29        (Name => "Recover",
30          Test => (GlitchRecovery > 0),
31          Effect => (GlitchRecovery = max(0, GlitchRecovery - 1);
32                     t_g ++;),
33          Target => "Recovering"),
34        (Name => "Recovered",
35          Test => (GlitchRecovery <= 0),
36          Effect => (GlitchRecovery = 0;),
37          Target => "Cabling")
38      }
39    }
40    State Done {
41      Stop
42    }
43  };
```

Listing B.4: A slightly simplified form of the *CommHaul* TaskSim model.

```
1  Machine CommHaul(Var DistanceTravelled = 0, Var GlitchRecovery = 0) {
2    Var targetDist = 50;
3    Var step = normal(2, 0.25);
4    Var glitch_r = uniform (0,1);
5    Var glitch_p = 0.01;
6
7    VarRef glitchCost = ( glitch_r <= glitch_p )*100;
8
9    State Hauling {
10     Start
11     Arcs => {
12       (Name => "Haul",
13         Test => ( targetDist - DistanceTravelled > 0 && glitchCost <= 0),
14         Effect => (DistanceTravelled = min( targetDist , DistanceTravelled + step );
15                    t_g ++;),
16         Target => "Hauling"),
17       (Name => "Glitch",
```

```
18          Test => ( glitchCost > 0),
19          Effect => (GlitchRecovery = glitchCost ;),
20          Target => "Recovering"),
21        (Name => "Finished",
22          Test => ( targetDist - DistanceTravelled <= 0 && glitchCost <= 0),
23          Effect => (),
24          Target => "Done")
25      }
26    }
27    State Recovering {
28      Arcs => {
29        (Name => "Recover",
30          Test => (GlitchRecovery > 0),
31          Effect => (GlitchRecovery = max(0, GlitchRecovery - 1);
32                     t_g ++;),
33          Target => "Recovering"),
34        (Name => "Recovered",
35          Test => (GlitchRecovery <= 0),
36          Effect => (GlitchRecovery = 0;),
37          Target => "Hauling")
38      }
39    }
40    State Done {
41      Stop
42    }
43  };
```

Listing B.5: A slightly simplified form of the *CommSetup* TaskSim model.

```
1  Machine CommSetup(Var Progress = 0, Var GlitchRecovery = 0) {
2    Var step = normal(0.05, 0.01);
3    Var glitch_r = uniform(0,1);
4    Var glitch_p = 0.02;
5
6    VarRef glitchCost = ( glitch_r <= glitch_p )*200;
7
8    State Setting {
9      Start
10     Arcs => {
11       (Name => "Setup",
12         Test => (Progress < 1.0 && glitchCost <= 0),
13         Effect => (Progress = min(1.0, Progress + step );
14                    t_g ++;),
15         Target => "Setting" ),
16       (Name => "Glitch",
```

```
17          Test => (glitchCost > 0),
18          Effect => (GlitchRecovery = glitchCost ;),
19          Target => "Recovering"),
20        (Name => "Finished",
21          Test => (Progress >= 1.0 && glitchCost <= 0),
22          Effect => (),
23          Target => "Done")
24      }
25    }
26    State Recovering {
27      Arcs => {
28        (Name => "Recover",
29          Test => (GlitchRecovery > 0),
30          Effect => (GlitchRecovery = max(0, GlitchRecovery - 1);
31                     t_g ++;),
32          Target => "Recovering"),
33        (Name => "Recovered",
34          Test => (GlitchRecovery <= 0),
35          Effect => (GlitchRecovery = 0;),
36          Target => "Setting")
37      }
38    }
39    State Done {
40      Stop
41    }
42  };
```

Listing B.6: A slightly simplified form of the *HabMaint* TaskSim model.

```
1  Machine HabMaint(Var Progress = 0, Var GlitchRecovery = 0) {
2    Var step = normal(0.025, 0.01);
3    Var glitch_r = uniform(0,1);
4    Var glitch_p = 0.05;
5
6    VarRef glitchCost = ( glitch_r <= glitch_p )*20;
7
8    State Maintaining {
9      Start
10     Arcs => {
11       (Name => "Maintain",
12         Test => (Progress < 1.0 && glitchCost <= 0),
13         Effect => (Progress = min(1.0, Progress + step );
14                    t_g ++;),
15         Target => "Maintaining"),
16       (Name => "Glitch",
```

