Abstract Reasoning for Multiagent Coordination and Planning

by

Bradley J. Clement

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2002

Doctoral Committee:
Professor Edmund H. Durfee, Chair
Professor John E. Laird
Professor Martha E. Pollack
Professor William C. Rounds
Professor Chelsea C White III

ABSTRACT

Abstract Reasoning for Multiagent Coordination and Planning

by

Bradley J. Clement

Chair: Edmund H. Durfee

As autonomous software and robotic systems (or *agents*) grow in complexity, they will increasingly need to communicate and coordinate with each other. These agents will need planned courses of action to achieve their goals while sharing limited resources. This dissertation addresses the problem of efficiently interleaving planning and coordination for multiple agents.

As part of my approach, I represent agents as having hierarchies of tasks that can be decomposed into executable primitive actions. Using task hierarchies, an agent can reason efficiently about its own goals and tasks (and those of others) at multiple levels of abstraction. By exploiting hierarchy, these agents can make planning and coordination decisions while avoiding complex computation involving unnecessary details of their tasks.

To reason at abstract levels, agents must be aware of the constraints an abstract task embodies in its potential decompositions. Thus, I provide algorithms that summarize these constraints (represented as propositional state conditions and metric resource usages) for each abstract task in an agent's library of hierarchical plans. This summary information can be derived offline for a domain of problems and used for any instance of tasks (or plans) assigned to the agents during coordination and planning. I also provide algorithms for reasoning about the concurrent interactions of abstract tasks, for identifying conflicts, and for resolving them.

I use these algorithms to build sound and complete refinement-based coordination

and planning algorithms. I also integrate summary information with a local search planner/scheduler, showing how the benefits can be extended to different classes of planning algorithms. Complexity analyses and experiments show where abstract reasoning using summary information can reduce computation and communication exponentially along a number of dimensions for coordination, planning, and scheduling in finding a single agent's plan or in optimally coordinating the plans of multiple agents. In addition, I provide pruning techniques and heuristics for decomposition that can further dramatically reduce computation. Overall, the techniques developed in this thesis enable researchers and system designers to scale the capabilities of interleaved coordination, planning, and execution by providing agents with tools to reason efficiently about their plans at multiple levels of abstraction.

To Heather, who gave the most to make this happen.

ACKNOWLEDGMENTS

Work reported in this thesis was supported by grants from NSF and DARPA while I worked as a graduate student and also by NASA when I joined the Artificial Intelligence Group at the Jet Propulsion Laboratory, California Institute of Technology. Ed Durfee, my adviser, provided the gentle guidance I needed to channel my efforts in the right direction. His idea of using summary information for coordination is the foundation for this dissertation.

The work on metric resource reasoning and experiments in the ASPEN Planning System was performed jointly with Tony Barrett and Gregg Rabideau at JPL. Tony helped me characterize the metric resource summarization algorithm, and Gregg helped me set up experiments to evaluate abstract reasoning within ASPEN. Both Gregg and Rus Knight helped me integrate abstract reasoning into ASPEN. I thank Steve Chien and the rest of the Artificial Intelligence Group for accommodating me during the last year of effort on my Ph.D.

At the University of Michigan, Pradeep Pappachan helped me with related work and provided many of the data structures upon which the implementation of my algorithms rests. I am also grateful to Jeff Cox and Thom Bartold for helping construct demonstrations of the software and suffering through Grid problems with me. Chris Brooks and Eric Klavins have provided a lot of feedback and suggestions along the way. I also thank my committee (John Laird, Martha Pollack, Bill Rounds, and Chip White) and Mike Wellman for their valuable comments. I am also indebted to Terence Kelly, who introduced me to LaTeX.

Of course, this dissertation would not exist without the support of my wife, Heather, our family, and friends. I am grateful for their encouragement and patience during this challenging time.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF APPENDICES

APPENDIX

PART I

Motivation and Foundations

CHAPTER 1

Introduction

As the world becomes more automated, single autonomous systems will increasingly need to communicate and coordinate with each other. In manufacturing, multiple spacecraft operation, and multi-objective military operations, goals of different decision-making entities (or *agents*) interact and require coordination. These agents may compete for common resources, such as parts and tools, physical space, a spacecraft's instrument, a common communication channel, an airport, transports, *etc.* In order to coordinate their actions, these agents must reason about both their individual and combined actions in several ways.

A single agent itself can have competing goals requiring the agent to carefully plan its actions. For example, a manufacturing agent may need to perform maintenance on machines that the agent is also using to manufacture parts from other parts. The agent must be careful not to plan maintenance on a machine at the same time it uses the machine to manufacture parts. Otherwise, the agent may damage the machine or the parts.

In addition to avoiding conflicts, an agent often also needs to plan its actions to *efficiently* achieve its goals. The manufacturing agent may have a deadline for manufacturing certain parts or may maximize profit by increasing throughput. If the agent can manufacture parts on some machines while performing maintenance on others, the agent can achieve its goals sooner.

At the same time, an agent must spend time to develop its plan. If an agent takes too much time planning, it may not have enough time to execute the plan. However, less efficient plans are often easier to discover than optimal plans, so an agent needs to both plan efficiently and balance its planning time with plan quality. If the manufacturing agent spends an appropriate time searching for efficient plans, it can ensure that it meets deadlines while minimizing the profit it sacrifices in foregoing an optimal plan.

If an agent must share resources with other agents, the agent may not be able to achieve its goals unless it coordinates with others. The manufacturing agent may be using parts that another inventory agent needs to move on and off the floor of the factory to meet shipping requirements. If the inventory does not move certain parts onto the floor early enough, the manufacturing agent's plan may not be feasible. Thus, not only do interacting agents need to be able to coordinate their plans to avoid conflicts, the agents need to be able to be able to make coordination decisions quickly while still ensuring that they can accomplish their goals efficiently.

However, first the agents must be able to recognize whether they need to coordinate, with whom they should coordinate, and over which resources they should coordinate. Thus, an agent must be aware of what resources it potentially uses to achieve its goals. The agent must also be aware of the needs and effects of other agents' actions to identify conflicts over shared resources. If these resource constraints are not communicated explicitly (maybe by inferring the constraints from the perceived actions of others), the agents may lack certain information, making coordination more difficult. The agents will also need to know when and how much of the resource each other uses in order to understand when conflicts arise.

Because interacting agents can execute their actions concurrently, they also need to be able to represent and reason about the temporal relationships of their actions. If the inventory agent is in the middle of transporting a part, the manufacturing agent cannot use that part until it is released by the inventory agent. Coordination may also require that agents reason about the durations of their actions. This is often necessary in spacecraft domains. If a Mars rover has a short window of opportunity to communicate with an overpassing orbiter, the two will need to carefully plan what information will be transmitted and the time that each transmission will take.

In addition, agents need mechanisms for resolving conflicts over shared resources. Such a mechanism may decide who gets to use a resource first or how much of the resource each agent receives. Another kind of coordination decision is choosing among optional methods for achieving a goal. If Mars rovers have optional paths to reach their intended destinations, they should coordinate their choices of paths to avoid collision while minimizing the distance they each travel.

This thesis addresses this problem of coordinating a group of agents that have either developed or need to develop plans to achieve their own separate goals.

1.1 Problem Statement

When agents try to concurrently accomplish separate goals in a shared environment with limited resources, they may need to coordinate and/or plan their actions carefully to avoid conflicts that could prevent them from reaching their goals. The agents may not know the plans, goals, or capabilities of others, but I assume that they can communicate these in a common language. For example, if one agent uses a transport resource called "transport1," the other agents agree on which specific transport resource that is, but they may not initially know how or when that transport will be used. Some agents may have already developed plans to meet their goals, and others may still need to plan for their goals.

The agents may also have preferences over alternative sequences and timings of states in the executions of their planned actions. I assume that these preferences can be elicited from some utility or cost function. The agents may associate great costs with plan failure (e.g. lives at stake in military operations), in which case if solutions exists, the agents must find one.

In addition, because the environment in which the agents must execute their plans is often uncertain, the agents cannot always guarantee that one particular course of action will achieve their goals. The agents also may not have time to replan their actions when one action fails. Thus, agents may need to plan for alternative courses of action to handle different potential environmental contexts and provide some means to recover from failure.

So, I propose to solve the problem of efficiently finding preferable elaborations or modifications to plans for a group of agents that will accomplish the goals of the agents and provide flexibility to handle an uncertain environment.

Thus, in essence, the coordination problem I address has the following characteristics:

- the agents have individual goals that require them to carefully plan their actions before execution;

- the agents have a shared environment with limited resources that are needed to accomplish their goals;
- the agents do not initially know the intentions of others;
- the agents' rewards depend on how and when they accomplish their goals (i.e. on the sequences and timings of states they witness while executing their planned actions); and
- unexpected events can cause agents' actions to fail.

The agents cannot always maximize their rewards for a particular coordination scenario. For example, if agents must accomplish their goals quickly, there may not be time to find optimally efficient coordinated plans. A general approach should have mechanisms that permit explicit tradeoffs between competing objectives. So, a coordination mechanism should allow agents to either sacrifice plan quality in order to reach agreement on their plans quickly or delay execution to find better ways to coordinate their plans.

This problem assumes that the agents need to develop their plans independently. Manufacturing agents working together in a factory may have different responsibilities, and the agents will receive separate rewards for achieving their goals even though some amount of cooperation is required to avoid conflicts. Thus, the agents may wish to develop their own plans to best achieve their own goals. The agents also may have physical restrictions that prohibit them from communicating their potential goals and plans. For example, a Mars rover and an orbiter may have a limited communication window (a few minutes in a day) for coordinating their science and communication needs. Because most of the information needed for determining capabilities and goals and for planning to achieve the goals is physically separate, planning is best done separately before the coordination window. Even if the rover could communicate its states and goals to the orbiter, the orbiter may not have the computation power to centrally plan for both of them in a reasonable amount of time. In addition, in military domains communicating all relevant information may present a large security hole. Therefore, interacting agents need to be able to develop separate plans when the agents are self-interested, the information is physically distributed, parallel processing is warranted, or the information must remain private. Because some of these constraints are limits on communication, the agents may additionally need to minimize communication costs during coordination.

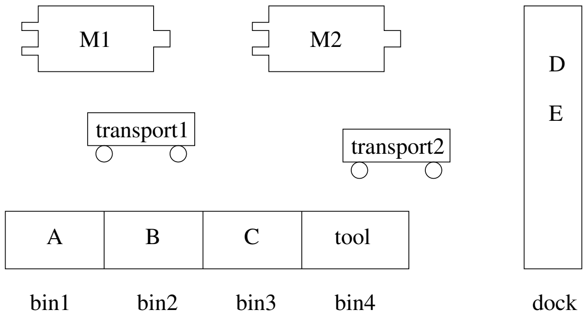


Figure 1.1: A simple example of a manufacturing domain

The foundations in this first part of the thesis apply to both coordination and single-agent planning problems. While Part II focuses on multiagent coordination, Part III addresses the single-agent planning problem specifically.

1.2 Manufacturing Example

The following example illustrates a particular problem to motivate this work and will be used throughout this work. Consider a manufacturing plant where a production manager, a facilities manager, and an inventory manager each have their own goals and have separately constructed hierarchical plans to achieve them. However, they still need to coordinate over the use of equipment, the availability of parts used in the manufacturing of other parts, storage for the parts, and the use of transports for moving parts. The state of the factory is shown in Figure 1.1. In this domain agents can produce parts using machines M1 and M2, service the machines with a *tool*, and move parts to and from the shipping dock and storage bins on the shop floor using transports. Initially, machines M1 and M2 are free for use, and the transports (transport1 and transport2) and all of the parts (A through E) shown in their storage locations are available.

The production manager is responsible for creating a G and an H part using machines M1 and M2. Both M1 and M2 can consume parts A and B to produce G. M2 can also produce H from G. The production manager's hierarchical plan for manufacturing H involves using the transports to move the needed parts from storage to the input trays of the machines, manufacturing G and H, and transporting H back to storage. This plan is shown in Figure 1.2. Arcs through subplan branches mean that the conjunction of subplans must be executed. Branches without arcs mean that only one plan must be executed

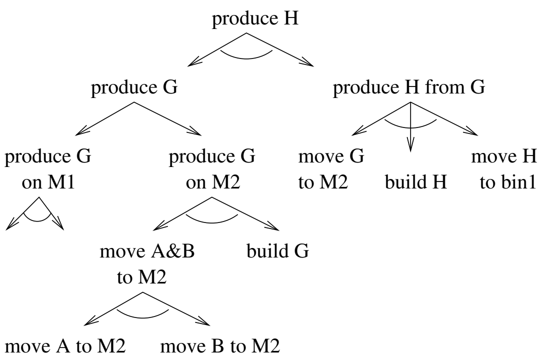


Figure 1.2: The production manager's hierarchical plan

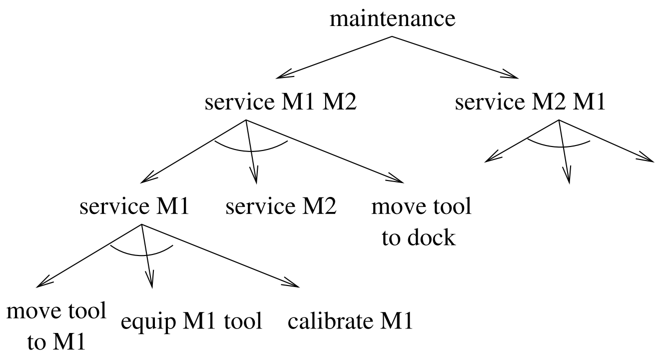


Figure 1.3: The facilities manager's hierarchical plan

successfully to achieve the goal of the parent. The decomposition of $produce\_G\_on\_M1$ is similar to that of $produce\_G\_on\_M2$.

The facilities manager must service each of these machines by equipping it with a tool and then calibrating it. The machines are unavailable for production while they are being serviced. The facilities manager's hierarchical plan branches into choices of servicing the machines in different orders and uses the transports for getting the tool from storage to the machines (Figure 1.3). The decomposition of $service\_M2M1$ is similar to that of $service\_M1M2$.

The parts must be "available" on the space-limited shop floor in order for an agent to use the machines to produce them. Whenever an agent moves or uses a part, it becomes unavailable. The inventory manager's goal is just to move part C to the dock and move D and E into bins on the shop floor (shown in Figure 1.4).

So what aspects of this problem map to the one described in the previous section? I outline them to correspond to the previously listed characteristics:

- The managers' plans use common resources, requiring them to coordinate or risk

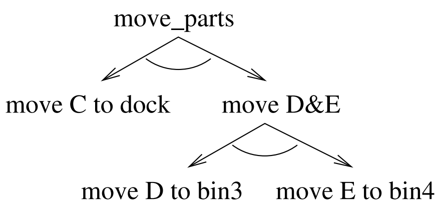


Figure 1.4: The inventory manager's hierarchical plan

resource conflicts leading to failure of achieving their goals.

- The managers developed their own plans and are ignorant of the intentions of the others.

- It may be that unless the managers quickly find plans that execute with the most concurrency, costs for unused resources and reduced throughput will cause profit loss.

- If the machine upon which the production manager builds part G unexpectedly fails, and the coordinated plans do not allow G to be built on the other machine, then the managers may not have time to replan their actions, resulting in plan failure and profit loss.

1.3 Summary of Approach and Results

Here I introduce my approach to the coordination and planning problem, introduce terminology used throughout the thesis to show how the approach applies to the example in the previous section, and describe the contributions of this work.

1.3.1 Summary of Approach

In short, my approach to solving the problem stated in Section 1.1 is to provide representations and algorithms for reasoning abstractly about the concurrent interactions of agents' plans that can leverage and extend the capabilities of different kinds of existing planning, scheduling, and execution systems. I provide agents with mechanisms that allow them to quickly reason at abstract levels so that they can make good decisions in the planning/coordination process when computation time is limited.[1]

While multiagent research has proposed many models of an agent, this dissertation draws on mature research in planning and scheduling to represent the capabilities of an agent, to model the environment with which the agent interacts, and to leverage algorithms for efficiently manipulating an agent's intended actions. Planning and scheduling research aims to provide general, robust, and efficient representations and algorithms for deciding what course of actions an agent can take to achieve a set of potentially conflicting objectives based on its current context. These actions are commonly represented according to the constraints and effects they have on states and resources describing the agent and its environment. Based on this information and the initial state of the environment, the planning problem is finding a sequence of actions that manipulate the state of the environment in a way that achieves a goal state and avoids violating the actions' constraints. By basing my approach to coordination on planning, the coordination problem additionally involves recognizing the need to resolve conflicts, identifying what needs to be resolved, identifying which agents should be involved, and finding ways to resolve their conflicts.

A hierarchical plan representation provides a domain expert a natural way of specifying multiple courses of action for an agent to achieve an abstract goal or perform an abstract task. Although the complexity of planning is intractable in general [Bylander, 1994], hierarchical representations promise the ability to solve problems efficiently by localizing subproblems and solving them in a divide-and-conquer manner. Another advantage is that an algorithm can reason about the problem at abstract levels and avoid more tedious manipulation of details deeper in the problem hierarchy. A group of agents can employ hierarchical task structures for efficient coordination. A central coordinating agent can use hierarchical planning techniques (see Section 2.3.2) to search for consistent coordinated plans by reasoning about the combined task hierarchies of the agents. Thus, my approach focuses on exploiting hierarchical plans.

My representation of concurrent hierarchical plans (called CHiPs) extends that of HTNs (Hierarchical Task Networks) [Erol *et al.*, 1994a] to include time durations as well as constraints and effects that occur during execution in addition to just preconditions

[1]However, I do not solve the problem of taking into account expected planning/coordination time in order to meet real time constraints.

and effects. The purpose of this is to enable agents to reason about concurrent execution. The representation allows algorithms to detect and resolve conflicts between plans whose execution intervals have relations such as *overlaps* or *during*. Because CHiPs (and HTNs) are based on the STRIPS representation [Fikes and Nilsson, 1971], which is subsumed by the representations of almost all current planners, the techniques introduced in this thesis can be applied to many planning systems.

These conditions and effects are conjunctions of constraints on propositional state variables. I do not represent disjunctive effects or universal quantification. I provide a discussion of the use of variables in predicates in Section 4.1, but they are not included in the formalisms of this thesis. These limitations in representational expressiveness are topics for future work.

The motivation for using hierarchical representations is not simply to make coordination and planning more efficient, but also to support another crucial application of hierarchical task concepts—specifically, flexible plan execution systems, such as PRS [Georgeff and Lansky, 1986], UMPRS [Lee *et al.*, 1994], RAPS [Firby, 1989], JAM [Huber, 1999], etc., that similarly exploit hierarchical plan spaces. These systems use hierarchical task representations similar to CHiPs and HTNs.[2] Rather than refine abstract tasks into a detailed end-to-end plan, these systems interleave refinement with execution. By postponing refinement until absolutely necessary, such systems leave themselves flexibility to choose refinements that best match current circumstances. However, this means that refinement decisions at abstract levels are made and acted upon before all of the detailed refinements need be made. If such refinements at abstract levels introduce unresolvable conflicts at detailed levels, the system ultimately gets stuck part way through a plan that cannot be completed. While backtracking to a decision point higher up in a task hierarchy is possible for an offline planner (since no actions are taken until plans are completely formed), it might not be possible when some (irreversible) plan steps have already been taken. It is therefore critical that the specifications of abstract plan operators be rich enough to summarize all of the relevant refinements to anticipate and avoid such conflicts. As a foundation for this thesis, I formally characterize methods for deriving and exploiting such rich summaries to support interleaved local planning, coordination,

[2]They also provide representations for iteration and monitoring. CHiPs do not include these language elements. Iteration loops, for example, would need to be unrolled into a multiple CHiP instantiations. Handling these representations is a subject of future work.

and execution.

To this end, I introduce algorithms to summarize the constraints and effects of tasks in the decomposition of an abstract task. This information captures the uncertainty of conditions for alternative decompositions (*or* branches) by keeping track of whether the conditions must or may hold in the abstract plan.[3] This information also captures uncertainty in the timing of conditions, whether these summary conditions hold at the beginning, at the end, sometimes (during execution), or always (throughout execution). Based on a model of concurrent hierarchical plan execution, the algorithm for computing summary conditions is proven sound and complete.[4]

In order for this work to apply to a wider class of planners and problem domains, I also give a representation of summarized metric resource usage and an algorithm to compute it for an abstract task. This represents the uncertainty in the amount and timing of usage with ranges of values for the minimum and maximum usages during execution and following execution. These summarization algorithms can be run offline to preprocess a library of hierarchical plans for a domain.

Sub-procedures of the summarization algorithms for both the summary conditions and summary resource usage determine interactions between abstract or primitive (non-abstract) tasks. These interactions describe how constraints are established and violated in other tasks. Thus, these procedures can be used to detect a conflict between a pair of tasks and changes that can resolve the conflicts. These changes involve eliminating or selecting alternative subtasks in an *or* branch and adding temporal constraints on the task intervals. The procedures culminate in a sound and complete algorithm (*CanAnyWay*) to determine whether a set of partially ordered tasks (abstract or primitive) are conflict free or may have conflicts. Another algorithm (*MightSomeWay*) can detect unresolvable conflicts soundly in set of partially ordered tasks and soundly and completely in a set of totally ordered tasks.

These algorithms can be integrated into a coordinator, planner, or scheduler so that conflicts can be resolved at abstract levels. As illustrated in Figure 1.5, by finding coordi-

[3] I often use "plan" for "task" in this thesis since an abstract task is a subgoal that is achieved by the plan induced by its refinement. Thus, a hierarchical plan can be decomposed into other hierarchical plans (subplans) for its subgoals.

[4] Soundness is the property of an algorithm that only generates solutions that achieve the algorithm's objective. Completeness is the property that if such solutions exist, the algorithm will generate one. Soundness and completeness, as used in this thesis, is explained in greater detail in [Russell and Norvig, 1995].

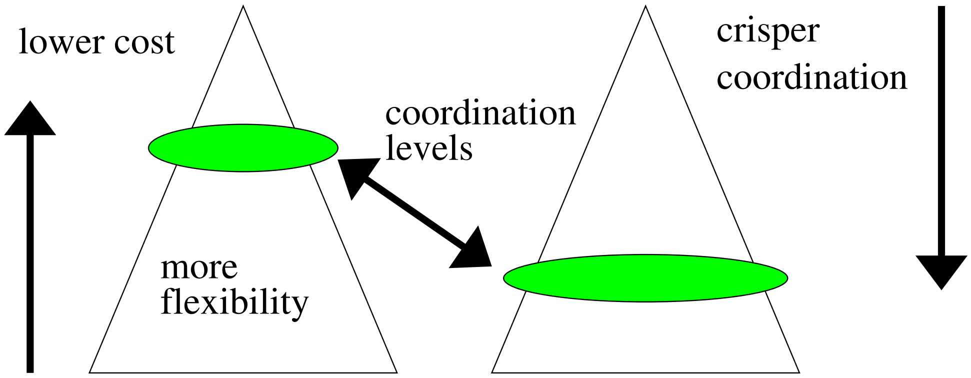


Figure 1.5: Tradeoffs of coordinating at multiple levels

nated plans at higher levels of abstraction, computation is minimized, and decomposition choices are preserved to enable robust execution systems to flexibly control and monitor an agent's actions in varying contexts while reacting to unexpected events and failed actions. At the same time, the ability to coordinate and plan at lower levels of abstraction enables an algorithm to find potentially better solutions by more crisply (or concurrently) synchronizing detailed actions. I present a sound and complete hierarchical coordination algorithm based on summary conditions that searches for coordinated plans by interleaving conflict resolution with refinement of the agents' plan hierarchies from the top down. The algorithm searches for preferable elaborations or modifications to the hierarchical plans of a group of agents that will accomplish the highest-level goals of the agents. These modifications include

- eliminating subtask choices for achieving higher-level goals or tasks and
- placing additional ordering constraints on primitive or abstract tasks within or across plan hierarchies.

For this algorithm, I assume that the plan hierarchies represent all feasible orderings of actions that the agents can take.[5] I analyze the complexity of the search algorithm at different levels of abstraction to evaluate the effectiveness of this approach, and I investigate how the cost of combined computation and execution can be optimized by coordinating at multiple levels of abstraction.

[5]Thus, just as in HTN planning [Erol *et al.*, 1994a], I do not consider modifying plans by inserting additional actions into the hierarchies (except for synchronization actions to enforce ordering constraints). However, there are no assumptions in the representations and algorithms upon which this coordinator builds that precludes their use within a planner that elaborates plans by inserting actions.

I also present search techniques for guiding the decomposition of the hierarchies, pruning the search space at abstract levels, and maximizing the utility of individuals or of a group of agents. I evaluate the heuristics for their effectiveness against state-of-the-art HTN planning heuristics in the application of the coordination algorithm to an evacuation domain. Other experiments in the manufacturing domain show how a domain designer can reduce communication overhead exponentially during coordination based on bandwidth, latency, and the level at which solutions are commonly found.

This thesis also investigates the use of summary information within different classes of single-agent planners. The coordination algorithm is altered to serve as a hierarchical refinement planning algorithm. I also integrate summary information into a local search planner/scheduler. I explore differences in complexity for the two kinds of planners and compare these to experiments in a multi-rover domain with a local search planner.

In showing the applicability of this approach to coordinating the plans (or planning) of a group (or single) agents, I demonstrate the coordination and planning in several problem domains. I apply the refinement-based coordination algorithm to an evacuation domain, where transports must bring evacuees from different locations back to safety points. I also apply the algorithm to coordinate the manufacturing agents described in Section 1.2. In addition, I describe how to wrap the coordination algorithm into an agent that continually coordinates requesting client agents in episodes. This multi-level coordination agent (MCA) is integrated with other agent-based technologies in a fictional military coalition scenario where the United Nations must maintain peace between nations warring over a territory in Africa. Finally, I apply abstract reasoning within ASPEN [Chien *et al.*, 2000b], an iterative repair planner, to schedule the actions of a team of rovers in a potential Mars surface exploration mission.

This dissertation does not study alternative coordination and negotiation protocols. There are already many techniques (e.g. voting, market mechanisms, ...) that could be used for negotiating over candidate coordinated plans (merged plans of a group of agents). Instead, I focus on recognizing the need to resolve conflicts, identifying what needs to be resolved, identifying which agents should be involved, and finding candidate solutions for different settlements. My approach enables agents to make tradeoffs among their competing objectives (e.g. minimizing computation, maximizing utility, preserving flexibility), but specific protocols are used to determine how agents make these tradeoffs.

This thesis also does not offer new methods for reactive execution. Instead, I show how agents can reason abstractly to preserve plan flexibility for existing execution systems to exploit.

1.3.2 Example

Here I show how my approach applies to the manufacturing problem example of Section 1.2 and informally introduce terminology used throughout the dissertation.

Suppose that the factory managers in the example wish to coordinate their plans. Each agent could have preprocessed its plan to derive summary information for every abstract plan operator in the hierarchy. For a particular abstract plan, this information includes the summary pre-, post-, and inconditions that correspond to the *external preconditions*, *external postconditions* and *internal conditions* respectively of the plan based on its potential refinements. Below I show a subset of the summary conditions for the production manager's top-level plan. (I give the complete set of summary conditions for many of the managers' plans in Section 4.1 and Appendix A.) Following each literal are modal tags for *existence* and *timing* information. "Mu" is *must*; "Ma" is *may*; "F" is *first*; "L" is *last*; "S" is *sometimes*; and "A" is *always*.

Production manager's *produce_H* **plan:**

Summary preconditions:

`available(A)MuF, available(M1)MaS, available(M2)MaS`

Summary inconditions:

`available(M1)MaS, available(M2)MuS, available(G)MuS, ¬available(A)MuS, available(A)MuS, ¬available(M1)MaS, ¬available(M2)MuS, ¬available(G)MuS, available(H)MuS, ¬available(H)MuS`

Summary postconditions:

`¬available(A)MuS, available(M1)MaS, ¬available(G)MuS, available(M2)MuS, available(H)MuL`

I now describe the meaning of some of these conditions and how they are derived. I revisit this example is Section 4.1, where the derivation is fully explained. $available(M2)$ is a *must* precondition of the top-level plan, *produce_H*, because no matter how the hierarchy is decomposed, M2 must be used to produce H, so $available(M2)$ must be established externally to the production manager's plan. However, $available(M1)$ is a *may* precondition because the production manager may not use M1 if it chooses to use M2 instead to produce G. $available(A)$ is a $first$ summary precondition because part A is needed at the beginning of execution when it is transported to one of the machines. Since the machines are not needed at the very beginning when parts are being transported, they are not $first$ but *sometimes* conditions since they are needed at some point in time during execution.

$available(G)$ is not an external precondition because, although G is needed to produce H, G is supplied by the execution of the *produce_G* plan. Thus, $available(G)$ is met *internally*, making $available(G)$ an internal condition. Another summary incondition (representing an internal need or effect) of *produce_H* is $available(M1)$. $available(M1)$ is an external precondition, an internal condition, and an external postcondition because it is needed externally and internally; it is an effect of $produce\_G\_on\_M1$ which releases M1 when it is finished; and no other plan in the decomposition undoes this effect. It is a *may* condition because G may be manufactured on M2. $\neg available(M1)$ is also a summary incondition of *produce_H* because M1 may be used to produce G. Having both $available(M1)$ and $\neg available(M1)$ as inconditions is consistent because they are *sometimes* conditions, implying that they hold at different times during the plan's execution. So, these conditions would conflict if they were both *must* and *always* (meaning that they must always hold throughout the plan's execution).

$\neg available(A)$ is a *must* summary postcondition of the top-level plan because A will definitely be consumed by *make_G* and is not produced by some other plan in the decomposition of $produce\_H\_from\_G$. Even though $available(G)$ is an effect of *produce_G*, it is not an external postcondition of *produce_H* because it is undone by $produce\_H\_from\_G$, which consumes G to make H. $available(H)$ is a *last* summary postcondition because the production manager releases H at the very end of execution. $available(M2)$ is not last because the manager finishes using M2 before moving H into storage.

With summary information derived for the managers' plan hierarchies, the production and inventory managers could send the summary information for their top-level plans to the facilities manager. The facilities manager could then reason about the top-level summary information for each of their plans to determine that if the facilities manager serviced all of the machines before the production manager started producing parts, and the production manager finished before the inventory manager began moving parts on and off the dock, then all of their plans *can* be executed (refined) in *any way*, or *CanAnyWay*. Then the facilities manager could instruct the others to add communication actions to their plans so that they synchronize their actions appropriately.

This top-level solution maximizes robustness in that the choices in the production and facilities managers' plans are preserved, but the solution is inefficient because there is no concurrent activity—only one manager is executing its plan at any time. The production manager might not want to wait for the facilities manager to finish maintenance and could negotiate for a solution with more concurrency. In that case, the facilities manager could determine that they could not overlap their plans in any way without risking conflict ($\neg$*CanAnyWay*). However, the summary information could tell them that there might be some way to overlap their plans (*MightSomeWay*). Then, the facilities manager could request the production manager for the summary information of each of *produce_H*'s subplans, reason about the interactions of lower level actions in the same way, and find a way to synchronize the subplans for a more fine-grained solution where the plans are executed more concurrently.

1.3.3 Contributions

This dissertation extends work multiagent coordination, planning, and scheduling by enabling agents to reason abstractly about their plans. Much of the work in the planning community has traditionally focused on plan generation. Pollack argues that this is not enough to handle the demands of an uncertain, dynamic world, which requires an agent to also be able to manage its generated plans in a number of ways [Pollack and Horty, 1999]. While this thesis describes how different classes of planners can generate plans more efficiently, it concentrates on methods for agents to efficiently manage their plans when the plans interact with each other.

The algorithms for deriving and reasoning about summary information serve as a formal toolbox that can be used to enable abstract planning, scheduling, and coordination in different classes of existing systems. Complexity analyses and experiments explain and quantify the benefits of abstract reasoning so that researchers can estimate the potential performance improvements of integrating summary information into their own coordination/planning/scheduling systems. These benefits are not obvious since summary information for a task can grow exponentially up the hierarchy, potentially complicating the search for solutions to a problem.

The multi-level coordination algorithm extends the capabilities of existing approaches (see Chapter 2) by including interval temporal reasoning about hierarchical tasks that may execute concurrently. In addition, the ability to reason at abstract levels can greatly improve computation and communication performance by orders of magnitude to apply to even larger problem domains. The algorithms for summarizing metric resource usage in addition to propositional state variables and the adaptation of ASPEN, a local search planner, to exploit this summary information serves as a model for integrating this approach into a wide variety of planning and scheduling systems. The ability to find solutions at abstract levels and preserve alternative decompositions for achieving subgoals enables robust execution systems, such as PRS [Georgeff and Lansky, 1986], UMPRS [Lee *et al.*, 1994], RAPS [Firby, 1989], JAM [Huber, 1999], etc., to handle more situations and unexpected events when decompositions depend on the state and to recover from the failure of one branch of decomposition by choosing another.

This approach enables agents to discover potential conflicts among their plans without revealing details by controlling the summary information they communicate during coordination. During coordination (and planning) agents can discover at abstract levels which tasks are not involved in conflicts and avoid refining them unnecessarily and causing the search space to grow. By finding solutions much more quickly at abstract levels, the agents can have a solution ready in case there is no time to further coordinate. If the agents need more efficient solutions that can only be found at deeper levels within the hierarchy, the abstract solutions can be used to prune the search space where solutions are guaranteed to be worse.[6] Summary information can also be used to detect and prune search spaces where conflicts are unresolvable. These search techniques and heuristics for guiding the decomposition of plan hierarchies are another contribution of this thesis.

Complexity analyses and experiments show a number of ways agents can reduce computation and communication exponentially. Previous work shows that hierarchical prob-

[6]Note that just as it is not the purpose of this thesis to specify particular negotiation protocols, I also do not introduce methods for determining *when* agents need these more efficient solutions. Instead I provide the *ability* to find better solutions using efficient decomposition techniques.

lem solving can reduce the size of the search space exponentially when subproblems do not interact [Korf, 1987; Knoblock, 1991]. Summary information enables these speedups even with interacting subproblems. At higher levels of abstraction, the coordinator (and planner) reasons about exponentially fewer actions and potentially exponentially fewer constraints compared to lower levels. This reduction in information translates into exponentially fewer and less complex planning and scheduling operations. I give analyses both for refinement-based coordination and planning and for iterative repair planning to show how different classes of planners and schedulers can benefit from abstract reasoning based on summary information. Experiments show how these benefits are realized in a number of domains and that the search techniques and heuristics enable multi-level coordination and planning to find optimal solutions at lower levels faster than state-of-the-art HTN heuristics. They also show where reasoning at abstract levels fails to improve search performance.

In summary, the work presented here makes the following contributions:

- Algorithms for deriving summary information for propositional conditions and metric resource usage and for determining potential and definite interactions of abstract tasks;

- Sound and complete algorithms for concurrent hierarchical refinement planning and plan coordination supporting flexible plan execution systems;

- The integration of abstract reasoning using summary information in a local search planner/scheduler, showing applicability to other classes of planners;

- Search techniques and heuristics, including *fewest-threats-first* (FTF) and *expand-most-threats-first* (EMTF), for both hierarchical refinement and local search planners/schedulers that take advantage of summary information to prune the search space; and

- Complexity analyses and experiments showing that the search for abstract solutions at higher levels is exponentially less expensive than at lower levels for both hierarchical refinement and local search planners/schedulers.[7]

[7] In doing this I also prove that resolving threats among partially ordered STRIPS operators is NP-complete.

The collection of algorithms presented here for summarization, abstract reasoning, coordination, planning, and scheduling enables researchers and system designers to

- construct new coordination and planning systems,
- expand and improve existing ones, and
- apply those presented here to many domains where an agent or group of agents must reason (collectively) about their actions to achieve short and long-term goals.

1.4 Overview

This dissertation is broken down into four parts. The remainder of Part I describes related work and introduces the representation of hierarchical plans, the summarization of propositional state and metric resource constraints, and mechanisms for reasoning about the interactions of summarized abstract plans. Part II describes how these mechanisms are integrated into a coordination algorithm. It also gives analyses and experiments evaluating multi-level coordination and decomposition heuristics. Part III explains how the benefits of the coordination algorithm are also realized in a refinement-based single-agent planner. In addition, it describes how an iterative repair planner takes advantage of summary information through analyses and experiments and follows with a comparison of the advantages of abstract reasoning in different classes of planners. Finally, Part IV summarizes results and describes future research directions.

More specifically, Chapter 2 explains how others have addressed this problem or variations and where my approach extends this work. Chapter 3 describes the concurrent hierarchical plan (CHiP) representation, formalizes the concurrent execution of CHiPs, and describes different kinds of metric resource usage. Chapter 4 formalizes summary information for propositional state conditions and metric resource usage. As part of the summarization algorithm, definitions and algorithms for determining *achieve*, *clobber*, and *undo* interactions among abstract tasks are presented with pointers to appendices for soundness and completeness proofs. Chapter 5 explains how these algorithms can be used to detect conflicts in set of plans with ordering constraints, whether the plans are threat-free, or whether there is an unresolvable conflict.

In Part II, Chapter 6 shows how these algorithms for reasoning about a set of partially ordered plans can be integrated into a coordination algorithm. It also presents search techniques that the coordination algorithm can use to efficiently guide decomposition and prune the search space. This is followed by complexity analyses explaining how reasoning at higher levels of abstraction can doubly exponentially reduce the search space in the worst case that summarization does not reduce the size of information and showing how the combined cost of computation and execution can be optimized by coordinating at appropriate levels of abstraction. Chapter 7 reports experiments in an evacuation domain that evaluate the EMTF and FTF heuristics compared to other state-of-the-art HTN heuristics in finding optimally coordinated plans. Another experiment in the manufacturing domain shows how a domain designer can exponentially reduce communication delay by altering the granularity at which agents exchange summary information. Chapter 7 also describes how the coordination algorithm is wrapped in a multi-level coordination agent that continually coordinates a group of military coalitions in a fictional peace-keeping scenario, that integrates many agent-based technologies.

Part III focuses on the single-agent planning problem. Chapter 8 describes a refinement-based hierarchical planner based on the coordination algorithm that similarly exploits the advantages of summary information. It then describes how summary information is used in a local search planner (ASPEN) with variations of the EMTF and FTF search heuristics and analyzes scheduling complexity. Chapter 9 reports experiments with ASPEN for a team of Mars rovers that verify the analyses and point out where abstract reasoning can be inefficient. Section 9.3 compares the complexity advantages of summary information in the refinement planner, local search planner, and other kinds of planners.

Part IV concludes by summarizing results and contributions in Chapter 10 and identifies new research needs in Chapter 11.

CHAPTER 2

Background and Related Work

Here I give background on research that this dissertation builds on and describe how other work has addressed the problem of this thesis (given in Section 1.1).

2.1 Multiagent Coordination

The approach I have taken for abstract reasoning draws on earlier work involving a hierarchical behavior-space search where agents represent their planned behaviors at multiple levels of abstraction [Durfee and Montgomery, 1991]. Distributed protocols are used to decide at what level of abstraction coordination is needed and to resolve conflicts there. This approach capitalizes on domains where resources can be abstracted naturally. This earlier work can be viewed as a very limited, special case of this thesis. It is justified only intuitively and with limited experiments and analyses.

In closer relation to my approach, Pappachan shows how to interleave hierarchical plan coordination with plan execution for cooperative agents using an online iterative constraint relaxation (OICR) algorithm [Pappachan, 2001]. OICR uses a disjunctive temporal constraint network to keep track of task interactions among agents at the primitive level and derive legal temporal orderings of abstract tasks. A protocol enables agents to commit to legal orderings at abstract levels from the top of their hierarchies down until the necessary temporal commitments are made to ensure consistency across the agents plans. Like my approach, coordination can be achieved at higher levels of abstraction for more flexible execution, or the agents can decompose their tasks to lower levels to for tighter coordination that can improve plan quality. In focusing on efficient interleaved coordination and execution, completeness of the algorithm can be sacrificed to avoid backtracking up the hierarchy (breaking commitments), and the only computation needed at execution time involves updating the temporal constraint network based on ordering commitments.[1]

However, effort is needed up front to build the initial constraint network for the agents' task hierarchies. In my approach, the preprocessing of hierarchies (deriving summary information) can be done by the agents separately before they encounter each other. Thus, there are no extra costs of agents joining or leaving a coordination group while the temporal constraint network must be rebuilt for OICR in such cases.

The main difference between the OICR approach and using summary information is that OICR effectively propagates legal temporal ordering constraints while I propagate constraints up the hierarchy. The OICR approach is tailored towards efficiently interleaving coordination and flexible execution at the price of completeness while my coordination algorithm is aimed at efficient, complete interleaved coordination and planning at the price of potentially delaying execution due to backtracking. Another difference is that it is unclear how metric resource usage constraints can be efficiently integrated into OICR's temporal constraint network while this is explored in detail in this thesis.

At the same time, summary information could complement OICR. The temporal constraint network could be built with abstract tasks using summary information and expanded whenever a decomposition is needed. This could bring computational advantages to updating the temporal constraint network because the network will be smaller when tasks are not fully decomposed.

TÆMS (a framework for Task Analysis, Environment Modeling, and Simulation) takes a different approach to abstract reasoning by allowing the domain modeler to specify a wide range of relationships for abstract or primitive tasks [Decker, 1995]. This work offers quantitative methods for analyzing and simulating agents as well as their interactions. While not all of these interactions can be represented and discovered using summary information, summary information is automatically discovered through analysis in my approach. I build on this work by showing how coordination can exploit abstraction for more efficient search for conflict resolutions. However, my approach does not currently use quantitative representations for reasoning about progress and the achievement

[1] The sacrifice in completeness is exhibited by only considering temporal orderings at the abstract level that will lead to coordinated plans no matter which decomposition branches the agents choose. This is the *CanSomeWaySynchronize* property to which Section 5.1 refers.

of a subset of goals.

Social laws have also been used to resolve conflicts among agents [Shoham and Tennenholtz, 1992]. A social law is a mechanism for resolving a conflict over a shared resource among a group of agents. If agents are committed to abide by social laws, and these laws have a large enough scope to prevent such conflicts for all of the different kinds of agent interactions that may occur, then an agent can expect to achieve its goals by executing its plan according to these laws without coordinating with others. However, often the efficiency of plans is unnecessarily sacrificed when laws are not tailored for handling many of the potentially large numbers of cases of agent interactions. In other words, it would be difficult to define social laws that specify the efficient alternative actions each agent can take for each possible combination of conflicts over any combination of shared resources for any potential plan of the agent and those of the other agents involved. In addition, for some domains, the interactions can be so complicated to the point that there are no social laws that can generally resolve conflicts (i.e. social laws are not complete in resolving conflicts). It is also difficult to avoid cases where agents are unnecessarily coordinating according to a social law even though no conflicts exist. For the example in Section 1.2, if a social law requires that the facilities manager wait until after 5pm to service machines so that the production manager can manufacture parts without interruption during the day, it would be inefficient for the facilities manager to postpone servicing machines if the production manager had no plans to manufacture parts that day. My approach to handling this problem suggests that the agents should efficiently search for tailored social laws to resolve present and future conflicts in the context of their short and long-term plans while maximizing local or global utility.