```
17          Test => (glitchCost > 0),
18          Effect => (GlitchRecovery = glitchCost ;),
19          Target => "Recovering"),
20        (Name => "Finished",
21          Test => (Progress >= 1.0 && glitchCost <= 0),
22          Effect => (),
23          Target => "Done")
24      }
25    }
26    State Recovering {
27      Arcs => {
28        (Name => "Recover",
29          Test => (GlitchRecovery > 0),
30          Effect => (GlitchRecovery = max(0, GlitchRecovery - 1);
31                     t_g ++;),
32          Target => "Recovering"),
33        (Name => "Recovered",
34          Test => (GlitchRecovery <= 0),
35          Effect => (GlitchRecovery = 0;),
36          Target => "Maintaining")
37      }
38    }
39    State Done {
40      Stop
41    }
42  };
```

Listing B.7: A slightly simplified form of the *SoilObservation* TaskSim model.

```
1  Machine SoilObservation(Var Progress = 0) {
2    Var step = normal(0.025, 0.01);
3
4    State Observing {
5      Start
6      Arcs => {
7        (Name => "Look",
8          Test => (Progress < 1.0),
9          Effect => (Progress = min(1.0, Progress + step );
10                    t_g ++;),
11         Target => "Observing"),
12       (Name => "Finished",
13         Test => (Progress >= 1.0),
14         Effect => (),
15         Target => "Done")
16     }
```

```
17   }
18   State Done {
19     Stop
20   }
21 };
```

Listing B.8: A slightly simplified form of the *SkyObservation* TaskSim model.

```
1  Machine SkyObservation(Var Progress = 0) {
2    Var step = normal(0.05, 0.01);
3
4    State Observing {
5      Start
6      Arcs => {
7        (Name => "Look",
8         Test => (Progress < 1.0),
9         Effect => (Progress = min(1.0, Progress + step);
10                   t_g ++;),
11        Target => "Observing"),
12       (Name => "Finished",
13        Test => (Progress >= 1.0),
14        Effect => (),
15        Target => "Done")
16     }
17   }
18   State Done {
19     Stop
20   }
21 };
```

B.2 CommTower Scenario

The following listings are simplified forms of the TaskSim (Section 8.1.3) models used in the mutable teams and complete proactive replanning system experiments (Chapters 6 and 8, respectively). In all models, t_g is the model's global time variable, which is used to synchronize the execution of the model with the rest of the simulator.

Listing B.9: A slightly simplified form of the *Move* TaskSim model.

```
1  Machine Move(Var DistanceRemaining = 60, Var GlitchRecovery = 0) {
2    VarRef step          = normal(1.0, 0.1);
3
4    State Moving {
```

```
5       Start
6       Arcs => {
7         (Name => "Move",
8          Test  =>  (DistanceRemaining > 0),
9          Effect  => (DistanceRemaining = max(DistanceRemaining − step, 0);
10                     t_g ++;),
11         Target  => "Moving"),
12        (Name => "Finished",
13         Test  =>  (DistanceRemaining <= 0),
14         Effect  => (),
15         Target  => "Done")
16      }
17    }
18    State Done {
19      Stop
20    }
21  };
```

Listing B.10: A slightly simplified form of the *Transport* TaskSim model.

```
1  Machine Transport(Var NumTransporters = 1,
2                    Var DistanceRemaining = 60,
3                    Var GlitchRecovery = 0) {
4    Var      step_r          = normal(0.5, 0.1);
5    VarRef step              = max(0, step_r );
6    Var      glitch_r        = uniform (0,1);
7    Var      glitch_p        = 0.0075;
8    VarRef glitchCost        = ( glitch_r  <= glitch_p )∗100;
9    Var     optRecoveryStep = 50;
10
11   State Hauling {
12     Start
13     Arcs => {
14       (Name => "Haul",
15        Test  =>  (DistanceRemaining > 0 && glitchCost <= 0
16                   && GlitchRecovery <= 0 && NumTransporters >= 1),
17        Effect  => (DistanceRemaining = max(DistanceRemaining − step, 0);
18                    t_g ++;),
19        Target  => "Hauling"),
20       (Name => "Noop",
21        Test  =>  (NumTransporters < 1),
22        Effect  => (t_g ++;),
23        Target  => "Hauling"),
24       (Name => "Glitch",
25        Test  =>  (( glitchCost  > 0 || GlitchRecovery > 0) && NumTransporters >= 1),
```

```
26          Effect  => (GlitchRecovery = max(GlitchRecovery,  glitchCost  );),
27          Target  => "Recovering"),
28        (Name =>  "Finished",
29          Test  =>   (DistanceRemaining <= 0 && glitchCost <= 0
30                      && GlitchRecovery <= 0 && NumTransporters >= 1),
31          Effect  => (),
32          Target  => "Done")
33      }
34    }
35    State  Recovering {
36      Arcs => {
37        (Name =>  "Recover",
38          Test  =>   (GlitchRecovery  > 0 && NumTransporters >= 1),
39          // Additional  transporters  massively  speed up recovery
40          Effect  => (GlitchRecovery = max(0, GlitchRecovery
41                                                  - 1.0
42                                                  - NumTransporters*optRecoveryStep
43                                                  + optRecoveryStep);
44                      t_g ++;),
45          Target  => "Recovering"),
46        (Name =>  "Noop",
47          Test  =>   (NumTransporters < 1),
48          Effect  => (t_g ++;),
49          Target  => "Recovering"),
50        (Name =>  "Recovered",
51          Test  =>   (GlitchRecovery  <= 0 && NumTransporters >= 1),
52          Effect  => (GlitchRecovery =  0;),
53          Target  => "Hauling")
54      }
55    }
56    State  Done {
57      Stop
58    }
59  };
```

Listing B.11: A slightly simplified form of the *Lift* TaskSim model.

```
1  Machine Lift(Var NumLifters = 1,  Var NumBracers = 0, Var  Progress  = 0)  {
2    Var       step_r          = normal(0.025,   0.01);
3    VarRef  step              = max(0,  step_r );
4    Var       slip_r          = uniform (0,1);
5    VarRef  slip_p            = (NumBracers <= 0)*0.02;
6    VarRef  slipOccurred  =  slip_r  <  slip_p ;
7
8    State   Lifting  {
```

```
9       Start
10      Arcs => {
11        (Name => "Lift",
12         Test  =>  (Progress < 1.0 && !slipOccurred && NumLifters >= 1),
13         Effect => (Progress = min(1.0, Progress + step);
14                    t_g ++;),
15         Target => "Lifting"),
16        (Name => "Noop",
17         Test  =>  (NumLifters < 1),
18         Effect => (t_g ++;),
19         Target => "Lifting"),
20        (Name => "Slip",
21         Test  =>  (slipOccurred && NumLifters >= 1),
22         Effect => (Progress = 0.0;
23                    t_g ++;),
24         Target => "Lifting"),
25        (Name => "Finished",
26         Test  =>  (Progress >= 1.0 && !slipOccurred && NumLifters >= 1),
27         Effect => (),
28         Target => "Done")
29      }
30    }
31    State Done {
32      Stop
33    }
34  };
```

Listing B.12: A slightly simplified form of the *Assemble* TaskSim model.

```
1  Machine Assemble(Var NumAssemblers = 1, Var Progress = 0, Var Pickup = 0) {
2    Var     step_r  = normal(0.05, 0.01);
3    VarRef  step    = max(0, step_r );
4    Var     drop_r  = uniform (0,1);
5    VarRef  drop_p  = (1 / NumAssemblers)*0.02;
6
7    Var     pickup_r    = normal(60, 5.0);
8    VarRef  pickupTime = (drop_r <= drop_p)*max(0, pickup_r );
9    Var     optPickupStep = 60;
10
11   State Assembling {
12     Start
13     Arcs => {
14       (Name => "Assemble",
15        Test  =>  (Progress < 1.0 && pickupTime <= 0
16                   && Pickup <= 0 && NumAssemblers >= 1),
```

```
17          Effect => (Progress = min(1.0, Progress + step);
18                     t_g ++;),
19          Target => "Assembling"),
20        (Name => "Noop",
21          Test => (NumAssemblers < 1),
22          Effect => (t_g ++;),
23          Target => "Assembling"),
24        (Name => "Drop",
25          Test => ((pickupTime > 0 || Pickup > 0) && NumAssemblers >= 1),
26          Effect => (Pickup = max(Pickup, pickupTime);),
27          Target => "PickingUp"),
28        (Name => "Finished",
29          Test => (Progress >= 1.0 && pickupTime <= 0
30                   && Pickup <= 0 && NumAssemblers >= 1),
31          Effect => (),
32          Target => "Done")
33      }
34    }
35    State PickingUp {
36      Arcs => {
37        (Name => "Pickup",
38          Test => (Pickup > 0 && NumAssemblers >= 1),
39          Effect => (Pickup = max(0, Pickup − 1 − (NumAssemblers − 1)∗optPickupStep);
40                     t_g ++;),
41          Target => "PickingUp"),
42        (Name => "Noop",
43          Test => (NumAssemblers < 1),
44          Effect => (t_g ++;),
45          Target => "PickingUp"),
46        (Name => "Picked",
47          Test => (Pickup <= 0 && NumAssemblers >= 1),
48          Effect => (Pickup = 0;),
49          Target => "Assembling")
50      }
51    }
52    State Done {
53      Stop
54    }
55  };
```

Listing B.13: A slightly simplified form of the *Cable* TaskSim model.