Grosz's shared plans model of collaboration presents a theory for modeling multi-agent belief and intention [Grosz and Kraus, 1996]. While the shared plans work is directed towards cooperative agents, it represents action hierarchies and provides mental models at a higher level than represented in this thesis. However, my use and analysis of summary information complements Grosz's work by providing a way to automatically represent and efficiently reason about the intentions of agents at multiple levels of abstraction. Future work is needed to understand how summary information can be bridged with mental states of agents to exploit the techniques employed in shared plans and other work based on BDI (belief-desire-intention) models of agents [Rao and Georgeff, 1995].

TEAMCORE is a system based on similar models of shared intentions that integrates team coordination and communication [Pynadath *et al.*, 1999; Tambe, 1997]. Teams are built using a team-oriented programming (TOP) framework, that allows a programmer to build an agent-based system by specifying an agent organization hierarchy and team tasks. TEAMCORE allows for robust, flexible execution, monitoring, and failure recovery for a group of collaborative agents. This work assumes that agents are cooperative and focuses on the level of execution instead of the level of deliberative planning that this dissertation investigates. My algorithms for deriving and using summary information present an alternative to building team plans. Instead of a user programming the tasks and team organization while working out shared resource conflicts, the user can just model the task plans individually, and the coordination software can automatically generate consistent team plans that can be adapted for use within TEAMCORE. However, research is needed to understand how hierarchical plans and the algorithms that generate and coordinate them can be adapted so that agents can take advantage of the execution monitoring and failure recovery of TEAMCORE.

Work in Generalized Partial Global Planning (GPGP) investigates how to combine different coordination strategies to approach specific domains [Decker, 1995]. Combining different techniques for sharing information and establishing commitments based on the TÆMS task representation can minimize communication overhead. While this thesis does not evaluate specific coordination strategies, the algorithms that use summary information to determine interactions of tasks at multiple levels of abstraction automatically determine where conflicts exist and what plan information should be communicated to resolve them. By decomposing tasks to necessary levels of abstraction, agents using my approach can automatically minimize the information needed to complete coordination. In TÆMS, the domain expert must specify the constraints of each abstract task in order to minimize communication in the same fashion.

The hierarchical plan representation (CHiP) described in Chapter 3 is constructed to be compatible with many other planning and execution task representations. The formalization of the concurrent execution of CHiPs tries to distill appropriate aspects of theories for multiagent action and reasoning [Georgeff, 1984; Fagin *et al.*, 1995]. As mentioned earlier, future work is needed to extend this agent model to take advantage of techniques developed for other models of interacting agents [Grosz and Kraus, 1996;

Rao and Georgeff, 1995; Tambe, 1997].

2.2 Plan Merging

The term "plan merging" has taken several meanings in previous work:

1. merging redundant tasks within a single plan that achieve the same effect in separate subgoals,
2. incorporating plans for new goals into a plan that achieves current goals,
3. distributing the goals of a planning problem to agents that cooperatively solve them (merging solutions), and
4. resolving conflicts among the plans of separate agents (merging into a coordinated plan).

These are all closely related, and merging techniques that address one of those will likely apply to or complement the others. This is because merging plans (whether of one agent or of a group of agents) involves resolving conflicts and/or eliminating redundant actions using existing planning techniques. Likewise, reasoning about temporal concurrency of multiagent interactions can be applied to parallel actions of a single agent.

The fourth form of plan merging is the one addressed in this dissertation. For this problem, conflicts can be avoided by reducing or eliminating interactions by localizing plan effects to particular agents [Lansky, 1990], and by merging the individual plans of agents by introducing synchronization actions [Georgeff, 1983]. In fact, planning and merging can be interleaved, such that agents can propose next-step extensions to their current plans and reconcile conflicts before considering extensions for subsequent steps. By formulating extensions in terms of constraints rather than specific actions, a "least commitment" policy can be retained [Ephrati and Rosenschein, 1994]. These techniques all assume that actions are non-hierarchical and atomic, meaning that an action either executes before, after, or at exactly the same time as another. Hence, they do not represent the possibility of several actions of one agent executing during the execution of one action of another agent. While not focused on multiagent applications, there are several planning systems that can reason about such concurrent temporal interactions [Allen *et al.*, 1991; Laborie and Ghallab, 1995; Muscettola, 1994; Rabideu *et al.*, 1999].

Techniques directed at the second plan merging problem include reasoning about concurrent actions with temporal extent [Tsamardinos *et al.*, 2000]. This work combines *or* branches and simple temporal networks (STNs) into conditional simple temporal networks (CSTNs) that are used to identify conflicts between plans and resolve them using constraint satisfaction. My approach uses less expressive temporal constraints than those of a simple temporal network, but the CSTN approach does not exploit task decomposition. To extend my approach to handle simple temporal networks would only involve substituting the temporal constraint algorithm (see Section 4.3.1) with one that can handle STNs [Meiri, 1992].

Corkill studied interleaved planning and merging (the fourth form) in a distributed version of the NOAH hierarchical planner [Corkill, 1979]. He recognized that, while most of the conditions affected by an abstract plan operator might be unknown until further refinement, those that deal with the overall effects and preconditions that hold no matter how the operator is refined can be captured and used to identify and resolve some conflicts. He recognized that further choices of refinement or synchronization choices at more abstract levels could lead to unresolvable conflicts at deeper levels, and backtracking could be necessary. The work presented here is directed toward avoiding such backtracking by improving how an abstract plan operator represents all of the potential needs and effects of all of its potential refinements (in summary conditions).

2.3 Planning

In this section I discuss other planning techniques upon which my approach builds. First I describe different classifications of planners as background for the comparison of planning techniques in this dissertation. Then I describe related work in hierarchical planning in Section 2.3.2.

2.3.1 Refinement and Local Search Planning

This thesis investigates the differences in using summary information in refinement and local search planners in Chapters 8 and 9. Here I describe classifications of different planning algorithms and ASPEN (a local search planner) as background for understanding these chapters. For a lengthier discussion about planning algorithms and search, see [Russell and Norvig, 1995].

In order to understand the differences between classes of planners, one must first understand how planning is based on search. A search algorithm defines a state of the search, a set of search operators for generating new search states from an existing one, and the criteria for a solution state. Search starts with an initial state, tests to see if the state is a solution, chooses a search operator to generate new search states, and picks a new state to explore. With the new state the process is repeated typically until a desired solution is found. For example, a search algorithm for solving a Rubik's Cube[2] puzzle could define the search state as the colors on each face of the cube. Search operators could be turning a layer of the cube, and the solution state is reached when the faces each have one color.

A planning problem includes an initial state of the environment[3], a goal[4], and a set of actions that have constraints and effects on the state of the environment. A planner finds a collection of actions with ordering constraints that takes the agent from the initial state to a state that satisfies the goal. Planners are special kinds of search algorithms. Planners define the search state either as a state of the environment or as a partially elaborated plan. The former is a *state-space planner*, and the latter is a *plan-space planner*.

A state-space planner's initial search state is the initial state of the environment. Search operators for the state-space planner include adding an applicable action to the plan that generates the next state of the environment from the current state, removing an action from the plan, or replacing an action with another applicable action. An action is applicable if its constraints agree with the current state. The operator that adds an action is a refinement operator, and the others are modification operators. A solution is found

[2] ©Rubik/Seven Towns

[3] Note that there are two kinds of "states" mentioned here: the state of the environment (that the agent senses and manipulates) and the state of the search (a construct in the search algorithm).

[4] A goal is a partial specification of the state of the environment. Often we speak of an agent having multiple goals. The goal mentioned here can represent multiple goals as a conjuction of subgoals.

when the resultant search state generated from the actions satisfies the goal. A regression state-space planner works similarly from the goal back to the initial state.

A plan-space planner's search state is a plan. Initially this plan is already instantiated with actions or is empty. Search operators typically add, remove, or order actions so that their constraints are met. Refinement operators elaborate the plan by adding actions and constraints. Modification operators remove or substitute actions and constraints. A constraint could be a temporal ordering of the actions or a binding of a value to a variable in an actions conditions and effects. A solution is found when there are no constraint violations, and the goal state is achieved by the effects of the actions.

A planner is also classified along another dimension as either a *refinement planner* or a *local search planner*. A refinement planner elaborates a plan by using refinement operators to successively add details about how the goal will be accomplished. Typically the refinement planner is complete in that it will find a solution if one exists, but it must keep track of all search states expanded, which can cause memory to grow. Completeness is achieved by *backtracking*. Whenever the search ends up in a search state where no there are no applicable operators, and the goal has not been achieved, it backtracks to another expanded state, and continues the search. The search terminates when the goal has been achieved, and all constraints have been met or when the search fails because there are no more search states to explore.

A local search planner typically keeps little or no memory of previous search states and, thus, requires only a constant amount memory. While it may elaborate a plan with refinement operators, a local search planner also uses modification operators and often starts out with a flawed plan that it modifies. Although these planners can potentially revisit search states, they often choose operators stochastically and eventually break out of the local search space to find other plans. The search terminates when the goal is satisfied. The search is not guaranteed to find a solution (i.e. is not complete) and may never terminate (especially if no solution exists) unless the algorithm explicitly "gives up" with a timer, for example.

While hierarchical planners commonly take a least commitment, refinement approach to problem solving, research in the operations research community illustrates that a simple local search is surprisingly effective [Papadimitriou and Steiglitz, 1998]. A heuristic iterative repair planner is a type of local search plan-space planner that focuses on repairing flaws. It starts with an initial (usually flawed) plan and iteratively chooses a flaw, chooses a modification (repair) operator, and changes the plan by applying the operator. It can also use these operators to optimize the plan according to specified criteria. Because it employs local search, the iterative repair planner never backtracks. Since taking a random walk through a large space of plans is inefficient, heuristics guide the choices (of flaws and repair operators) by determining the probability distributions for each choice. I build on this approach to planning by studying the integration of summary information into the ASPEN planner [Chien *et al.*, 2000b]. ASPEN assigns actual time values to start and end times of actions, so the plan (or schedule) is always fully ordered. (In contrast, many refinement plan-space planners search through partially ordered actions, only specifying an ordering between actions to resolve a conflict.) ASPEN allows for abstract tasks, and one of its repair operators is to abstract a hierarchy so that it can potentially choose a different decomposition branch. Parameters of a task (that are used to specify constraints on propositional state and metric resource variables) can be passed from a parent task or a child task or can be computed by a function of other variables.

2.3.2 Hierarchical Planning

Hierarchical Task Network (HTN) planners such as NOAH [Sacerdoti, 1977], NONLIN [Tate, 1977], and UMCP [Erol *et al.*, 1994b], use task hierarchies to search through all combinations of alternatives to achieve particular goals within a particular context. Rather than building a plan from the beginning forward (or end backward), hierarchical planners identify promising classes of long-term activities (abstract plans), and incrementally refine these to eventually converge on specific actions. However, planners of this class commit to one such combination of alternative actions in a found solution, resulting in a linearized action sequence that does not preserve any alternatives for handling uncertainty during execution. This thesis introduces techniques that exploit hierarchy for more efficient planning while preserving alternative choices for achieving subtasks.

While HTN planning is fairly well understood [Erol *et al.*, 1994a], using HTN planning for concurrently-executing agents is less well understood. If several HTN planning agents are each generating their own plans, how and when should these plans be merged? Certainly, merging could wait until the plans were fully refined, and merging techniques mentioned before [Georgeff, 1983] would work. But interleaving planning and merging holds greater promise for identifying and resolving key conflicts as early in the process as possible to try to avoid backtracking or failure. Such interleaving requires the ability to identify potential conflicts among abstract plans.

Another seeming advantage of hierarchical planners is their ability to reduce the search space through the encoding of domain knowledge in the structure of the task decompositions. By structuring the refinement of tasks and goals, the planner indirectly prunes the space of inconsistent or poor plans by avoiding sequences of operators that do not effectively achieve higher-level goals. While the planner is restricted to produce only those plans dictated by the structure of the hierarchy, the domain expert may still (if desired) guarantee completeness by carefully structuring the abstract plan operators. The domain expert has a similar responsibility when crafting operators for non-hierarchical planners. In addition, under certain restrictions, different forms of hierarchical problem solving can reduce the size of the search space by an exponential factor [Korf, 1987; Knoblock, 1991]. I demonstrate that a hierarchical planner can achieve similar speedups without these restrictions by reasoning about the summary conditions of abstract operators. Another analysis of hierarchical planning [Yang, 1997] explains that, in the case of interacting subgoals, certain structurings of the hierarchy that minimize these interactions can reduce worst case planning complexity exponentially. Yang's analysis applies to the coordination and planning algorithms in this thesis. However, the complexity analyses in Sections 6.3, 8.2.2, and 9.3 explain how using summary information can achieve exponential performance gains in addition to Yang's analysis by limiting the decomposition of task hierarchies and compressing the information manipulated by a coordinator, planner, or scheduler.

Tsuneto *et. al.* also discovered the value of identifying external conditions for HTN planning and showed that these conditions can help direct the search to avoid backtracking to improve planning efficiency for problems with interacting goals [Tsuneto *et al.*, 1998]. Tsuneto's ExCon heuristic identifies *must* external conditions and is used for ordering the expansion of tasks but not the ordering of decomposition choices to explore as discussed in Section 7.

Another class of hierarchical planners based on ABSTRIPS [Sacerdoti, 1974] introduces conditions at different levels of abstraction so that more critical conflicts are handled at higher levels of abstraction and less important (or easier) conflicts are resolved later at lower levels. While this approach similarly resolves conflicts at abstract levels, the planning decisions may not be consistent with conditions at lower levels resulting in backtracking. Summary information provides a means to make sound and complete decisions at abstract levels without the need to decompose and check consistency with lower levels. However, resolving conflicts based on criticality can still improve performance. This dissertation does not investigate this approach.

Yang presented a method (similar to summarization) for preprocessing a plan hierarchy in order to be able to detect unresolvable conflicts at an abstract level so that the planner could backtrack from inconsistent search spaces [Yang, 1990]. This corresponds to the use of $\neg MightSomeWay$ in Section 6.2. However, his approach requires that the decomposition hierarchy be modeled so that each abstract operator have *a unique main subaction* that has the same preconditions and effects as the parent. I avoid this restriction by analyzing the subplans' conditions and ordering constraints to automatically compute the parent's conditions.

The DPOCL (Decompositional Partial-Order Causal-Link) planner [Young *et al.*, 1994] adds action decomposition to SNLP [McAllester and Rosenblitt, 1991]. Like other HTN planners, preconditions and high level effects can be added to abstract tasks in order to help the planner resolve conflicts during decomposition. In addition, causal links can be specified in decomposition schemas to isolate external preconditions that DPOCL must satisfy. However, because these conditions and causal links do not necessarily capture all of the external conditions of abstract tasks, the planner does not find solutions at abstract levels and requires that all tasks be completely decomposed. In addition, DPOCL cannot determine that an abstract plan has unresolvable conflicts ($\neg MightSomeWay$) because there may be effects hidden in the decompositions of yet undetailed tasks that could achieve open preconditions. By deriving summary conditions automatically and using sound and complete algorithms for determining causal link information (e.g. must-achieve), the planner introduced in this thesis can find and reject abstract plans during search without adding burden to the domain expert to specify redundant conditions or causal links for abstract tasks. Furthermore, DPOCL does not handle concurrent activity.

Allen's temporal planner [Allen *et al.*, 1991] uses hierarchical representations of tasks and could be applied to reasoning about the concurrent actions of multiple agents. However, it does not exploit hierarchy by reasoning about abstraction levels separately and generates a plan by proving the consistency of the collective constraints. Allen's model of temporal plans [Allen and Koomen, 1983] and subsequent work on interval point algebra [Vilain and Kautz, 1986] strongly influenced my hierarchical task representation and algorithms that reason about them.

2.4 Summary of Related Work

In summary, this dissertation builds on work in multiagent coordination, planning, and plan merging to efficiently coordinate planning agents. The main extension is the use of summary information to reason about concurrent plans at multiple levels of abstraction. Previous analyses explain how hierarchical problem solving can reduce the search space exponentially. The use of summary information achieves these exponential gains without the harsh assumptions on the problem structure that these prior analyses make. Work in multiagent planning and plan merging does not exploit hierarchy and existing planning and scheduling algorithms to the extent of this thesis, but that work offers directions to extend my approach to include more sophisticated mental models of agents and explore efficient coordination protocols. In bridging multiagent and planning strategies, my approach extends representations of HTN planners while providing principled methods for reasoning efficiently about concurrently executing plans at abstract levels.

CHAPTER 3

A Model of Hierarchical Plans and their Concurrent Execution

The original purpose of developing the following theory was to provide, as simply as possible, a consistent model of execution to generally reason about the concurrent interaction of hierarchical plans. However, if the model shared important aspects of plans used by PRSs, HTNs, Allen's temporal plans, and many STRIPS-style plan representations, it could more easily draw on the strengths of many current planning and plan execution systems. Therefore, this theory of action tries to distill appropriate aspects of other theories, including Allen's temporal plans [Allen and Koomen, 1983], Georgeff's theory for multiagent plans [Georgeff, 1984], and Fagin *et. al.*'s theory for multiagent reasoning about knowledge [Fagin *et al.*, 1995].

First I give an example explaining why this formalization is necessary. Because this section details many definitions upon which the theory rests, the casual reader may skip to a summary of the concurrent hierarchical plan and metric resource usage formalization in Section 3.8. I also describe the model of metric resource usage in Section 3.7 separate from the model of concurrent hierarchical plans that is based those of existing planners [Chien *et al.*, 2000b; Laborie and Ghallab, 1995]. Resource usage constraints are described separately from those of state constraints and effects for simplicity since there are no interactions between state and resource constraints.

3.1 Example to Motivate Formalization

The example here (based on the one in Section 1.2) illustrates the need to formalize plan execution, summary information, and the algorithms that use summary information. (Terminology described in this section is introduced in the example in Section 1.3.2.) Consider a situation where the facilities manager plans to perform maintenance on the two machines and then transport his tool to storage bin3 using transport1, and the inventory manager wants to move part F from the dock to bin3 using transport2. Assume that the agents' plan libraries do not include subplans for moving their parts out of bin3; so, in effect, once a part is in bin3, it will stay there. Thus, it is not possible to coordinate their plans.

But, how would they recognize this based on the summary information for their top-level plans? They would find that if the *maintenance* top-level plan is ordered before $move\_F\_to\_bin3$, *MightSomeWay* is false since the external postconditions of *maintenance* must always clobber the external preconditions of $move\_F\_to\_bin3$. But how would the agents figure out that they could not overlap their plans? If one wanted to describe a method for determining whether two actions can or might overlap, it is not obvious how this should be done. One could conclude that because *must* postconditions conflict (($at(tool, bin3)$ and $\neg at(F, bin3)$) conflicts with ($at(F, bin3)$ and $\neg at(tool, bin3)$)), the algorithm for determining when tasks conflict should account for this and determine that *MightSomeWay* is false for all ordering constraints. But, this algorithm would be unsound in general because the $\neg at(tool, bin3)$ postcondition of the inventory manager's plan could be interpreted as pushing the tool out of the goal cell. In other words, if the facilities manager's plan is just to *visit* the goal location, then the facilities manager would still have accomplished its goal even if the inventory manager pushed the tool out of bin3.

It turns out that *MightSomeWay* is true for overlapping the plans because there are other plans with the same summary conditions that can be overlapped successfully. There is not enough information at the highest level to prove that *MightSomeWay* is false for this case. The real reason these particular plans cannot be overlapped is that the preconditions of the primitive plan moving F into bin3 are violated by the postconditions of the facilities manager's plan. The agents could detect this by digging deeper into the maintenance plan and considering the conditions of the primitive actions.

The difficulty of composing such algorithms for determining task interactions stems from an imprecise specification of concurrent plan execution and the large space of potential plans that have the same summary information. If the *MightSomeWay* algorithm is not specified to be complete, the agent may not determine that the *overlaps* relation cannot hold until it has exhaustively checked all synchronizations of the agents' primitive subplans. As the number of subplans grows, this becomes an intractable procedure [Vilain and Kautz, 1986]. Even worse would be if, for the sake of trying to be complete, an algorithm is specified such that it is unsound (as just discussed) leading to a synchronization choice that causes failure. Thus, it is necessary to formalize concurrent hierarchical plans, their execution, and the derivation of summary conditions to avoid costly, irreversible, as well as inconsistent decisions made during planning, plan execution, and coordination.

3.2 CHiPs

A concurrent hierarchical plan p is a tuple $\langle pre, in, post, type, subplans, order\rangle$. $pre(p)$, $in(p)$, and $post(p)$ are sets of literals (v or $\neg v$ for some propositional variable v) representing the preconditions, inconditions, and postconditions defined for plan p.[1] In order to reason about plan hierarchies as and/or trees of actions, the *type* of a plan p, $type(p)$, is given a value of either *primitive*, *and*, or *or*. An *and* plan is a non-primitive plan that is accomplished by carrying out all of its subplans. An *or* plan is a non-primitive plan that is accomplished by carrying out one of its subplans. So, *subplans* is a set of plans, and a *primitive* plan's *subplans* is the empty set. $order(p)$ is only defined for an *and* plan p and is a set of temporal relations [Allen, 1983] over pairs of subplans that together are consistent; for example, $before(p_i, p_j)$ and $before(p_j, p_i)$ could not both be in *order*. Plans left unordered with respect to each other are interpreted to potentially execute concurrently.

For the example in Figure 1.1, the production manager's highest level plan *produce_H*

[1]Functions such as $pre(p)$ are used for referential convenience throughout this paper. Here, *pre* and $pre(p)$ are the same, and $pre(p)$ is read as "the preconditions of p."

is the tuple

$$\langle\{\},\{\},\{\},and,\{produce\_G, produce\_H\_from\_G\},\{before(0,1)^2\}\rangle.$$

There are no conditions defined because *produce_H* can rely on the conditions defined for the primitive plans in its refinement. The plan for moving part A from bin1 to the first input tray of M1 using transport1 is the tuple

$$\langle\{\},\{\},\{\},and,\{start\_move, finish\_move\},\{meets(0,1)\}\rangle.$$

This plan decomposes into two half moves which help capture important intermediate effects. The parent orders its children with the *meets* relation to bind them together into a single move. The *start_move* plan is

$$\begin{aligned}&\langle\{at(A,bin1),available(A),free(transport1),\neg full(M1\_tray1)\},\\&\{\neg available(A),\neg at(A,bin1),\neg full(bin1),\neg full(M1\_tray1),free(transport1)\},\\&\{\neg at(A,bin1),\neg full(bin1),\neg available(A),\neg full(M1\_tray1),\neg free(transport1)\},\\&primitive,\{\},\{\}\rangle.\end{aligned}$$

The *finish_move* plan is

$$\begin{aligned}&\langle\{\},\{\neg available(A),\neg at(A,bin1),\neg full(bin1),full(M1\_tray1),\neg free(transport1)\},\\&\{at(A,M1\_tray1),available(A),full(M1\_tray1),\neg at(A,bin1),\neg full(bin1),\\&free(transport1)\},primitive,\{\},\{\}\rangle.\end{aligned}$$

I split the move plan into these two parts in order to ensure that no other action that executes concurrently with this one can use transport1, part A, or the input tray to M1. It would be incorrect to instead specify $\neg free$(transport1) as an incondition to a single plan because another agent could, for instance, use transport1 at the same time because its $\neg free$(transport1) incondition would agree with the $\neg free$(transport1) incondition of this move action. By representing the transition from $free$ to $\neg free$ and explicitly never decomposing the parent move plan, the modeler ensures that another action that tries to

[2] 0 and 1 are indices of the subplans in the decomposition referring to *produce_G* and *produce_H_from_G* respectively.

use transport1 concurrently with this one will cause a conflict.[3]

A postcondition is required for each incondition to specify whether the incondition changes. This clarifies the semantics of inconditions as conditions that hold only *during* plan execution whether because they are *caused* by the action or because they are *necessary conditions* for successful execution. If a plan's postconditions did not specify the truth values of the inconditions' variables at the end of execution, then it is not intuitive how those values should be determined in the presence of other concurrently executing plans. Requiring postconditions to specify such values resolves all ambiguity and simplifies rules specifying how the state transitions (described in Section 3.4).

The decomposition of a CHiP is in the same style as that of an HTN as described in [Erol *et al.*, 1994a]. An *and* plan is a task network, and an *or* plan is an extra construct representing a set of all tasks that accomplish the same goal or compound task. Tasks in a network are subplans of the plan corresponding to the network. High-level effects [Erol *et al.*, 1994a] are simply the postconditions of a non-primitive CHiP. CHiPs can also represent a variety of interesting procedures that can be executed by robust systems such as PRS [Georgeff and Lansky, 1986].

Note that the conditions of a CHiP are not summary conditions (formally introduced in Chapter 4). There is no modal information about the existence (*must*, *may*) or timing ($first$, *last*, *sometimes*, *always*) of these conditions. As will become clear in the following sections describing the semantics of CHiP execution, inherently preconditions are $must$, $first$; inconditions are *must*, *always*; and postconditions are $must$, $last$. Conditions in CHiPs were specified differently than summary conditions in order to ground the semantics of primitive actions and to keep greater compatibility with STRIPS-based operator specifications, which many planning systems use.

[3]Using universal quantification [Weld, 1994] a single plan could have a $\forall agent, agent \neq$ production-Manager $\rightarrow \neg$*using*(transport1, *agent*) condition that would exclude concurrent access to the transport. Other planning systems have explicit mechanisms for mutually exclusive (or atomic) resources [Rabideu *et al.*, 1999]. Thus, I could have simply specified transport1 as a metric resource with maximum capacity of one.

3.3 Executions

Informally, an *execution* of a CHiP is recursively defined as an instance of a decomposition and an ordering of its subplans' executions. Intuitively, when an agent executes a plan, it fixes the plan's start and finish times, chooses how it is refined, and determines at what points in time its conditions must hold. This formalism helps us reason about the outcomes of different ways to execute a group of plans, describe state transitions, and formalize other terms.

An *execution* of CHiP p, e, is a tuple $\langle d, t_s, t_f \rangle$. $t_s(e)$ and $t_f(e)$ are positive, non-zero real numbers representing the start and finish times of execution e, and $t_s < t_f$. $d(e)$ is a set of subplan executions representing the decomposition of plan p under this execution e. Specifically, if p is an *and* plan, then it contains one execution from each of the subplans; if it is an *or* plan, then it contains only one execution of one of the subplans; and it is empty if it is *primitive*. In addition, for all subplan executions, $e' \in d$, $t_s(e')$ and $t_f(e')$ must be consistent with the relations specified in $order(p)$. Also, the first subplan(s) to start must start at the same time as p, $t_s(e') = t_s(e)$; and the last subplan(s) to finish must finish at the same time as the p, $t_f(e') = t_f(e)$. The possible executions of a plan p is the set $\mathcal{E}(p)$ that includes all possible instantiations of an execution of p, meaning all possible values of the tuple $\langle d, t_s, t_f, constraints \rangle$, obeying the rules just stated.

For the example in Section 1.2, an execution for the production manager's top-level plan *produce_H* would be some $e \in \mathcal{E}(produce\_G\&H)$. e might be $\langle \{e_1, e_2\}$, 2.0, 9.0 $\rangle$ where $e_1 \in \mathcal{E}(produce\_G)$, and $e_2 \in \mathcal{E}(produce\_H)$. This means that the execution of *produce_H* begins at time 2.0 and ends at time 9.0. e_1 also starts at 2.0, and e_2 ends at 9.0. Figure 3.1 shows how these executions and those of their subplans are placed on a timeline.

For convenience, the *subexecutions* of an execution e, often later referred to as $subex(e)$, is defined recursively as the set of subplan executions in e's decomposition unioned with their subexecutions. For the production manager, $subex(e) = \{e_1, e_2\} \cup subex(e_1) \cup subex(e_2)$. In addition, we say that a condition of a plan with an execution in the set containing e and e's subexecutions is a *condition of e*. So, if the production manager executes its top-level plan, since $\neg at$(A, bin1) is an incondition of a primitive for moving part A from bin1 to machine M1 (the *start_move* plan described in Section

3.2), $\neg at$(A, bin1) is an incondition of the primitive's execution as well as the execution of *produce_H*.

3.4 Histories and Runs

Hypothetical possible worlds, called *histories*, are defined so that it can be determined what happens in all worlds, some, or none. Agents that must consider the ramifications of their abstract tasks before they decompose them must reason about possible histories to make sound decisions. Then I define *runs* as specifications of how the state transforms according to a particular history.[4] A state of the world, s, is a truth assignment to a set of propositions, each representing an aspect of the environment. I will refer to the state as the set of true propositional variables.

A *history*, h, is a tuple $\langle E, s_I \rangle$. E is a set of plan executions including those of all plans and subplans executed by all agents, and s_I is the initial state of the world before any plan begins executing. So, a history h is a hypothetical world that begins with s_I as the initial state and where only executions in $E(h)$ occur. So a history where only the production manager executes its plan might have an initial state as shown in Figure 1.1 where all parts and machines are available, and both transports are free. The set of executions E would contain e (the execution of *produce_H*), e_1 and e_2 (the executions of *produce_G* and *produce_H_from_G*), and their subexecutions, $subex(e_1)$ and $subex(e_2)$. Each execution specifies a start time, finish time, and refinement choices for the corresponding plan. Figure 3.1 shows the executions of a history h where the production manager executes *produce_H* concurrently with the facilities manager's *service_M1* plan. Here the production and facilities managers' plan executions conflict over the availability of machine M1 in this example history and corresponding run. Executions are labeled by their associated plans' names. The run $r(h)$ specifies the truth values of the state predicates over time. All of the line segments above the timeline represent executions in $E(h)$. The leaf plans in the figure each have two primitive subplans for beginning and finishing the low-level actions as signified by the intermediate points in the execution intervals. These hidden subplans function similarly to the *start_move* and *finish_move* primitives described in Section 3.2

[4]The formalization of histories and runs follows closely to that of Fagin *et. al.* in describing multiagent execution [Fagin *et al.*, 1995].

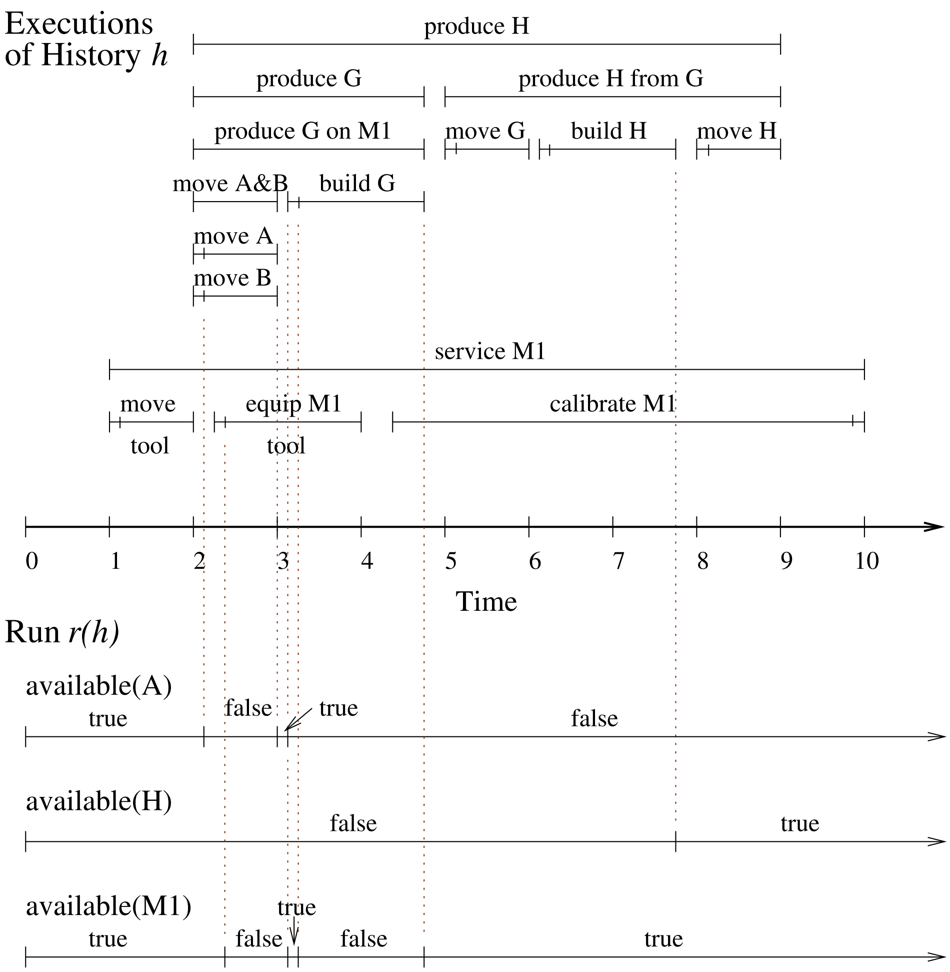


Figure 3.1: The production and facilities managers' conflicting plan executions

to conflict with other actions that access resources required by the plans throughout the duration of their executions.

A *run*, r, is a function mapping time to states. It gives a complete description of how the state of the world evolves over time, which ranges over the positive real numbers.[5] $r(t)$ denotes the state of the world at time t in run r. So, a condition c is *met* at time t if and only if the condition is a non-negated propositional variable v, and $v \in r(t)$ or if the condition is a negated propositional variable $\neg v$, and $v \notin r(t)$.

For each history h there is exactly one run, $r(h)$[6], that specifies the state transitions *caused* by the plan executions in $E(h)$. The interpretation of a history by its run is defined as follows. The world is in the initial state at time zero: $r(h)(0) = s_I(h)$. In the smallest interval after any point where one or more executions start and before any other start or end of an execution, the state is updated by adding all non-negated inconditions of the plans and then removing all negated inconditions (as specified in Axiom 1 below). Similarly, at the point where one or more executions finish, the state is updated by adding all non-negated postconditions of the plans and then removing all negated postconditions (as specified in Axiom 2). Lastly, if no execution of a plan begins or ends between two points in time, then the state must be the same at those points (Axiom 3).

Axiom 1

$$
\begin{aligned}
&\forall h, e_1, t, e_1 \in E(h) \wedge \\
&\quad [t > t_s(e_1) \wedge (\forall e_2 \in E(h), (t_s(e_2) > t_s(e_1) \rightarrow \\
&\quad t \leq t_s(e_2)) \wedge (t_f(e_2) > t_s(e_1) \rightarrow t < t_f(e_2)))] \rightarrow \\
&\quad r(h)(t) = r(h)(t_s(e_1)) \cup \{v \mid \exists p, e_3 \in E(h), \\
&\qquad e_3 \in \mathcal{E}(p) \wedge t_s(e_3) = t_s(e_1) \wedge v \in in(p)\} - \\
&\qquad \{v \mid \exists p, e_3 \in E(h), e_3 \in \mathcal{E}(p) \wedge \\
&\qquad t_s(e_3) = t_s(e_1) \wedge \neg v \in in(p)\}.
\end{aligned}
$$

[5]Real valued time points are used instead of discrete time to avoid complications in formalizing actions with durations of an atomic unit of time.

[6]For convenience, I now treat r as a function mapping histories to runs, so $r(h)(t)$ is a mapping of a history and a time to a state.

Axiom 2

$$\begin{aligned}
&\forall h, e_1, t, e_1 \in E(h) \wedge \\
&\quad [t < t_f(e_1) \wedge (\forall e_2 \in E(h), (t_s(e_2) < t_f(e_1) \rightarrow \\
&\quad t > t_s(e_2)) \wedge (t_f(e_2) < t_f(e_1) \rightarrow t \geq t_f(e_2)))] \rightarrow \\
&\quad r(h)(t_f(e_1)) = r(h)(t) \cup \{v \mid \exists p, e_3 \in E(h), \\
&\qquad e_3 \in \mathcal{E}(p) \wedge t_f(e_3) = t_f(e_1) \wedge v \in post(p)\} - \\
&\qquad \{v \mid \exists p, e_3 \in E(h), e_3 \in \mathcal{E}(p) \wedge \\
&\qquad t_f(e_3) = t_f(e_1) \wedge \neg v \in post(p)\}.
\end{aligned}$$

Axiom 3

$$\begin{aligned}
&\forall h, t_1, t_2, t_1 < t_2 \wedge (\neg \exists e \in E(h), t_1 \leq t_s(e) < t_2 \\
&\quad \vee t_1 < t_f(e) \leq t_2) \rightarrow r(h)(t_2) = r(h)(t_1).
\end{aligned}$$

The run of a history allows both inconditions and postconditions to change the state of the world. An agent should expect that if a plan's preconditions are met, the inconditions would not be clobbered in the absence of other interacting plan executions. Otherwise, the inconditions should be preconditions.

So, for the history given in Figure 3.1, the state at time 0.0, $r(h)(0.0)$, will be the initial state where all parts and machines are available, and the transports are free for use. Since *service_M*1 executes first beginning at 1.0, the run of this history specifies that the initial state (including *available*(A) and *available*(M1) as shown in the figure) remains the same up to time 1.0 since no action takes place before then (i.e. $r(h)(0.0) = r(h)(0.5) = r(h)(1.0) = s_I(h)$). During the production of part G on machine M1, part A is transferred to machine M1. As a result of the incondition $\neg$*available*(A) in the *finish_move* subexecution of *move_A*, *available*(A) becomes *false* at time 2.1 (*available*(A) $\notin r(h)(2.1)$). In the brief interval after part A is moved, A is available just before being consumed by *build_G*. Part H does not become available until the postcondition of the execution of *build_H* asserts *available*(H). Machine M1 becomes unavailable when it is equipped with the tool, but because the subexecutions of *build_G* assert both *available*(M1) and $\neg$*available*(M1), there is a short interval starting at 3.1 where M1 is available, violating the *equip_M1_tool* plan's incondition that M1 is unavailable. In addition, the *calibrate_M*1 plan expects M1 to be unavailable, but *build_G* asserts that M1 is available when it is finished with the machine at 4.8, thus, violating the incondition of the $calibrate\_M1$ plan.

Now that I have specified state transitions, I can define what it means for a plan to execute successfully. An execution $e = \langle d, t_s, t_f \rangle$ *succeeds in h* if and only if the executing plan's preconditions are met at t_s; the inconditions are met throughout the interval (t_s, t_f); the postconditions are met at t_f; and all executions in e's decomposition are in $E(h)$ and succeed. Otherwise, *e fails*. So, in a history h where the production manager successfully executes its plan to produce G and H, $E(h) = \{e\} \cup subex(e)$, and all conditions of all plans with executions in $E(h)$ are met at the appropriate times. In the history of Figure 3.1, inconditions of the facilities manager's $service\_M1$ execution fails because conditions of $equip\_M1\_tool$ and $calibrate\_M1$ are not met causing their executions to fail. Because they are subexecutions of $service\_M1$, $service\_M1$ also fails. $produce\_H$ also fails because *available*(M1) is $false$ at 3.1 when it is required as a precondition to $build\_G$.

3.5 Asserting, Clobbering, Achieving, and Undoing

Conventional planning literature often speaks of *clobbering* and *achieving* preconditions of plans [Weld, 1994]. In CHiPs, these notions are slightly different since inconditions can clobber and be clobbered, as seen in the previous section. Formalizing these concepts and another, *undoing* postconditions, helps prove properties of summary conditions. However, it will be convenient to define first what it means to *assert* a condition. Figure 3.2 gives examples of executions involved in these interactions, and I define these terms as follows:

> An execution e of plan p *asserts* a condition ℓ at time t in a history h if and only if ℓ is an incondition of p, t is in the smallest interval beginning after $t_s(e)$ and ending before a following start or finish time of any execution in $E(h)$, and ℓ is satisfied by $r(h)(t)$; or ℓ is a postcondition of p, $t = t_f(e)$, and ℓ is satisfied by $r(t)$.

Asserting a condition only *causes* it to hold if the condition was not previously met. Otherwise, the condition was already satisfied and the action requiring it did not really *cause* it. In Figure 3.1, wherever the state changed in the run of the history, an execution *caused* the state to change by asserting a condition in the state that previously did not hold.

> A *precondition* ℓ of plan p_1 is [*clobbered*, *achieved*][7] in e_1 (an execution of p_1) by e_2 (an execution of plan p_2) at time t if and only if e_2 asserts [ℓ', ℓ] at t; $\ell \Leftrightarrow \neg\ell'$; and e_2 is the last execution to assert ℓ or ℓ' before or at $t_s(e_1)$. An [*incondition*, *postcondition*] ℓ of plan p_1 is *clobbered* in e_1 by e_2 at time t if and only if e_2 asserts ℓ' at t; $\ell \Leftrightarrow \neg\ell'$; and [$t_s(e_1) < t < t_f(e_1), t = t_f(e_1)$].

Achieving inconditions and postconditions does not make sense for this formalism, so it is not defined. Achieving inconditions does not make sense because they are not required external to the execution—that is the purpose of preconditions. Inconditions are asserted internally at the time that they are first required to be met so that they, in effect, achieve themselves. Postconditions are also asserted at the time they are required.

Figure 3.2c shows the five different ways an execution clobbers a condition of another. For the two on the left, the postcondition of e clobbers the precondition of e', and the incondition of e clobbers the precondition of e'. In the middle, the postcondition of e clobbers an incondition of e'; the incondition of e clobbers the incondition of e'. On the right, the postcondition of e clobbers the postcondition of e'. Figure 3.1 shows *build_G* execution clobbering the incondition of *equip_M1_tool* that machine M1 is available.

> A *postcondition* ℓ of plan p_1 is *undone* in e_1 (an execution of p_1) by e_2 (an execution of plan p_2) at time t if and only if e_2 asserts ℓ' at t; $\ell \Leftrightarrow \neg\ell'$; and e_2 is the first execution to assert ℓ or ℓ' at or after $t_f(e_1)$.

3.6 External Conditions

As recognized in [Tsuneto *et al.*, 1998], external conditions are important for reasoning about potential refinements of abstract plans. Although the basic idea is the same, I define them a little differently and call them *external preconditions* to differentiate them from other conditions that I call *external postconditions*. Intuitively, an external precondition of a group of partially ordered plans is a precondition of one of the plans that is

[7] I use braces [ ] as a shorthand when defining similar terms and procedures. For example, saying "[a, b] implies [c, d]" means a implies c, and b implies d.

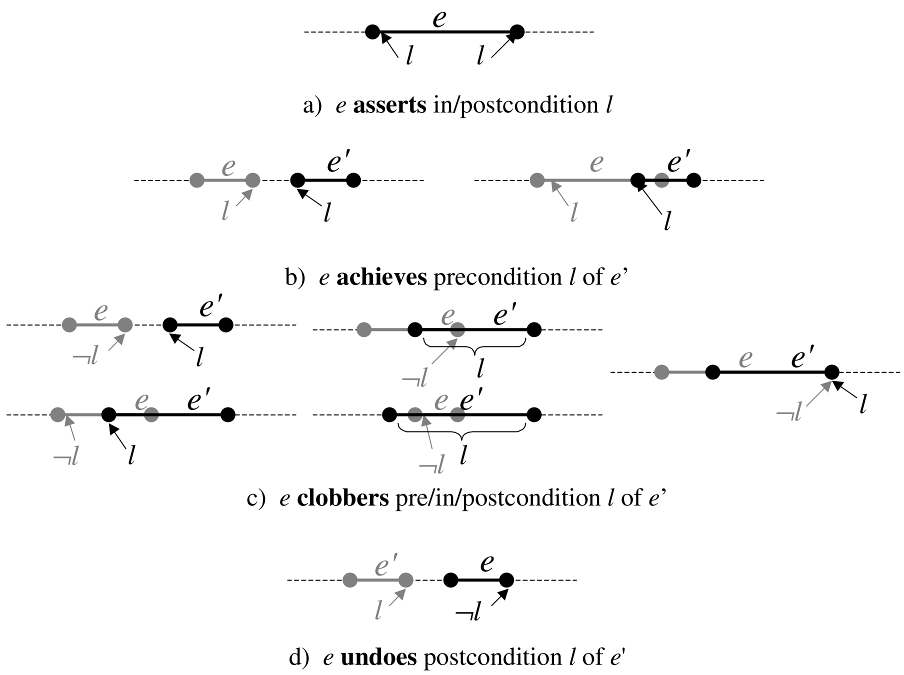


Figure 3.2: Interval interactions of plan steps

not achieved by another in the group and must be met external to the group. External postconditions, similarly, are those that are not undone by plans in the group and are net effects of the group.