```
1  Machine Cable(Var NumCablers = 1, Var ConnectorsDone = 0) {
2    Var    numConnectors = 5;
3    Var    step_r        = normal(0.3, 0.25);
```

```
4   VarRef step            = max(0, step_r + (NumCablers − 1)∗0.1);
5   Var    drop_r          = uniform(0,1);
6   VarRef drop_p          = 0.05 − (NumCablers − 1)∗0.025;
7   VarRef dropOccurred    = drop_r < drop_p;
8
9   State Cabling {
10    Start
11    Arcs => {
12      (Name => "Cable",
13       Test => (numConnectors − ConnectorsDone > 0
14                && !dropOccurred && NumCablers >= 1),
15       Effect => (ConnectorsDone = min(numConnectors, ConnectorsDone + step);
16                  t_g ++;),
17       Target => "Cabling"),
18      (Name => "Noop",
19       Test => (NumCablers < 1),
20       Effect => (t_g ++;),
21       Target => "Cabling"),
22      (Name => "Drop",
23       Test => (dropOccurred && NumCablers >= 1),
24       Effect => (ConnectorsDone = floor(ConnectorsDone);
25                  t_g ++;),
26       Target => "Cabling"),
27      (Name => "Finished",
28       Test => (numConnectors − ConnectorsDone <= 0
29                && !dropOccurred && NumCablers >= 1),
30       Effect => (),
31       Target => "Done")
32    }
33  }
34  State Done {
35    Stop
36  }
37 };
```

Listing B.14: A slightly simplified form of the *LayCable* TaskSim model.

```
1  Machine LayCable(Var NumCablers = 1,
2                   Var DistanceRemaining = 60,
3                   Var SnagRecovery = 0) {
4
5   // The rate of progress is degraded by the amount of cable the
6   // agents are transporting. This degration is partially
7   // compensated for by adding additional agents. This works out to
8   // a minimum progress of 0.1 for a single cabler at distance 60
```

```
// from the goal.
VarRef overload      = max(0, DistanceRemaining
                              − (NumCablers >= 2)∗20
                              − (NumCablers == 1)∗15);
Var    step_r        = normal(0.0, 0.1);
VarRef step          = max(0, step_r + max(0.1, 1.0 − overload/50));
Var    snag_r        = uniform(0,1);
VarRef snag_p        = 0.0075;
Var    snagCost_r    = uniform(120, 130);
VarRef snagCost      = (snag_r <= snag_p)∗snagCost_r;
Var    snag_step     = normal(1, 0.1);
Var    optSnagStep   = 60;

State Laying {
  Start
  Arcs => {
    (Name => "Lay",
     Test => (DistanceRemaining > 0 && snagCost <= 0
              && SnagRecovery <= 0 && NumCablers >= 1),
     Effect => (DistanceRemaining = max(DistanceRemaining − step, 0);
                t_g ++;),
     Target => "Laying"),
    (Name => "Noop",
     Test => (NumCablers < 1),
     Effect => (t_g ++;),
     Target => "Laying"),
    (Name => "Snag",
     Test => ((snagCost > 0 || SnagRecovery > 0) && NumCablers >= 1),
     Effect => (SnagRecovery = max(SnagRecovery, snagCost);),
     Target => "Clearing"),
    (Name => "Finished",
     Test => (DistanceRemaining <= 0 && snagCost <= 0
              && SnagRecovery <= 0 && NumCablers >= 1),
     Effect => (),
     Target => "Done")
  }
}
State Clearing {
  Arcs => {
    (Name => "Clear",
     Test => (SnagRecovery > 0 && NumCablers >= 1),
     Effect => (SnagRecovery = max(0, SnagRecovery
                                   − max(0, snag_step)
                                   − NumCablers∗optSnagStep
                                   + optSnagStep);
```

```
                    t_g ++;),
      Target => "Clearing"),
    (Name => "Noop",
      Test => (NumCablers < 1),
      Effect => (t_g ++;),
      Target => "Clearing"),
    (Name => "Cleared",
      Test => (SnagRecovery <= 0 && NumCablers >= 1),
      Effect => (SnagRecovery = 0;),
      Target => "Laying")
  }
 }
 State Done {
  Stop
 }
};
```

Listing B.15: A slightly simplified form of the *Supply Habitat* TaskSim model.