Formally, an *external precondition* ℓ of an *interval* (t_1, t_2) in history h is a precondition of a plan p with some execution $e \in E(h)$ for which $t_1 \leq t_s(e) < t_2$, and ℓ is neither achieved nor clobbered by an execution at a time t where $t_1 \leq t \leq t_s(e)$. An *external precondition* of an *execution* $e = \langle d, t_s, t_f \rangle$ is an external precondition of an interval (t_1, t_2) in some history where $t_1 \leq t_s$; $t_f \leq t_2$; and there are no other plan executions other than the subexecutions of e. So, if the history in Figure 3.1 only included the execution of *produce_H*, *available*(M1) would be an external precondition of any interval beginning by the start of *build_G* and ending after that point because no execution other than *build_G* achieves or clobbers *available*(M1) before the start of *build_G*. *available*(M1) is thus an external precondition of the execution *produce_H* because in this history *available*(M1) is an external precondition of the interval of the execution *produce_H*. An *external precondition* of a *plan p* is an external precondition of any of p's executions. It is called a *must* precondition if it is an external precondition of all executions; otherwise it is called a *may* precondition. So, because *available*(M1) is an external precondition of the execution *produce_H* for the execution in Figure 3.1, it is an external precondition of the plan *produce_H*. It is a *may* external precondition because it is not an external precondition of the execution of *produce_H* where *produce_g_on_M*2 is selected in the decomposition of *produce_G*.

Similarly, an *external postcondition* ℓ of an *interval* (t_1, t_2) in h is a postcondition of a plan p with some execution $e \in E(h)$ for which $t_1 \leq t_f(e) \leq t_2$; ℓ is asserted by e; and ℓ is not undone by any execution at a time t where $t_f(e) < t \leq t_2$. External postconditions of executions and plans can be defined in the same way as external preconditions. $\neg$*available*(A) is a *must* external postcondition of *produce_H* because it is definitely consumed in the production of G, and no other execution makes A available again.

3.7 Resource Usage

Metric resources, such as a battery and a collection of available part-assembling machines, typically have maximum capacities. Battery energy is limited by the capacity of the battery, and there are only so many part-assembling machines that can be available. At the same time, these resource variables can have minimum values. The number of available machines cannot fall below zero, and a modeler may constrain the battery to never fall below 20% of its capacity.

The use of these resources have different properties. Battery energy, once used, cannot be restored without explicit recharging. This is an example of a *depletable* resource—when a task depletes the resource, the resource level is not restored. The machines, however, are only unavailable while they are being used. Once a task is finished using a machine, it automatically becomes available again. Resources of this type are *non-depletable*—the resource level is restored at the end of the task.

Thus, my model of a metric resource is the tuple $\langle$*name*, *minvalue*, *maxcapacity*, *type*$\rangle$. *name* is an identifier for the resource. The min and max values can be integer or real valued. The *type* of the resource is either depletable or non-depletable. An instance of metric resource usage is simply $\langle$*name*, *usage*$\rangle$ where *usage* is the value that depletes the resource *name*.

In order to understand how resource conflicts can be recognized by a planner, a model of execution is needed just as is given for conditions in Section 3.3. The state of a resource is a level value (again, integer or real). For depletable resource usage, a task that depletes a resource is modeled to instantaneously deplete the resource (subtract *usage* from the current state) at the start of the task by the full amount. For non-depletable resource usage, a task also depletes the *usage* amount at the start of the task, but *usage* is restored (added back to the resource state) at the end of execution. A task can restore a resource by having a negative *usage*.

To extend this to concurrent hierarchical execution, any task (abstract or primitive) can have usages over any resource.[8] If a set of tasks using the same resource begin at the same time, the total usage is summed and subtracted from the resource state. Likewise, for a set of tasks using a non-depletable resource and all ending at the same, the usages are summed and added back to the resource level. Conflicts occur when the level of the resource falls below the min value or exceeds the max capacity. This simple model of resources and usage is based directly on those of the ASPEN [Chien *et al.*, 2000b] and IxTeT [Laborie and Ghallab, 1995] planners. After I describe summary resource usage in Section 4.4, Section 5.2 explains how these planners recognize these conflicts.

3.8 Summary of Representations

In summary, in this section I have described the structure of a concurrent hierarchical plan (CHiP) in its specification of propositional state variable conditions and also a representation of metric resource usage. A CHiP has pre-, in-, and postconditions as well as a decomposition (unless it is primitive). Preconditions are requirements on the state at the point where plan execution begins. Postconditions are effects realized at the end of execution, and inconditions are intermediate requirements or effects that must hold throughout the duration of the plan's execution. If the CHiP is an *and* plan, all of the subplans must be executed according to a set of specified partial ordering constraints. If it is an *or* plan, just one subplan must be executed. An execution of a plan is an instance of a CHiP with a particular decomposition and with fixed start and finish times. A history is a possible world specifying an initial state and a set of executions taking place together. A run describes how the state is updated over time for a particular history with concur-

[8] Although it is trivial to extend the algorithms in this paper to handle resource constraints specified for abstract tasks, the algorithms will assume only primitive tasks can use resources.

rently executing CHiPs. A set of histories can enumerate different ways a set of plans can be decomposed and executed. This is helpful for describing potential plan interactions when the decomposition and ordering of actions is yet uncertain. Based on this model of execution, I defined different interactions among plan executions: asserting, achieving, clobbering, and undoing conditions (see Figure 3.2). I then defined which conditions of an abstract plan's subplans are the external pre- and postconditions of the abstract plan. An external precondition is a condition that is not established within the abstract plan's decomposition but is required to be established externally. An external postcondition is an effect of the abstract plan's decomposition that is not undone internally. Examples of external conditions are given in Section 1.2. Metric resources are categorized as either depletable or non-depletable, depending on whether the resource is restored at the end of its use, and have maximum capacities and minimum values.

Metric resource usage is represented (as is common) as a value that is subtracted from the current state of the resource at the beginning of the task. A resource's state at any point in time is the initial state minus the usages of tasks whose intervals overlap the time point and, for depletable resources, the usages of tasks completed before this time point. These formalisms enable us to define summary information and describe the algorithms that derive this information and use it within a planner or plan coordinator.

CHAPTER 4

Plan Summary Information

With the previous formalisms, I can now define summary information and describe a method for computing it for non-primitive plans (in Section 4.1). After pointing to soundness and completeness proofs in the appendices for the algorithm that derives summary conditions (Section 4.2), Section 4.3 also defines and gives sound, complete algorithms for determining a variety of interactions among concurrently executing plans based on their summary information. Then I describe a representation for summarized metric resource usage of a task and an algorithm that derives it in Section 4.4. In Section 4.5, I summarize the many terms and algorithms given in this chapter.

The *summary information* for a plan p is p_{sum}. p_{sum} is a tuple $\langle pre_{sum}, in_{sum}, post_{sum}\rangle$, whose members are sets of *summary conditions*. The *summary* [*pre*, *post*][1] *conditions* of p, $[pre_{sum}(p),\ post_{sum}(p)]$, contain the external [pre, post] conditions of p. The *summary inconditions* of p, $in_{sum}(p)$, contain all conditions that must hold within some execution of p for it to be successful. A condition c in one of these sets is a tuple $\langle \ell, existence, timing\rangle$. $\ell(c)$ is a literal. The *existence* of c can be *must* or *may*. If $existence(c) = must$, then c is called a *must* condition because ℓ holds for every successful plan execution (ℓ *must* hold). For convenience I usually write $must(c)$. c is a *may* condition ($may(c)$ is $true$) if there is at least one successful plan execution where $\ell(c)$ must hold.

The *timing* of a summary condition c can take the values $always$, $sometimes$, $first$, $last$. $timing(c)$ is $always$ for $c \in in_{sum}$ if $\ell(c)$ is an in-condition that must hold through-

[1] As mentioned earlier, I use braces [ ] as a shorthand when defining similar terms and procedures. For example, saying "[a, b] implies [c, d]" means a implies c, and b implies d.

out the execution of p (ℓ holds *always*); otherwise, $timing(c) = sometimes$ meaning $\ell(c)$ holds at one point, at least, within an execution of p. The $timing$ is $first$ for $c \in pre_{sum}$ if $\ell(c)$ holds at the beginning of an execution of p; otherwise, $timing = sometimes$. Similarly, $timing$ is $last$ for $c \in post_{sum}$ if $\ell(c)$ is asserted at the end of a successful execution of p; otherwise, it is $sometimes$. Although $existence$ and $timing$ syntactically only take one value, semantically $must(c) \Rightarrow may(c)$, and $always(c) \Rightarrow sometimes(c)$.

Examples of $must$, may, $always$, and $sometimes$ conditions are given for the manufacturing domain in Section 1.2. For the production manager's *produce_H* plan, $available$(A) and $available$(M1) are both summary preconditions, but only $available$(A) is $first$ because part A must be available to move to machine M1 at the start of the execution, and $available$(M1) is $sometimes$ because the production manager does not need the machine to build G and H until part way into the *produce_H* plan. Likewise, $available$(G) is a $sometimes$ summary postcondition because it is made available in the middle of the *produce_H* plan, and $available$(H) is a $last$ summary postcondition because it is an external postcondition of the last plan step of *produce_H* where H is moved to bin1.

4.1 Deriving Summary Conditions

The method for deriving the summary conditions of a plan p is recursive. First, summary information must be derived for each of p's subplans, and then the following procedure derives p's summary conditions from those of its subplans and its own sets of conditions. This procedure only applies to plans whose expansion is finite.

Summary conditions for primitives and non-primitives

- First, for each literal ℓ in $pre(p)$, $in(p)$, and $post(p)$, add a condition c with literal ℓ to the respective set of summary conditions for plan p. $existence(c)$ is $must$, and $timing(c)$ is $first$, $always$, or $last$ if ℓ is a pre-, in-, or postcondition respectively.

Summary [pre, post] conditions for *and* plan

- Add a condition c to the summary [pre, post] conditions of *and* plan p for each summary [pre, post] condition c' of p's subplans that is not [*must-achieved*, *must-undone*] or *must-*

clobbered[2] by another of p's subplans, setting $\ell(c) = \ell(c')$.[3]

- Set $existence(c) = must$ if $\ell(c)$ is a [pre, post] condition of p or is the literal of a *must* summary [pre, post] condition in a subplan of p that is not [*may-achieved*, *may-undone*] or *may-clobbered* by any other subplans. Otherwise, set $existence(c) = may$.
- Set $timing(c) = [first, last]$ if $\ell(c)$ is a [pre, post] condition of p or the literal of a [$first$, $last$] summary [pre, post] condition of a [least, greatest] temporally ordered subplan (i.e. no others are constrained by $order(p)$ to [begin before, end after] it).[2] Otherwise, set $timing(c) = sometimes$.

Summary [pre, post] conditions for *or* plan

- Add a condition c to the summary [pre, post] conditions of *or* plan p for each summary [pre, post] condition c' in p's subplans, setting $\ell(c) = \ell(c')$.
- Set $existence(c) = must$ if $\ell(c)$ is a [pre, post] condition of p or a *must* summary [pre, post] condition of all of p's subplans. Otherwise, set $existence(c) = may$.
- Set $timing(c) = [first, last]$ if $\ell(c)$ is a [pre, post] condition of p or the literal of a [$first$, $last$] summary [pre, post] condition in a subplan. Otherwise, set $timing(c) = sometimes$.

Summary inconditions for *and* plan

- Add a condition c to the summary inconditions of *and* plan p for each c' in C defined as the set of summary inconditions of p's subplans unioned with the set of summary preconditions of the subplans that are not always $first$ in a least temporally ordered subplan and with the set of summary postconditions of the subplans that are not always $last$ in a greatest temporally ordered subplan[2], and set $\ell(c) = \ell(c')$.
- Set $existence(c) = must$ if $\ell(c)$ is an incondition of p or a literal of a *must* summary condition $c' \in C$, as defined above, and is always not a $first$ or $last$ condition.[2] Otherwise, set $existence(c) = may$.
- Set $timing(c) = always$ if $\ell(c)$ is an incondition of p or a literal in the *always* summary inconditions in the subplans of p whose intervals must *cover* the interval within the execution of p.[2] Otherwise, set $timing(c) = sometimes$.

[2]See the definitions and algorithms in Section 4.3.4 about how to determine *must-achieved*, *may-achieved*, *must-undone*, *may-undone*, *must-clobbered*, and *may-clobbered* as well as those about determining least and greatest temporally ordered subplans, always $first/last$ conditions, and interval covering.

[3]To resolve ambiguity with set membership, any two summary conditions (c and c') are equal if $\ell(c) = \ell(c')$, and if they belong to the same set of summary conditions for some plan.

Summary inconditions for *or* plan

- Add a condition c to the summary inconditions of *or* plan p for for each summary incondition c' in p's subplans, setting $\ell(c) = \ell(c')$.
- Set $existence(c) = must$ if $\ell(c)$ is an incondition of p or a *must* summary incondition of all of p's subplans. Otherwise, set $existence(c) = may$.
- Set $timing(c) = always$ if $\ell(c)$ is an incondition of p or an *always* summary incondition of all of p's subplans. Otherwise, set $timing(c) = sometimes$.

Below is the summary information derived for the production manager's top-level plan (*produce_H*) for the example given in Section 1.2, the *produce_G* subplan, and a *move* plan. The summary conditions for the other factory managers' plans and the subplans of *produce_G* (to give an example of the summarization of an *or* plan) are given in Appendix A. In addition to the discussion following these example plans, Section 1.3.2 explains the derivation of some of the summary conditions. Following each literal is a tag for the *existence* and *timing* information. "Mu" is *must*; "Ma" is *may*; "F" is *first*; "L" is *last*; "S" is *sometimes*; and "A" is *always*. The language of the coordination code allows for variables (preceded by "$") in the conditions.

Variables are often used as parameters to a general plan operator schema or for introducing least commitment in constraints on objects. Below, $*srcG* is the location where part G is placed after its production, and $*imtransport* is the transport used by the inventory manager for some of its tasks. Yang and Chan show how variable binding can be used successfully in least commitment planners and point to many other planners that use them [Yang and Chan, 1994]. I do not give details in this thesis on the summarization of plans with unbound variables; however, the basic approach is to unify conditions with variables to see if the domains of the variables overlap for determining whether one condition may or must be the same as another. This unification test also enables a planner to determine whether one plan must or may achieve, clobber, or undo a condition of another.

Production manager's *produce_H* plan:

Summary preconditions:

`at(bin1, A)MuF, at(bin2, B)MuF, available(A)MuF, free(transp1)MaF,`
`¬full($srcG)MuF, available(B)MuS, ¬full(M1_tray2)MaS, available(M1)MaS,`
`free(transp2)MaF, ¬full(M2_tray2)MaS, available(M2)MaS, ¬full(M2_tray1)MaS`

Summary inconditions:

available(B)MuS, ¬full(M1_tray2)MaS, available(M1)MaS, ¬full(M2_tray2)MaS, available(M2)MuS, full($srcG)MuS, ¬at(bin1, A)MuS, ¬full(bin1)MuS, ¬at(bin2, B)MuS, ¬full(bin2)MuS, free(transp1)MaS, at($srcG, G)MaS, available(G)MuS, ¬available(A)MuS, ¬available(B)MuS, ¬at($srcG, A)MuS, ¬at(M1_tray2, B)MaS, free(transp2)MuS, ¬at(M2_tray2, B)MaS, at(bin2, B)MuS, at($srcG, A)MuS, available(A)MuS, at(M1_tray2, B)MaS, full(M1_tray2)MaS, ¬full($srcG)MuS, ¬free(transp1)MaS, ¬available(M1)MaS, at(M2_tray2, B)MaS, full(M2_tray2)MaS, ¬free(transp2)MuS, ¬available(M2)MuS, ¬full(M2_tray1)MuS, ¬at($srcG, G)MuS, ¬available(G)MuS, ¬at(M2_tray1, G)MuS, at(M2_tray1, G)MaS, full(M2_tray1)MuS, at(M2_tray1, H)MaS, available(H)MuS, ¬at(M2_tray1, H)MuS, ¬available(H)MuS

Summary postconditions:

¬at(bin1, A)MuS, ¬full(bin1)MaS, ¬at(bin2, B)MuS, ¬full(bin2)MaS, free(transp1)MaS, ¬available(A)MuS, ¬available(B)MuS, ¬at($srcG, A)MuS, ¬at(M1_tray2, B)MaS, ¬full(M1_tray2)MaS, available(M1)MaS, ¬at(M2_tray2, B)MaS, ¬full(M2_tray2)MaS, ¬at($srcG, G)MuS, ¬available(G)MuS, ¬at(M2_tray1, G)MuS, available(M2)MuS, at($srcG, H)MuL, available(H)MuL, full($srcG)MuL, ¬at(M2_tray1, H)MuL, ¬full(M2_tray1)MuL, free(transp2)MuL

Production manager's *produce_G* plan:

Summary preconditions:

at(bin1, A)MuF, at(bin2, B)MuF, available(A)MuF, free(transp1)MaF, ¬full($srcG)MuF, available(B)MuS, ¬full(M1_tray2)MaS, available(M1)MaS, free(transp2)MaF, ¬full(M2_tray2)MaS, available(M2)MaS

Summary inconditions:

at(bin2, B)MuS, available(B)MuS, ¬full(M1_tray2)MaS, at($srcG, A)MuS, available(A)MuS, full($srcG)MuS, ¬at(bin1, A)MuS, ¬full(bin1)MuS, at(M1_tray2, B)MaS, full(M1_tray2)MaS, ¬at(bin2, B)MuS, ¬full(bin2)MuS, free(transp1)MaS, ¬available(A)MuS, ¬full($srcG)MuS, ¬free(transp1)MaS, ¬available(B)MuS, available(M1)MaS, ¬available(M1)MaS, ¬full(M2_tray2)MaS, at(M2_tray2, B)MaS, full(M2_tray2)MaS, free(transp2)MaS, ¬free(transp2)MaS, available(M2)MaS, ¬available(M2)MaS

Summary postconditions:

`full($srcG)MaS, ¬at(bin1, A)MuS, ¬full(bin1)MuS, ¬at(bin2, B)MuS, ¬full(bin2)MuS, free(transp1)MaS, at($srcG, G)MuL, available(G)MuL, ¬available(A)MuL, ¬available(B)MuL, ¬at($srcG, A)MuL, ¬at(M1_tray2, B)MaL, ¬full(M1_tray2)MaL, available(M1)MaL, free(transp2)MaS, ¬at(M2_tray2, B)MaL, ¬full(M2_tray2)MaL, available(M2)MaL`

Production manager's *move_A_to_$srcG* plan:

Summary preconditions:

`at(bin1, A)MuF, available(A)MuF, free(transp2)MuF, ¬full($srcG)MuF`

Summary inconditions:

`¬at(bin1, A)MuA, ¬available(A)MuS, available(A)MuS, full($srcG)MuS, ¬full($srcG)MuS, ¬full(bin1)MuA, ¬free(transp2)MuS, free(transp2)MuS`

Summary postconditions:

`at($srcG, A)MuL, available(A)MuL, full($srcG)MuL, ¬at(bin1, A)MuL, ¬full(bin1)MuL, free(transp2)MuL`

Notice that only the *move_A_to_$srcG* plan has *always* inconditions. This is because its subplans have the *meets* ordering constraint (see the example in Section 3.2) such that summary inconditions in both of the subplans must span the entire duration of the parent. The other plans have *before* ordering constraints over their subplans, so even if all of the subplans have common inconditions, they may not span potential gaps between them (where plans of other agents or concurrent subgoals may be placed), so they do not *always* need to hold.

The *produce_G or* plan is achieved by either producing G on machine M1 or M2. The summary conditions that involve M1 and M2 are in only one of the subplans, so those conditions are summarized as *may* conditions in *produce_G*. However, the conditions involving parts A, B, and G are common in both subplans, so they are summarized as *must* conditions in the parent. The *produce_H* plan summary preconditions require A and B to be available externally to produce G. But, although G is needed to produce H as a summary precondition of the *produce_H_from_G* plan (shown in Appendix A), there are no preconditions that G must be available in the *produce_H* parent plan. This makes sense because G does not need to be made available externally since *produce_H_from_G*'s summary precondition for G is *must-achieved* by G's production in the *produce_G* subplan. Similarly, *produce_H* has no summary postcondition that G is available from its production since $available$(G) is *must-undone* by the *must* $\neg available$(G) summary postcondition of $produce\_H\_from\_G$ that consumes G.

Because these plans have no internal conflicts, there are no *must* or *may-clobbered* summary conditions. However if a condition is *must* or *may-clobbered*, this internal conflict may be resolved in the domain model by either adding temporal constraints or eliminating the *or* branch (subplan) containing the conflict. The definitions and algorithms for *must/may-achieved/clobbered/undone* are given in Section 4.3.4.

4.2 Proving the Properties of Summary Conditions

I prove that the procedure in the previous section derives summary conditions with their intended properties. This proof (given in Appendix C) is not broken down into separate lemmas for each property since the truth of each property depends on those of others, and it depends on soundness and completeness proofs of algorithms introduced in the next section. These results ease the proofs of soundness and completeness for procedures determining how CHiPs can definitely or potentially interact so that good planning and coordination decisions can be made at various levels within and among plan hierarchies.

4.3 Supporting Mechanisms

In this section, I describe algorithms for computing interval relations and determining plan interactions base on summary information. These plan interactions describe relationships where a plan must or may achieve, clobber, or undo the condition of another under particular ordering constraints. The algorithms for computing interval relations are used to determine the *timing* of summary conditions. The algorithms for determining plan interactions are used to determine which summary conditions are propagated as external conditions of the parent plan. They are also used to identify threats and solutions within planning and coordination algorithms (see Section 5.1). Readers who do not wish to fully understand proofs of these and other formalisms of Chapter 4 may safely skip to Section

4.5.

The procedure in Section 4.1 ensures that external and internal conditions are captured by summary conditions and *must*, *always*, *first*, and *last* have their intended meanings. The soundness and completeness proof of the algorithm depends on the soundness and completeness of the algorithms for determining the plan interactions (such as must-achieve and may-clobbers). However, these algorithms for determining the plan interactions rely on the correctness of the procedure for deriving summary information. To avoid a circular argument, I shall assume that the summary information is correct for the set of plans over which these relations are defined, and in the proof of the properties of summary information (Appendix C), I will be able to use the following formalisms since I also assume the properties of summary information for the set of subplans in the inductive step. These concepts will also aid in proving the soundness and completeness of methods for determining whether certain temporal relations can or might hold among abstract plans.

The definitions and algorithms throughout this section are given within the context of a set of plans P with a corresponding set of summary information P_{sum}, a set of ordering constraints *order*, and a set of histories H including all histories where $E(h)$ only includes an execution e of each plan in P and e's subexecutions, and $E(h)$ satisfies all constraints in *order*. They are all concerned with the ordering of plan execution intervals and the timing of conditions. By themselves, they do not have anything to do with whether conditions may need to be met or must be met for a plan execution.

4.3.1 Algorithms for computing interval relations

The algorithms for determining whether the defined relations hold between summary conditions for plans in P use a point algebra constraint table as described in [Vilain and Kautz, 1986]. The point algebra table is constructed for the interval endpoints corresponding to the executions of the plans in P; a row and column for both p^- (start endpoint) and p^+ (finish endpoint) are added for each plan $p \in P$. Then, the constraints in *order* are added to the table, and the transitive closure is computed to get all constraints entailed from those in *order*. This only needs to be done once for any P and *order*. For *must-assert* relations, each constraint specified in an entry of the case tables must be a subset of the constraints in the point algebra table in order for the relation to hold for that case. For *may-assert* relations, each constraint specified in an entry of the case tables must be missing from the constraints in the point algebra table in order for the relation to hold for that case. These inferences using the point algebra table are shown to be sound and complete in [Vilain and Kautz, 1986].

I determine that a plan q in p's subplans is temporally ordered [*least*, *greatest*] if and only if [q^-, q^+] is constrained [before, after] or equal to all other points in the point algebra table for p's subplans. This is done by looking at each entry in the row for [q^-, q^+] and checking to see that the constraint is [$<$, $>$], $=$, or [$\leq$, $\geq$]. If this is not the case, then q is not always [*first*, *last*]. A summary [pre, post] condition of plan q in p's subplans is *always* [*first*, *last*] if and only if q is a [least, greatest] temporally ordered subplan. A summary [pre, post] condition of plan q in p's subplans is *always not* [*first*, *last*] if and only if in the row for [q^-, q^+] there is an entry with the [$>$, $<$] constraint.

I determine that an *interval* i_0 is *covered* by a set of intervals $I = \{i_1, i_2, \ldots, i_k\}$ according to a conjunction of constraints *order* by constructing a point algebra table for all of these intervals, adding the constraints in *order* to the table, and computing the transitive closure. After this, the algorithm checks to see if i_0 is covered by looking at all pairs of intervals to see if they overlap. i_0 is not covered if either no intervals in I meet either i_0^- or i_0^+, there are any intervals that have an endpoint that is contained only by i_0 and do not meet the opposite side of another interval in I or an endpoint of i_0, or there are no intervals overlapping i_0. Otherwise, it is. For example, in Figure 4.1a, the parent plan A is covered by its subplans B, C, and D because B^- meets A^-; D^+ meets A^+; B^+ is contained by C; C^- is contained by B; and C^+ meets D^-.[4] In Figure 4.1b, E is not covered because F^+ is not *necessarily* contained by any interval other than E, and it doesn't meet another interval. In Figure 4.1c, I is not covered because there are no intervals overlapping it.

4.3.2 Summarizing, requiring, and attempting to assert summary conditions

In order to determine whether abstract plan executions can achieve, clobber, or undo conditions of others, an agent needs to be able to reason about how summary conditions

[4] Arrows show precedence constraints over the endpoints of the intervals, and double ended arrows with '=' beside them indicate that the endpoints must occur at the same time.

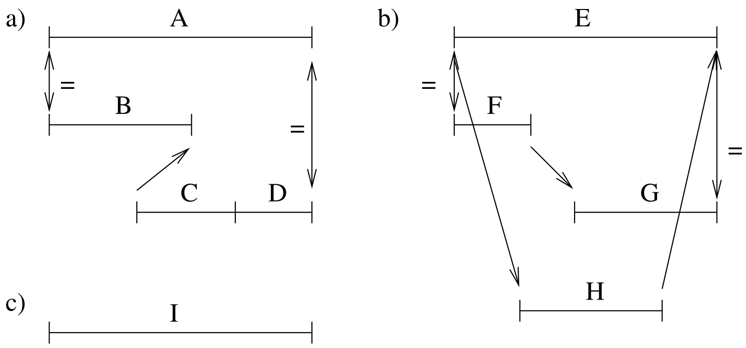


Figure 4.1: Subactivities' intervals *covering* their parent's interval

are asserted and required to be met. Ultimately, the agent needs to be able to determine whether a partial ordering of abstract plans can succeed, so it may be the case that an agent's action fails to assert a summary condition that is required by the action of another agent. Therefore, I formalize what it means for an action to *attempt* to assert a summary condition and to *require* that a summary condition be met. These definitions rely on linking the summary condition of a plan to the CHiP conditions it summarizes in the subplans of the plan's decompositions. Thus, I first define what it means for a summary condition to *summarize* these conditions.

> Definition: A summary condition c *summarizes* a *condition* ℓ in *condition set conds* of *plan* p iff c was added by the procedure for deriving summary information to a summary condition set of p'; $\ell = \ell(c)$; and either c was added for ℓ in a condition set *conds* of $p = p'$, or c was added for a summary condition of a subplan of p' that summarizes ℓ in *conds* of p.

For example, *at*(bin1, A) is a precondition of the $start\_move$ plan for moving part A from bin1 to machine M1 (as given in Section 3.2). When deriving the summary conditions for $start\_move$, *at*(bin1, A) is added to the summary preconditions. Thus, the summary precondition *must*, $first$ *at*(bin1, A) summarizes *at*(bin1, A) in the preconditions of $start\_move$. Because *at*(bin1, A) is added to the *move* plan's summary preconditions from the summary preconditions of $start\_move$, the *move* plan's summary precondition *must*, $first$ *at*(bin1, A) also summarizes *at*(bin1, A) in the preconditions of $start\_move$. Likewise, since this summary condition is propagated up the hierarchy to the top-level plan *produce_H*, *must*, $first$ *at*(bin1, A) in *produce_H*'s summary preconditions also summarizes *at*(bin1, A) in the preconditions of $start\_move$.

Definition: An execution e of p *requires* a *summary condition c to be met at t* iff c is a summary condition in p's summary information; there is a condition ℓ in a condition set *conds* of p' that is summarized by c; if $first(c)$, $t = t_s(e)$; if $last(c)$, $t = t_f(e)$; if $always(c)$, t is within $(t_s(e), t_f(e))$; and if $sometimes(c)$, there is an execution of a subplan of p in $d(e)$ that requires a summary condition c' to be met at t, and c' summarizes ℓ in *conds* of p'.

So, basically, an execution requires a summary condition to be met whenever the conditions it summarizes are required. The *build_G* execution (shown in Figure 3.1) has a summary precondition *must*, *first available*(M1). This execution requires this summary condition to be met at t_s(*build_G*) because *at*(A, M1_tray1) is a precondition of *build_G*'s first subplan that is summarized by *build_G*'s summary precondition. The *must*, *sometimes available*(M1) summary precondition of *produce_G_on_M*1 is also required by the plan's execution at t_s(*build_G*) because the summary condition summarizes the same *available*(M1) precondition of *build_G*'s first subplan; *build_G* is a subplan of *produce_G_on_M*1; *build_G*'s execution requires the *must*, *first available*(M1) summary precondition to be met at the same time (which is within the execution of *produce_G_on_M*1). Similarly, *produce_H*'s execution requires its *may*, *sometimes available*(M1) summary precondition to also be met at rt_s(*build_G*).

Similar to the previous definition: an execution e of p *attempts to assert* a *summary condition c* at t iff c is a summary condition in p's summary information; there is a condition ℓ in a condition set *conds* of p' that is summarized by c; $\neg first(c)$; if $always(c)$, t is in the smallest interval after $t_s(e)$ and before the start or end of any other execution that follows $t_s(e)$; if $last(c)$, $t = t_f(e)$; and if $sometimes(c)$, there is an execution of a subplan of p in $d(e)$ that attempts to assert a summary condition c' at t; and c' summarizes ℓ in *conds* of p'.

I say that an execution "attempts" to assert a summary condition because asserting a condition can fail due to a simultaneous assertion of the negation of the condition. Like the example above for requiring a summary condition, the executions of *build_G*, *produce_G_on_M*1, and *produce_H* all assert summary postconditions that M1 becomes available at t_f(*build_G*).

4.3.3 Definitions and algorithms for must/may asserting summary conditions

I have defined what it means for a plan execution to require or attempt to assert a summary condition at a particular point in time, but in order for agents to determine potential interactions among their abstract plans (such as clobbering or achieving), they need to reason about when a summary condition is asserted by one plan in relation to when it is asserted or required by another. Based on interval or point algebra constraints over a set of abstract plans, an agent specifically would need to be able to determine whether a plan would assert a summary condition *before* or *by* the time another plan requires or asserts a summary condition on the same state variable. In addition, to reason about clobbering inconditions, an agent would need to determine if a summary condition would be asserted during the time a summary incondition c was required (asserted *in* c). Agents also need to detect when a summary postcondition would be asserted at the same time as another summary postcondition c (asserted *when* c). I do not consider cases where executions attempt to assert a summary in- or postcondition at the same time an incondition is asserted because in these cases, clobber relations are already detected because executions always require the summary inconditions that they attempt to assert. For example, in Figure 3.1 if $equip\_M1$ attempted to assert the incondition that M1 was unavailable at the same time that $build\_G$ attempted to assert the postcondition that M1 was available, the incondition of $equip\_M1$ that requires M1 is unavailable would be clobbered by $build\_G$'s postcondition.

In the case that the ordering constraints allow for alternative synchronizations of the abstract plans, the assertions of summary conditions may come in different orders. Therefore, I formalize *must-assert* and *may-assert* to determine when these relationships must or may occur respectively. As mentioned at the beginning of Section 4.3, this use of "must" and "may" is based only on disjunctive orderings and not on the *existence* of summary conditions in different decompositions. For the following definitions and algorithms of must- and may-assert, I assume c and c' are summary conditions of plans in P. Soundness and completeness proofs of the algorithms are given in Appendix B.

Definition: $p' \in P$ *must-assert* c' [*by*, *before*] c iff for *all* histories $h \in H$ and all t where e is the top-level execution in $E(h)$ of some $p \in P$ that requires

			p' must-assert c' by c	p' must-assert c' before c
	$c' \in post(p')$	$c \in pre(p)$	*order* must impose these constraints	*order* must impose these constraints
	last	*first*		
1	T	T	$p'^{+} \leq p^{-}$	$p'^{+} < p^{-}$
2	T	F	$p'^{+} \leq p^{-}$	$p'^{+} < p^{-}$
3	F	?	$p'^{+} \leq p^{-}$	$p'^{+} < p^{-}$
4	?	?	$p'^{+} \leq p^{-}$	$p'^{+} < p^{-}$
	$c' \in in(p')$			
	always			
5	T	T	$p'^{-} < p^{-}$	$p'^{-} < p^{-}$
6	T	F	$p'^{-} \leq p^{-}$	$p'^{-} \leq p^{-}$
7	F	?	$false$	$false$
	$c' \in post(p')$	$c \in in(p)$		
	last	*always*		
8	T	?	$p'^{+} \leq p^{-}$	$p'^{+} \leq p^{-}$
9	F	?	$p'^{+} \leq p^{-}$	$p'^{+} \leq p^{-}$
	$c' \in in(p')$			
	always			
10	T	?	$p'^{-} \leq p^{-}$	$p'^{-} < p^{-}$
11	F	?	$false$	$false$
	$c' \in post(p')$	$c \in post(p)$		
	last	*last*		
12	T	T	$p'^{+} \leq p^{+}$	$p'^{+} < p^{+}$
13	T	F	$p'^{+} \leq p^{-}$	$p'^{+} \leq p^{-}$
14	F	T	$p'^{+} \leq p^{+}$	$p'^{+} \leq p^{+}$
15	F	F	$p'^{+} \leq p^{-}$	$p'^{+} \leq p^{-}$
	$c' \in in(p')$			
	always			
16	T	T	$p'^{-} < p^{+}$	$p'^{-} < p^{+}$
17	T	F	$p'^{-} \leq p^{-}$	$p'^{-} \leq p^{-}$
18	F	T	$false$	$false$
19	F	F	$false$	$false$

Table 4.1: Table for *must-assert by/before* algorithm

c to be met at t, and e' is the top-level execution of p' in $E(h)$, there is a t' where e' attempts to assert c' at t', and $[t' \leq t, t' < t]$.

Algorithm: $p' \in P$ *must-assert* c' [*by*, *before*] c if and only if for the row of Table 4.1 describing c' and c, all of the corresponding ordering constraints are imposed by (can be deduced from) *order* as described above using the point algebra table.

Definition: $p' \in P$ *may-assert* c' [*by*, *before*] c iff for *some* history $h \in H$, there exists a $p \in P$, e, e', t, and t' such that e is the top-level execution of p in $E(h)$; e' is top-level execution of p' in $E(h)$; e' attempts to assert c' at t'; e attempts to assert c at t, and $[t' \leq t, t' < t]$.

Algorithm: $p' \in P$ *may-assert* c' [*by*, *before*] c if and only if for the row of Table 4.2 describing c' and c, none of the corresponding ordering constraints are imposed by (can be deduced from) *order* as described above using the point algebra table.

I illustrate these relationships for the example in Figure 4.2a. Here the agents' plans

			p' may-assert c' by c	p' may-assert c' before c
	$c' \in post(p')$	$c \in pre(p)$	*order* cannot impose these constraints	*order* cannot impose these constraints
	$last$	$first$		
1	T	T	$p'^{+} > p^{-}$	$p'^{+} \geq p^{-}$
2	T	F	$p'^{+} \geq p^{+}$	$p'^{+} \geq p^{+}$
3	F	T	$p'^{-} \geq p^{-}$	$p'^{-} \geq p^{-}$
4	F	F	$p'^{-} \geq p^{+}$	$p'^{-} \geq p^{+}$
	$c' \in in(p')$			
	$always$			
5	?	T	$p'^{-} \geq p^{-}$	$p'^{-} \geq p^{-}$
6	?	F	$p'^{-} \geq p^{+}$	$p'^{-} \geq p^{+}$
	$c' \in post(p')$	$c \in in(p)$		
	$last$	$always$		
7	T	T	$p'^{+} > p^{-}$	$p'^{+} > p^{-}$
8	T	F	$p'^{+} \geq p^{+}$	$p'^{+} \geq p^{+}$
9	F	T	$p'^{-} \geq p^{-}$	$p'^{-} \geq p^{-}$
10	F	F	$p'^{-} \geq p^{+}$	$p'^{-} \geq p^{+}$
	$c' \in in(p')$			
	$always$			
11	?	T	$p'^{-} > p^{-}$	$p'^{-} \geq p^{-}$
12	?	F	$p'^{-} \geq p^{+}$	$p'^{-} \geq p^{+}$
	$c' \in post(p')$	$c \in post(p)$		
	$last$	$last$		
13	T	T	$p'^{+} > p^{+}$	$p'^{+} \geq p^{+}$
14	T	F	$p'^{+} \geq p^{+}$	$p'^{+} \geq p^{+}$
15	F	T	$p'^{-} \geq p^{+}$	$p'^{-} \geq p^{+}$
16	F	F	$p'^{-} \geq p^{+}$	$p'^{-} \geq p^{+}$
	$c' \in in(p')$			
	$always$			
17	?	T	$p'^{-} \geq p^{+}$	$p'^{-} \geq p^{+}$
18	?	F	$p'^{-} \geq p^{+}$	$p'^{-} \geq p^{+}$

Table 4.2: Table for *may-assert by/before* algorithm

are unordered with respect to each other. Part G is produced either on machine M1 or M2 depending on potential decompositions of the *produce_G* plan. *produce_G must-assert* $c' = must$, $last\ available$(G) before $c = must$, $first\ available$(G) in the summary preconditions of *move_G* because no matter how the plans are decomposed (for all executions and all histories of the plans under the ordering constraints in the figure), the execution of *produce_G* attempts to assert c' before the execution of *move_G* requires c to be met. The algorithm verifies this by finding that the end of *produce_G* is ordered before the start of *move_G* (row 1 in Table 4.1). It is also the case that *equip_M2_tool may-assert* $c' = must$, $last\ \neg available$(M2) by $c = may$, $sometimes\ available$(M2) in the summary preconditions of *produce_G* because the two plans are unordered with respect to each other, and in some history *equip_M2_tool* can precede *produce_G*. The algorithm finds that this is true since *equip_M2* is not constrained to start after the start of *produce_G* (row 2 in Table 4.2).

> Definition: $p' \in P$ *must-assert* c' *in* c iff $always(c)$, and for *all* histories $h \in H$ and all t where there's a plan $p \in P$ such that $c \in in_{sum}(p)$; e is the top-level execution in $E(h)$ of p that requires c to be met at t; and e' is the top-level

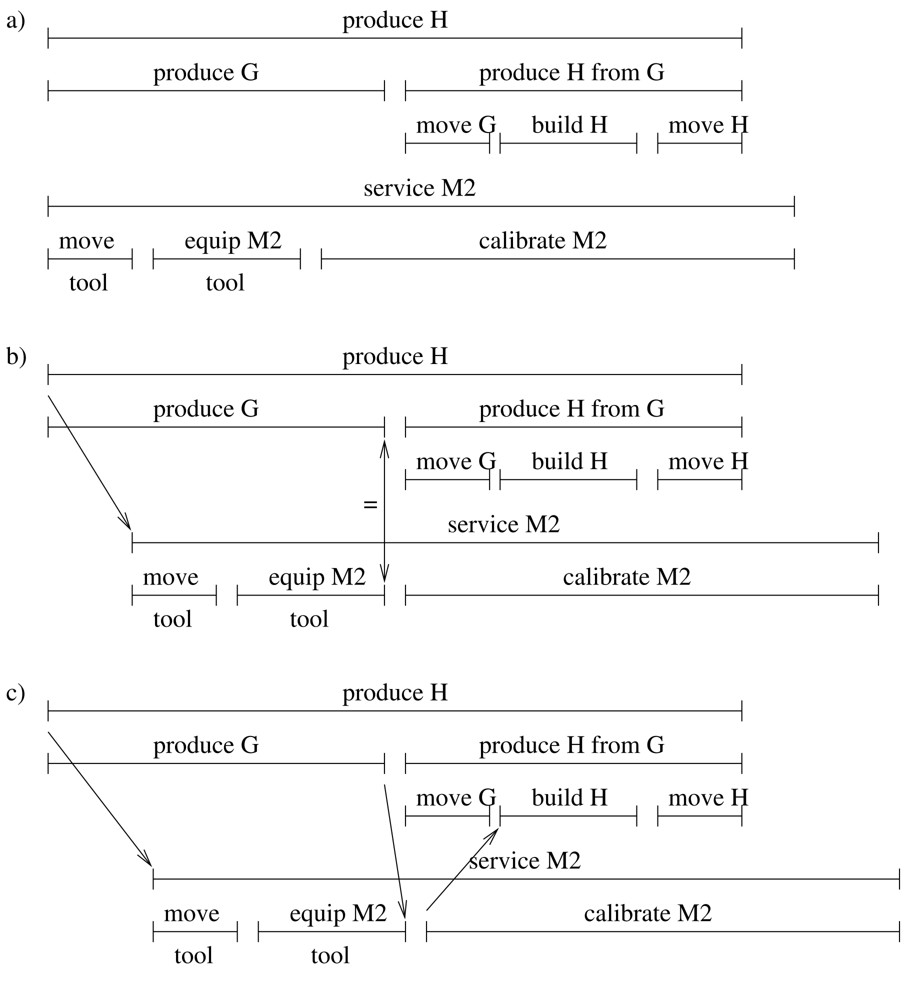


Figure 4.2: The production and facilities managers' plans partially expanded

			p' must-assert c' in c
	$c' \in post(p')$	$c \in in(p)$	*order* must impose these constraints
	last	*always*	
1	T	T	$p'^{+} > p^{-}$ and $p'^{+} < p^{+}$
2	T	F	$false$
3	F	T	$p'^{-} \geq p^{-}$ and $p'^{+} \leq p^{+}$
4	F	F	$false$
	$c' \in in(p')$		
	always		
5	T	T	$p'^{-} \geq p^{-}$ and $p'^{-} < p^{+}$
6	T	F	$false$
7	F	T	$false$
8	F	F	$false$
			p' may-assert c' in c
	$c' \in post(p')$	$c \in in(p)$	*order* cannot impose these constraints
	last	*always*	
1	T	T	$p'^{+} \leq p^{-}$ or $p'^{+} \geq p^{+}$
2	T	F	$p'^{+} \leq p^{-}$ or $p'^{+} \geq p^{+}$
3	F	T	$p'^{+} \leq p^{-}$ or $p'^{-} \geq p^{+}$
4	F	F	$p'^{+} \leq p^{-}$ or $p'^{-} \geq p^{+}$
	$c' \in in(p')$		
	always		
5	T	T	$p'^{+} \leq p^{-}$ or $p'^{-} \geq p^{+}$
6	T	F	$p'^{+} \leq p^{-}$ or $p'^{-} \geq p^{+}$
7	F	T	$p'^{+} \leq p^{-}$ or $p'^{-} \geq p^{+}$
8	F	F	$p'^{+} \leq p^{-}$ or $p'^{-} \geq p^{+}$

Table 4.3: Table for *must/may-assert in* algorithm

execution of p' in $E(h)$, there is a t' where e' attempts to assert c' at t', and $t_s(e) < t' < t_f(e)$.