```
Machine SupplyHabitat(Var NumSuppliers = 1,
                      Var DistanceRemaining = 60,
                      Var BoxesLoaded = 0) {
 Var    numBoxes       = 10;
 Var    step_r         = normal(0.0, 0.1);
 VarRef boxStep        = max(0, step_r + 1.0*NumSuppliers);
 VarRef moveStep       = max(0, step_r + 1.0*NumSuppliers);
 Var    drop_p         = 0.01;
 Var    drop_r         = uniform(0,1);
 Var    numDrop_r      = uniform(0,1);
 VarRef droppedBoxes   = (drop_r <= drop_p)
                         * ceil(numDrop_r * numBoxes)

 State Loading {
  Start
  Arcs => {
   (Name => 'Load',
    Test => (BoxesLoaded < numBoxes),
    Effect => (BoxesLoaded = min(BoxesLoaded + boxStep, numBoxes);
               t_g ++;),
    Target => 'Loading'),
   (Name => 'Loaded',
    Test => (BoxesLoaded >= numBoxes),
    Effect => (),
    Target => 'Hauling')
  }
```

```
27    }
28    State Hauling {
29      Arcs => {
30        (Name => 'Haul',
31         Test  =>  (DistanceRemaining > 0 && BoxesLoaded >= numBoxes),
32         Effect  => (DistanceRemaining = max(DistanceRemaining − moveStep, 0);
33                     BoxesLoaded = max(BoxesLoaded − droppedBoxes, 0);
34                     t_g ++;),
35         Target  => 'Hauling'),
36        (Name => 'Reload',
37         Test  =>  (BoxesLoaded < numBoxes),
38         Effect  => (),
39         Target  => 'Loading'),
40        (Name => 'Hauled',
41         Test  =>  (DistanceRemaining <= 0 && BoxesLoaded >= numBoxes),
42         Effect  => (),
43         Target  => 'Unloading')
44      }
45    }
46    State Unloading {
47      Arcs => {
48        (Name => 'Unload',
49         Test  =>  (DistanceRemaining <= 0 && BoxesLoaded > 0),
50         Effect  => (BoxesLoaded = max(BoxesLoaded − boxStep, 0);
51                     t_g ++;),
52         Target  => 'Unloading'),
53        (Name => 'Unloaded',
54         Test  =>  (DistanceRemaining <= 0 && BoxesLoaded <= 0),
55         Effect  => (),
56         Target  => 'Done')
57      }
58    }
59    State Done {
60      Stop
61    }
62  };
```

Listing B.16: A slightly simplified form of the *StowSupplies* TaskSim model.

```
1  Machine StowSupplies(Var NumStowers = 1, Var BoxesRemaining = 10) {
2    Var      step_r          = normal (0.5,  0.25);
3    VarRef step              = max(0, step_r + (NumStowers − 1)∗0.1);
4    Var      drop_r          = uniform (0,1);
5    VarRef drop_p            = max(0, 0.09 − (NumStowers − 1)∗0.06);
6    VarRef dropOccurred  = drop_r < drop_p;
```

```
7
8   State  Stowing {
9      Start
10     Arcs => {
11       (Name =>  "StowSupplies",
12        Test  =>   (BoxesRemaining > 0 && !dropOccurred && NumStowers >= 1),
13        Effect  => (BoxesRemaining = max(0, BoxesRemaining − step);
14                    t_g ++;),
15        Target  => "Stowing"),
16       (Name =>  "Noop",
17        Test  =>   (NumStowers < 1),
18        Effect  => (t_g ++;),
19        Target  => "Stowing"),
20       (Name =>  "Drop",
21        Test  =>   (dropOccurred && NumStowers >= 1),
22        Effect  => (BoxesRemaining = ceil(BoxesRemaining);
23                    t_g ++;),
24        Target  => "Stowing"),
25       (Name =>  "Finished",
26        Test  =>   (BoxesRemaining <= 0 && !dropOccurred && NumStowers >= 1),
27        Effect  => (),
28        Target  => "Done")
29     }
30  }
31  State  Done {
32     Stop
33  }
34 };
```