Algorithm: $p' \in P$ *must-assert* c' *in* c if and only if for the row of Table 4.3 describing c' and c, all of the corresponding ordering constraints are imposed by (can be deduced from) *order* as described above using the point algebra table.

Definition: $p' \in P$ *may-assert* c' *in* c iff for *some* history $h \in H$, there exists a $p \in P$, e, e', t, and t' such that $c \in in_{sum}(p)$; e is the top-level execution p in $E(h)$; e' is top-level execution of p' in $E(h)$; e' attempts to assert c' at t'; and $t_s(e) < t' < t_f(e)$.

Algorithm: $p' \in P$ *may-assert* c' *in* c if and only if for the row of the table describing c' and c, none of the corresponding ordering constraints are imposed by (can be deduced from) *order* as described above using the point algebra table.

In Figure 4.2b, *move_tool may-assert* $c' = must$, $last\ free$(transport1) in $c = may$, $sometimes\ \neg free$(transport1) in *produce_G*'s summary inconditions because in some history *move_tool* attempts to assert c' during the time that *produce_G* is using transport1

			p' must-assert c' when c
	$c' \in post(p')$	$c \in post(p)$	*order* must impose these constraints
	last	*last*	
1	T	T	$p'^{+} = p^{+}$
2	T	F	*false*
3	F	T	*false*
4	F	F	*false*
			p' may-assert c' when c
	$c' \in post(p')$	$c \in post(p)$	*order* cannot impose these constraints
	last	*last*	
1	T	T	$p'^{+} \neq p^{+}$
2	T	F	$p'^{+} \leq p^{-}$ or $p'^{+} \geq p^{+}$
3	F	T	$p'^{+} \leq p^{+}$ or $p'^{-} \geq p^{+}$
4	F	F	$p'^{+} \leq p^{-}$ or $p'^{-} \geq p^{+}$

Table 4.4: Table for *must/may-assert when* algorithm

to move part A to machine M2. The algorithm determines this by finding that *move_tool* ends neither before the start of *produce_G* nor after the end of *produce_G* (row 2 in the *may-assert* part of Table 4.3). It is *not* the case that *move_tool must-assert* c' in c because c is *sometimes* and may only be required in a subinterval of *produce_G* that does not overlap the end of *move_tool*. According to the algorithm, *must-assert* is false for this case corresponding to row 2 in the *must-assert* part of Table 4.3.

Definition: $p' \in P$ *must-assert* c' *when* c iff $c' \in post_{sum}(p')$; $last(c')$; and for *all* histories $h \in H$ and all t where e is the top-level execution in $E(h)$ of some $p \in P$ that requires c to be met at t, and e' is the top-level execution of p' in $E(h)$, there is a t' where e' attempts to assert c' at t', and $t = t'$.

Algorithm: $p' \in P$ *must-assert* c' *when* c if and only if for the row of Table 4.4 describing c' and c, all of the corresponding ordering constraints are imposed by (can be deduced from) *order* as described above using the point algebra table.

Definition: $p' \in P$ *may-assert* c' *when* c iff for *some* history $h \in H$, there exists a $p \in P$, e, e', t, and t' such that $c \in post_{sum}(p)$; e is the top-level execution p in $E(h)$; e' is top-level execution of p' in $E(h)$; e' attempts to assert c' at t'; and $t = t'$.

Algorithm: $p' \in P$ *may-assert* c' *when* c if and only if for the row of the table describing c' and c, none of the corresponding ordering constraints are imposed by (can be deduced from) *order* as described above using the point algebra table.

In Figure 4.2b, $equip\_M2\_tool$ *must-assert* $c' = must$, $last\ \neg available$(M2) when $c = may$, $last\ available$(M2) in $produce\_G$'s summary postconditions because $equip\_M2\_tool$ attempts to assert c' at the same time that $produce\_G$ requires c to be met. The algorithm agrees according to row 1 of the *must-assert* part of Table 4.4. It is *not* true that $equip\_M2\_tool$ *may-assert* $c' = must$, $last\ \neg available$(M2) when $c = must$, $sometimes\ available$(M2) in $produce\_H\_from\_G$'s summary postconditions because there are no histories where the end of $equip\_M2\_tool$ can be at the same time as the end of $produce\_H\_from\_G$. The algorithm verifies this by finding that the end of $equip\_M2\_tool$ is constrained to precede the end of $produce\_H\_from\_G$, falsifying the rule for row 2 in the *may-assert* part of Table 4.4.

4.3.4 Definitions and algorithms for must/may clobber, achieve, and undo

Since I have shown how an agent can detect particular orderings of assertions and requirements of summary conditions, I can now specify how the agent can use these definitions and algorithms to determine when clobbering, achieving, and undoing interactions must or may occur. Below I give definitions for must/may achieve, clobber, and undo and algorithms that an agent can use to discover such interactions. The use of "must" and "may" in these definitions takes into account disjunctive orderings of abstract plans as well as the *existence* of their summary conditions based on multiple decomposition choices.

The soundness and completeness proofs of these are actually trivial given the lemmas that established the soundness and completeness of must/may assert relations and the fact that the algorithms for determining these properties directly follow their definitions. Therefore, I only give a proof for the soundness of must-clobber and must-achieve (in Appendix B). In Appendix D, I prove the circumstances under which summarized conditions will be met depending on whether summary conditions are must- or may-clobbered. These lemmas are central to the soundness and completeness proofs of the *CanAnyWay* and *MightSomeWay* algorithms described in Section 5.1 that determine the potential success or failure of a set of abstract plans.

Definition: Plan $p' \in P$ *must*-[*achieve*, *clobber*] $c\ in\ pre_{sum}(p)$ iff there is a

c'; $\ell(c') \Leftrightarrow [\ell(c), \neg\ell(c)]$; and for *all* histories $h \in H$, where e is the top-level execution of p in $E(h)$, and e' is the top-level execution of $p' \in P$ in $E(h)$; if e requires c to be met, e' attempts to assert c' before or at the time e requires c to be met and after any other attempt to assert some c'' (where $\ell(c'') \Leftrightarrow [\neg\ell(c), \ell(c)]$) before or at the time e requires c to be met; and there are no $p'' \in P$ and c'' where $p'' \neq p'$; $\ell(c'') \Leftrightarrow [\ell(c), \neg\ell(c)]$; and for *all* histories $h'' \in H$ where e'' is the top-level execution p'' in $E(h'')$, e' attempts to assert c' before e'' attempts to assert c'', and e'' attempts to assert c'' before or at the time e requires c to be met.

For example, in Figure 4.2c, $p' = build\_H$ must-achieve $c = must, first\ available$(H) in the summary preconditions of $p = move\_H$. Here, c' is $must$, $last\ available$(H) in the summary postconditions of $build\_H$. In all histories, the execution of $build\_H$ attempts to assert c' before the execution of $move\_H$ requires c to be met, and there is no other execution (e'') that attempts to assert a condition (c'') on the availability of H. $equip\_M2\_tool$ does *not* must-clobber $c = must, first\ available$(M2) in the summary preconditions of $build\_H$ even though $equip\_M2\_tool$ attempts to assert $c' = must, last\ \neg available$(M1) before c is required to be met for *all* histories. The reason that $equip\_M2\_tool$ does not must-clobber c is not because there is a plan that could assert $available$(M2) (c'') after $equip\_M2\_tool$ attempts to assert c' and before c is required—$produce\_G$ attempts to assert $c'' = must, last\ available$(M2), but that happens before $equip\_M2\_tool$ attempts to assert c'. The reason is that $p'' = calibrate\_M2$ attempts to assert the summary incondition $c'' = must, always\ \neg available$(M2) between the time that $equip\_M2\_tool$ attempts to assert c' and when c is required. This means that $calibrate\_M2$ must-clobber c. This last restriction ensures that there is only one achiever or clobberer unless there are multiple plans that achieve/clobber at the same time. In this way, a planner or coordinator can identify the plan directly *causing* the achieving/clobbering.

Algorithm: $p' \in P$ *must*-[*achieve*, *clobber*] *c in* $pre_{sum}(p)$ if and only if there is a c' such that $p' \in P$ must-assert c' by c; $must(c')$; $\ell(c') \Leftrightarrow [\ell(c), \neg\ell(c)]$; and by checking inside a loop through the plans in P_{sum}, there is no p'' and c'' such that p' may-assert c' before c''; p'' may-assert c'' by c; and $\ell(c'') \Leftrightarrow [\neg\ell(c), \ell(c)]$; or p' must-assert c' before c''; p'' must-assert c'' by c; $\ell(c'') \Leftrightarrow$

$[\ell(c), \neg\ell(c)]$; and c'' is *must*.

The complexity of walking through the summary conditions checking for p'' and c'' is $O(nc)$ for c summary conditions for each of n plans represented in P_{sum}. It is possible to improve the algorithm by hashing the summary conditions based on the literal they summarize and only process plans with matching or contradicting literals. Then the algorithm does not have to visit conditions summarizing different propositional variables. However, in the worst case, all summary conditions summarize the same propositional variable, and all $O(nc)$ conditions must be visited, so the worst case complexity is not further reduced. The complexity of the other must/may achieve/clobber/undo algorithms described below is the same by similar argument.

> Definition: Plan $p' \in P$ *may*-[*achieve*, *clobber*] c *in* $pre_{sum}(p)$ iff there is a c' where $\ell(c') \Leftrightarrow [\ell(c), \neg\ell(c)]$, and for *some* history $h \in H$, e is the top-level execution p in $E(h)$; e' is the top-level execution of $p' \in P$ in $E(h)$; e' attempts to assert c' before or at the time e requires c to be met and after any other attempt to assert some c'' (where $\ell(c'') \Leftrightarrow \ell(c)$ or $\neg\ell(c)$ before or at the time e requires c to be met.

For example, in Figure 4.2a, $p' = equip\_M1\_tool$ may-clobber $c = may$, *sometimes available*(M2) in the summary preconditions of $p = produce\_G$ because there is *some* history where $equip\_M1\_tool$ ends before $produce_G$ starts, and $calibrate\_M2$ starts after $produce\_G$ starts. Thus, $equip\_M1\_tool$ attempts to assert $c' = must$, *last* $\neg available$(M2) before c is required to be met, and no other plan (p''), such as $calibrate\_M1$, can assert a condition (c'') on the availability of M1 in between p' and p. In Figure 4.2c, $p' = equip\_M1\_tool$ does *not* may-clobber $c = must$, *first available*(M2) in the summary preconditions of $p = build\_G$ because there is *no* history where $p'' = calibrate\_M2$ does not attempt to assert $c'' = must$, *always available*(M2) between $equip\_M1\_tool$ and $build\_G$. Note that must-[clobber, achieve] implies may-[clobber, achieve], and, thus, ¬may-[clobber, achieve] implies ¬must-[clobber, achieve].

> Algorithm: $p' \in P$ *may*-[*achieve*, *clobber*] c *in* $pre_{sum}(p)$ if and only if there is a c' such that $p' \in P$ may-assert c' by c; $\ell(c') \Leftrightarrow [\ell(c), \neg\ell(c)]$; and by checking inside a loop through the plans in P_{sum}, there is no p'' and c'' such

that p' must-assert c' before c''; p'' must-assert c'' by c; $\ell(c'') \Leftrightarrow \ell(c)$ or $\neg\ell(c)$; and c'' is *must*.

Definition: Plan $p' \in P$ *must-clobber c in* $in_{sum}(p)$ iff $always(c)$; there is a c' where $\ell(c') \Leftrightarrow \neg\ell(c)$; and for *all* histories $h \in H$, where e is the top-level execution $p \in P$ in $E(h)$, and e' is the top-level execution of p' in $E(h)$, if e requires c to be met, e' attempts to assert c' within e.

Algorithm: $p' \in P$ *must-clobber c in* $in_{sum}(p)$ if and only if there is a c' such that p' must-assert c' in c'; $must(c')$; and $\ell(c') \Leftrightarrow \neg\ell(c)$.

Definition: Plan $p' \in P$ *may-clobber c in* $in_{sum}(p)$ iff there is a c' where $\ell(c') \Leftrightarrow \neg\ell(c)$; and for *some* history $h \in H$, where e is the top-level execution $p \in P$ in $E(h)$, and e' is the top-level execution of p' in $E(h)$, e' attempts to assert c' within e.

Algorithm: $p' \in P$ *may-clobber c in* $in_{sum}(p)$ if and only if there is a c' such that p' may-assert c' in c; and $\ell(c') \Leftrightarrow \neg\ell(c)$.

For instance, in Figure 4.2c, the *equip_M2_tool* plan may-clobber $c = may$, *sometimes available*(M2) in the inconditions of *produce_G*, but it is not true that *equip_M2_tool* must clobber the incondition because only in *some* histories *equip_M2_tool* attempts to assert *must last* $\neg$*available*(M2) during the time that *produce_G* requires c to be met.

Definition: Plan $p' \in P$ *must-clobber c in* $post_{sum}(p)$ iff there is a c' where $\ell(c') \Leftrightarrow \neg\ell(c)$, and for *all* histories $h \in H$, where e is the top-level execution $p \in P$ in $E(h)$, and e' is the top-level execution of p' in $E(h)$, if e requires c to be met, e' attempts to assert c' at the same time e attempts to assert c.

Algorithm: $p' \in P$ *must-clobber c in* $post_{sum}(p)$ if and only if there is a c' such that p' must-assert c' when c; $must(c')$; and $\ell(c') \Leftrightarrow \neg\ell(c)$.

Definition: Plan $p' \in P$ *may-clobber c in* $post_{sum}(p)$ iff there is a c' where $\ell(c') \Leftrightarrow \neg\ell(c)$; and for *some* history $h \in H$, where e is the top-level execution $p \in P$ in $E(h)$, and e' is the top-level execution of p' in $E(h)$, e' attempts to assert c' at the same time e attempts to assert c.

Algorithm: $p' \in P$ *may-clobber c in* $post_{sum}(p)$ if and only if there is a c' such that p' may-assert c' when c; and $\ell(c') \Leftrightarrow \neg\ell(c)$.

For example, in Figure 4.2b, *produce_G* may-clobber $c = must, last\ \neg available$(M2) in the summary postconditions of *equip_M2_tool* because in some history, the production manager produces part G on M2 (instead of M1), and releases the machine at the same time that *equip_M2_tool* attempts to assert c. It is not true that *produce_G* must-clobber c because in some histories it uses machine M1 to produce G, resulting in no conflict. However, it is true that *equip_M2_tool* must-clobber *may*, *last available*(M2) in *produce_G*'s summary postconditions because in all histories *that produce_G uses M2*, *equip_M2_tool* attempts to assert *must*, *last* $\neg$*available*(M2) at the same time that *produce_G* requires *may*, *last available*(M2) to be met. This is why the algorithm for must-clobber only requires that c' is *must* and not c. This is necessary to ensure that *may* summary preconditions are not propagated as external preconditions of the parent CHiP when they must be achieved or clobbered.

As mentioned in Section 3.5, achieving inconditions and postconditions does not make sense for this formalism. Therefore, we do not define must/may achieving of summary inconditions and postconditions.

> Definition: Plan $p' \in P$ *must-undo* c *in* $post_{sum}(p)$ iff there is a c' where $\ell(c') \Leftrightarrow \neg\ell(c)$, and for *all* histories $h \in H$, where e is the top-level execution p in $E(h)$, and e' is the top-level execution of $p' \in P$ in $E(h)$; if e attempts to assert c, it does so before or at the time e' attempts to assert c' and after any other attempt to assert some c'' (where $\ell(c'') \Leftrightarrow \ell(c)$) before e' attempts to assert c'; and there are no $p'' \in P$ and c'', where $p'' \neq p'$, $\ell(c'') \Leftrightarrow \neg\ell(c)$, and for *all* histories $h'' \in H$ where e'' is the top-level execution p'' in $E(h'')$, e attempts to assert c before or at the time e'' attempts to assert c'', and e'' attempts to assert c'' before e' attempts to assert c'.
>
> Algorithm: $p' \in P$ *must-undo* c *in* $post_{sum}(p)$ if and only if there is a c' such that p must-assert c by c'; $must(c')$; $\ell(c') \Leftrightarrow \neg\ell(c)$; and by checking inside a loop through the plans P_{sum}, there is no p'' and c'' such that p may-assert c by c''; p'' may-assert c'' before c'; and $\ell(c'') \Leftrightarrow \ell(c)$; or p must-assert c by c''; p'' must-assert c'' before c'; $\ell(c'') \Leftrightarrow \neg\ell(c)$; and c'' is *must*.

In Figure 4.2, $p' = build\_H$ must-undo $c = must$, *last available*(G) in the summary postconditions of $p = move\_G$ because in *all* histories *build_H* attempts to assert $c' =$

$must$, $sometimes\ available$(G) after $move\_G$ asserts c, and there is no p'' that attempts to assert a condition on the availability of G (c'') between $move\_G$ and $build\_H$. In Figure 4.2c, it is not true that $p' = equip\_M2\_tool$ must-undo $c = may$, $last\ available$(M2) in the summary postconditions of $p = produce\_G$ even though in all histories $equip\_M2\_tool$ attempts to assert $c' = must$, $last\ \neg available$(M2) after c is asserted, and no *other* plan attempts to assert a condition about the availability of M2 inbetween. This may seem counterintuitive, but since $equip\_M2\_tool$ attempts to assert its summary incondition $c'' = must$, $sometimes\ \neg available$(M2) in some histories after c and before c', c' is not always the first to undo c. Thus, even though $equip\_M2\_tool$ will undo c in all histories, because the summary condition that undoes c is uncertain, must-clobber is false.

> Definition: Plan $p' \in P$ *may-undo* c *in* $post_{sum}(p)$ iff there is a c' where $\ell(c') \Leftrightarrow \neg\ell(c)$, and for *some* history $h \in H$, e is the top-level execution p in $E(h)$; e' is the top-level execution of $p' \in P$ in $E(h)$; e attempts to assert c before or at the time e' attempts to assert c' and after any other attempt to assert some c'' (where $\ell(c'') \Leftrightarrow \ell(c)$ or $\neg\ell(c)$) before e' attempts to assert c'.
>
> Algorithm: $p' \in P$ *may-undo* c *in* $post(p)$ if and only if there is a c' such that $p \in P$ may-assert c by c'; $\ell(c') \Leftrightarrow \neg\ell(c)$; and by checking inside a loop through the plans in P_{sum}, there is no p'' and c'' such that p must-assert c by c''; p'' must-assert c'' before c'; $\ell(c'') \Leftrightarrow \ell(c)$ or $\neg\ell(c)$; and c'' is $must$.

For the must-undo example above referring to Figure 4.2c, $equip\_M2\_tool$ may-undo $c = may$, $last\ available$(M2) in $produce\_G$'s summary postconditions either with its summary incondition or its summary postcondition. However, it is *not* true that $calibrate\_M2$ may-undo c because in *all* histories $p'' = equip\_M2\_tool$ attempts to assert its summary postcondition (c'') after c is asserted and before $calibrate\_M2$ asserts its summary incondition $c' = must$, $somtimes\ \neg available$(M2).

4.4 Summary Resource Usage

In this section, I extend summary information to include metric resources. I define a representation for capturing ranges of usage both local to the task interval and the depleted usage lasting after the end of the interval. Based on this I introduce a summariza-

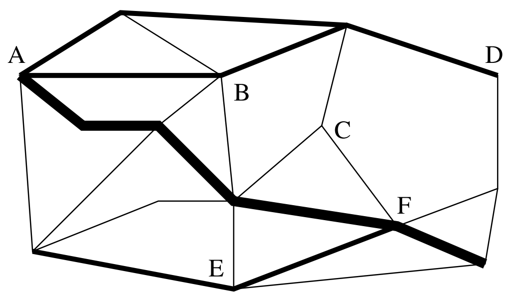


Figure 4.3: Example map of established paths between points in a rover domain

tion algorithm that captures in these ranges the uncertainty represented by decomposition choices in *or* plans and partial temporal orderings of *and* plan subtasks. This representation allows a coordinator or planner to reason about the potential for conflicts for a set of tasks. This reasoning will be discussed later in Section 5.2.

4.4.1 Representation

As an example, I will focus on coordinating a collection of rovers as they explore the environment around a lander on Mars. This exploration takes the form of visiting different locations and making observations. Each traversal between locations follows established paths to minimize effort and risk. These paths combine to form a network like the one mapped out in Figure 4.3, where vertices denote distinguished locations, and edges denote allowed paths. Thinner edges are harder to traverse, and labeled points have associated observation goals. While some paths are over hard ground, others are over loose sand where traversal is harder since a rover can slip.

Figure 4.4 gives an example of an abstract task. Imagine a rover that wants to make an early morning trip from point *A* to point *B* on the example map. During this trip the sun slowly rises above the horizon giving the rover the ability to progressively use *soak rays* tasks to provide more solar power (a non-depletable resource) to motors in the wheels. In addition to collecting photons, the morning traverse moves the rover, and the resultant *go* tasks require path dependent amounts of power. While a rover traveling from point *A* to point *B* can take any number of paths, the shortest three involve following one, two, or three steps.

A *summarized resource usage* consists of ranges of potential resource usage amounts during and after performing an abstract task, and I represent this summary information

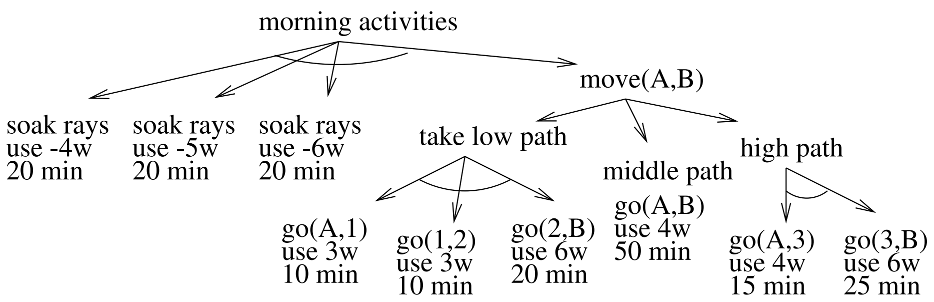


Figure 4.4: *and/or* tree defining abstract tasks

using the structure

$$\langle local\_min\_range, local\_max\_range, persist\_range \rangle,$$

where the resource's local usage occurs within the task's execution, and the persistent usage represents the usage that lasts after the task terminates for depletable resources.

The usage ranges capture the multiple possible usage profiles of a task with multiple decomposition choices and timing choices among loosely constrained subtasks. For example, the *high path* task has a ⟨[4,4],[6,6],[0,0]⟩ summary power use over a 40 minute interval. In this case the ranges are single points due to no uncertainty – the task simply uses 4 watts for 15 minutes followed by 6 watts for 25 minutes. The *move*(A,B) provides a slightly more complex example due to its decompositional uncertainty. This task has a ⟨[0,4],[4,6],[0,0]⟩ summary power use over a 50 minute interval. In both cases the *persist_range* is [0,0] because solar power is a nondepletable resource.

While a summary resource usage structure has only one range for persistent usage of a resource, it has ranges for both the minimum and maximum local usage because resources can have minimum as well as maximum usage limits, and it is necessary to detect whether a conflict occurs from violating either of these limits. As an example of reasoning with resource usage summaries, suppose that only 3 watts of power were available during a *move*(A,B) task. Given the [4,6] *local_max_range*, we know that there is an unresolvable problem without decomposing further. Raising the available power to 4 watts makes the task executable depending on how it gets decomposed and scheduled, and raising to 6 or more watts makes the task executable for all possible decompositions.

4.4.2 Resource Summarization Algorithm

The state summarization algorithm in Section 4.1 recursively propagates summary conditions upwards from an *and/or* hierarchy's leaves, and the algorithm for resource summarization takes the same approach. Starting at the leaves, the algorithm finds primitive tasks that use constant amounts of a resource. The resource summary of a task using x units of a resource is $\langle[x,x],[x,x],[0,0]\rangle$ or $\langle[x,x],[x,x],[x,x]\rangle$ over the task's duration for nondepletable or depletable resources respectively.

Moving up the *and/or* tree the summarization algorithm either comes to an *and* or an *or* branch. For an *or* branch the combined summary usage comes from the *or* computation

$$\begin{aligned}
\langle[&min_{c\in children}(lb(local_min_range(c))),\\
&max_{c\in children}(ub(local_min_range(c)))],\\
[&min_{c\in children}(lb(local_max_range(c))),\\
&max_{c\in children}(ub(local_max_range(c)))],\\
[&min_{c\in children}(lb(persist_range(c))),\\
&max_{c\in children}(ub(persist_range(c)))]\rangle,
\end{aligned}$$

where $lb()$ and $ub()$ extract the lower bound and upper bound of a range respectively. The *children* denote the branch's children with their durations extended to the length of the longest child. This duration extension alters a child's resource summary information because the child's usage profile has a zero resource usage during the extension. For instance, in determining the resource usage for *move*(A,B), the algorithm combines two 40 minute tasks with a 50 minute task. The resulting summary information describes a 50 minute abstract task whose profile might have a zero watt power usage for 10 minutes. This extension is why *move*(A,B) has a [0,4] for a $local\_min\_range$ instead of [3,4]. Planners that reason about variable durations could use [3,4] for a duration ranging from 40 to 50.

Computing an *and* branch's summary information is a bit more complicated due to timing choices among loosely constrained subtasks. The *take x path* examples illustrate the simplest sub-case, where subtasks are tightly constrained to execute serially. Here profiles are appended together, and the resulting summary usage information comes from the SERIAL-AND computation

$$\begin{aligned}
&\langle[min_{c\in children}(lb(local_min_range(c))+\Sigma_{lb}^{pre}(c)),\\
&\quad min_{c\in children}(ub(local_min_range(c))+\Sigma_{ub}^{pre}(c))],\\
&\ [max_{c\in children}(lb(local_max_range(c))+\Sigma_{lb}^{pre}(c)),\\
&\quad max_{c\in children}(ub(local_max_range(c))+\Sigma_{ub}^{pre}(c))],\\
&\ [\Sigma_{c\in children}(lb(persist_range(c))),\\
&\quad \Sigma_{c\in children}(ub(persist_range(c)))]\rangle,
\end{aligned}$$

where $\Sigma_{lb}^{pre}(c)$ and $\Sigma_{ub}^{pre}(c)$ are the respective lower and upper bounds on the cumulative persistent usages of children that execute before c. These computations have the same form as the Σ computations for the final *persist_range*.

The case where all subtasks execute in parallel and have identical durations is slightly simpler. Here the usage profiles add together, and the branch's resultant summary usage comes from the PARALLEL-AND computation

$$\begin{aligned}
&\langle[\Sigma_{c\in children}(lb(local_min_range(c))),\\
&\quad max_{c\in children}(ub(local_min_range(c))+\Sigma_{ub}^{non}(c))],\\
&\ [min_{c\in children}(lb(local_max_range(c))+\Sigma_{lb}^{non}(c)),\\
&\quad \Sigma_{c\in children}(ub(local_max_range(c)))],\\
&\ [\Sigma_{c\in children}(lb(persist_range(c))),\\
&\quad \Sigma_{c\in children}(ub(persist_range(c)))]\rangle,
\end{aligned}$$

where $\Sigma_{ub}^{non}(c)$ and $\Sigma_{lb}^{non}(c)$ are the respective sums of $local\_max\_range$ upper bounds and $local\_min\_range$ lower bounds for all children except c.

To handle *and* tasks with loose temporal constraints, I consider all legal orderings of child task endpoints. For example, in the rover's early morning tasks, there are three serial solar energy collection subtasks running in parallel with a subtask to drive to location B. Figure 4.5 shows one possible ordering of the subtask endpoints, which breaks the *move*(A,B) into three pieces, and two of the *soak rays* children in half. Given an ordering, the summarization algorithm can (1) use the endpoints of the children to determine subintervals, (2) compute summary information for each child task/subinterval combination, (3) combine the parallel subinterval summaries using the PARALLEL-AND computation, and then (4) chain the subintervals together using the SERIAL-AND computation.

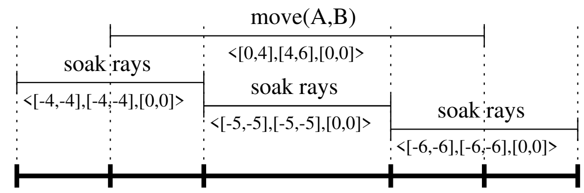


Figure 4.5: Possible task ordering for a rover's morning activities, with resulting subintervals.

Finally, the *and* task's summary is computed by combining the summaries for all possible orderings using an *or* computation.

Here I describe how step (2) generates different summary resource usages for the subintervals of a child task. A child task with summary resource usage $\langle[a,b],[c,d],[e,f]\rangle$ contributes one of two summary resource usages to each intersecting subinterval[5]:

$$\langle[a,b],[c,d],[0,0]\rangle, \langle[a,d],[a,d],[0,0]\rangle.$$

While the first usage has the tighter [a,b],[c,d] local ranges, the second has looser [a,d],[a,d] local ranges. Since the b and c bounds only apply to the subintervals containing the subtask's minimum and maximum usages, the tighter ranges apply to one of a subtask's intersecting subintervals. While the minimum and maximum usages may not occur in the same subinterval, symmetry arguments let us connect them in the computation. Thus one subinterval has tighter local ranges and all other intersecting subintervals get the looser local ranges, and the extra complexity comes from having to investigate all subtask/subinterval assignment options. For instance, there are three subintervals intersecting *move*(A,B) in Figure 4.5, and three different assignments of summary resource usages to the subintervals: placing [0,4],[4,6] in one subinterval with [0,6],[0,6] in the other two. These placement options result in a subtask with n subintervals having n possible subinterval assignments. So if there are m child tasks each with n alternate assignments, then there are n^m combinations of potential subtask/subinterval summary resource usage assignments. Thus propagating summary information through an *and* branch is exponential in the number of subtasks with multiple internal subintervals. However since the number of subtasks is controlled by the domain modeler and is usually bounded by a constant, this computation is tractable. In addition, summary information can often be

[5]For summary resource usages of the last interval intersecting the child task, we replace $[0,0]$ with $[e,f]$ in the *persist_range*.

derived offline for a domain. The propagation algorithm takes on the form:

- For each consistent ordering of endpoints:
  - For each consistent subtask/subinterval summary usage assignment:
    - Use PARALLEL-AND computations to combine subtask/subinterval summary usages by subinterval.
    - Use a SERIAL-AND computation on the subintervals' combined summary usages to get a consistent summary usage.
- Use *or* computation to combine all consistent summary usages to get *and* task's summary usage.

4.5 Summary of Formalisms

In this section, I defined summary information in terms of the properties of summary conditions. A summary condition's *existence* is either *must* or *may* depending on whether it must hold for all or just some decompositions. The *timing* of a summary condition is either *first*, *last*, *always*, or *sometimes*, specifying when the condition must hold in the plan's interval of execution. I described an algorithm for preprocessing a plan library to derive summary conditions for each CHiP. This algorithm relies on supporting mechanisms for temporal reasoning and for identifying interactions between pairs of plans based on their summary conditions and temporal constraints. I pointed to a proof in Appendix C for the soundness and completeness of the algorithm to derive summary conditions with their intended properties. I gave algorithms for determining the *timing* of a summary condition using a point algebra table to reason about the partial orderings of plan interval endpoints. In order to formalize higher level plan interactions, I first defined what it means for a summary condition of a plan to summarize a condition in the plan's decomposition. I also defined what it means for a plan execution to require or assert a summary condition.

Based on these definitions, I gave algorithms for determining when one plan execution asserts a summary condition with respect to the time that a summary condition of another plan is required or asserted based on the temporal constraints among the plans. These algorithms also determine whether these assertion relations *must* or *may* hold for the plans. I used these algorithms as functional blocks of higher level algorithms for determining whether one plan must or may achieve, clobber, or undo a summary condition of another. All of these algorithms are proven sound and complete and are used to derive summary conditions and identify threats in a set of partially ordered plans (as shown in the next section).

Summary resource usage is represented as three value ranges, $\langle$*local_min_range*, *local_max_range*, *persist_range*$\rangle$, where the resource's local usage occurs within the task's execution, and the persistent usage represents the usage that lasts after the task terminates for depletable resources. The summarization algorithm for an abstract task takes the summary resource usages of its subtasks, considers all legal orderings of the subtasks, and all possible usages for all subintervals within the interval of the abstract task to build multiple usage profiles. These profiles are combined with algorithms for computing parallel, sequential, and disjunctive usages to give the resultant summary usage of the parent task.

CHAPTER 5

Identifying Threats at Abstract Levels

Up to this point, I have detailed sound and complete algorithms for deriving summary conditions and for reasoning about potential (*may*) and definite (*must*) interactions between tasks in the context of those of other agents based on their summary information. In addition, I have outlined algorithms for deriving summarized resource usage but have not yet discussed interactions among tasks based on this information. In this chapter, I show how the interactions of summary conditions and summarized metric resource usage identify potentially resolvable threats and unresolvable conflicts among the plans of a group of agents. In Section 5.3, I summarize the main ideas developed in this chapter and review the material for this first part of the thesis to carry into subsequent parts.

5.1 Summary Conditions

With the properties of summary information proven, agents can safely reason about the interactions of their abstract plans without decomposing them. Agents can attempt to resolve conflicts among their plans by considering commitments to particular decompositions and ordering constraints. In order to do this, the agents must be able to identify remaining conflicts among their plans. Here I present an algorithm for identifying threats between abstract plans and their required conditions and show that it is sound and complete.

Formally, for a set of CHiPs P with ordering constraints *order*, a *threat* between an abstract plan $p \in P$ and a summary condition c' of another plan $p' \in P$ exists iff p may-clobber c'. The threat is unresolvable if p must-clobber c' and $must(c')$.[1] According to the lemmas in Appendix D, if there is no plan that may-clobber c', then any condition summarized by c' will be met no matter how the plans in P are decomposed and executed. However, if c' is must-clobbered, then all conditions summarized by c' will be clobbered for any execution of P. Thus, the potential success of an agent's plans depends on whether their conditions are threatened and whether those threats are resolvable.

So, a simple algorithm for identifying threats is to check to see if each of the $O(nc)$ summary conditions of n plans in P_{sum} is must- or may-clobbered by any other plan. Since the complexity of checking to see if a particular condition is must- or may-clobbered is $O(nc)$, this algorithm's complexity is $O(n^2c^2)$. This algorithm is sound and complete because the must/may-clobber algorithms are proven sound and complete, and the algorithms check for all cases where pre-, in-, or postconditions are clobbered from the assertion of inconditions or postconditions.

In many coordination tasks, if agents could determine that under certain temporal constraints their plans can be decomposed in any way (*CanAnyWay*) or that under those constraints there is no way they can be successfully decomposed ($\neg$*MightSomeWay*), then they can make coordination decisions at abstract levels without entering a potentially costly search for valid plan merges at lower levels. These relations over CHiPs with ordering constraints are dependent on the potential for threats among the plans—if there are no threats, the *CanAnyWay* property is true; if there are no unresolvable conflicts, then *MightSomeWay* is true. Not only can agents potentially make coordination decisions at abstract levels, but they can also focus their planning/coordination on resolving the threats and avoid reasoning about details of plans where there are no conflicts. In this section, I present sound and complete algorithms for determining *CanAnyWay* and *MightSomeWay* relations based on summary information. For convenience, I will abbreviate *Can* with *C*, *Any* with *A*, *Way* with *W*, and so on.

Informally, [$CAW(order, P_{sum})$, $MSW(order, P_{sum})$] says that the temporal relations in *order* [*can*, *might*] hold for the set of plans P whose corresponding summary information is in the set P_{sum} for [*any way*, *some way*] that the plans may be executed. I could also describe $CanSomeWay(order, P_{sum})$ and $MightAnyWay(rel, P_{sum})$ in the same fashion, but because these relations depend on both decomposition choices

[1] It is possible for a plan to must-clobber a *may* condition. (See Section 4.3.4.)

and how the subexecutions are synchronized, their meanings are not clear. For example, $MightAnyWay(\{\text{d}\}, \{p_{sum}, q_{sum}\})$ could mean that the *during* relation might hold for some decompositions of the plans in such a way that the subplans can be synchronized in any way (within the constraints of *during* at the top level). However, it could also mean that for any way the plans are decomposed, there might be a synchronization of the plans such that they will succeed. One might call the former interpretation the *MightAnyWaySynchronize* relation, and the latter the *MightAnyWayDecompose* relation. *CanSomeWay* is similar, but *CanAnyWay* and *MightSomeWay* do not have such ambiguity.

For example, in Figure 5.1a, the three top-level plans of the managers are unordered with respect to each other. The leaf plans of the partially expanded hierarchies comprise P_{sum}. Arrows represent the constraints in *order*. $CAW(\{\}, \{produce\_G, maintenance, move\_parts\})$ is false because there are several conflicts over the use of machines and transports that could occur for certain executions of the plans as described in Section 4.3.4 for Figure 4.2. However, $MSW(\{\}, \{produce\_G, maintenance, move\_parts\})$ is true because the plans might in some way execute successfully as shown in Figure 5.1b. With the ordering constraints in Figure 5.1b, $CAW(\{\text{before}(1,0)$[2], $\text{before}(0,2)\}, \{produce\_G, maintenance, move\_parts\})$ is true because the plans can execute in any way consistent with these ordering constraints without conflict. Here are the formal definitions of *CAW* and *MSW*:

> Definition: [$CAW(order, P_{sum})$, $MSW(order, P_{sum})$] iff for [all, some] sets of plans P with summary information P_{sum} and [all, some] histories h where $E(h)$ includes an execution of each plan in P as well as its subexecutions, and all executions meet the constraints in *order*, then all executions in $E(h)$ succeed.

I assume here that each of the plans in P would successfully execute by itself (i.e. they cannot clobber their own conditions) and that the initial state of h is such that it does not conflict with any preconditions external to the plans in P. In other words, we assume that the external preconditions of the plans do not conflict with each other.[3] So the task

[2] The 1 and 0 here are indices of the summarized plans. 0 refers to *produce_G*, 1 refers to *maintenance*, and 2 refers to *move_parts*.

[3] One could additionally check the summary preconditions of the plans against each other or against a

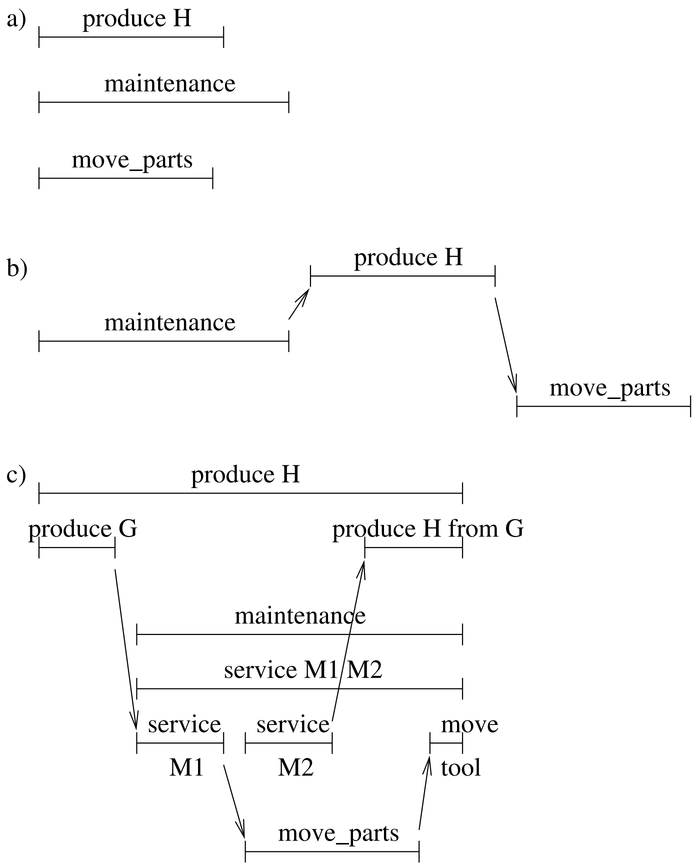


Figure 5.1: The top-level plans of each of the managers for the manufacturing domain

of determining whether CAW or MSW is true is basically to detect whether any plan may or must clobber a condition of another. In practice, if a planner/coordinator agent cannot assume that there are no conflicts within a plan, then conflicts can be resolved internally before reasoning about interactions with other plans.

> Algorithm: $[CAW(order, P_{sum}), MSW(order, P_{sum})]$ is *false* if and only if by checking inside two nested loops through the P_{sum}, there is a p_{sum} and p'_{sum} in P_{sum} and a summary condition c of p_{sum} such that one or more of the procedures for determining p' [may-clobber, must-clobber] c (using p'_{sum} as the summary information for p') returns $true$, and c is [*may* or $must$, $must$].

Figure 4.2c is an example where MSW is false because *calibrate_M*2 must-clobber the $must$, $first\ available$(M2) summary precondition of *build_H*. The complexity of this

particular initial state for conflicts. This could be simply done by treating the initial state as the external postconditions of a plan ordered before any plan in P.

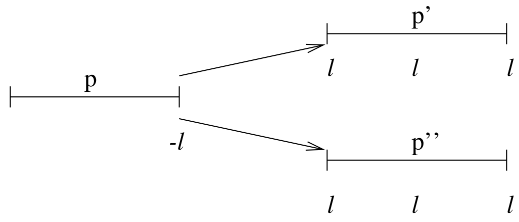


Figure 5.2: $\neg MSW$ is not complete for partially ordered CHiPs

algorithm is $O(n^2c^2)$ since the $O(nc)$ procedures for determining must/may-clobber must be run for each of nc conditions (c summary conditions in each of n plans represented by P_{sum}).

Soundness and completeness proofs of *CAW* and *MSW* are in Appendix D. For a partial ordering of plans in P_{sum}, *CAW* is sound and complete, but determining that *MSW* is false is only sound. This is because the algorithm for determining must-clobber only considers pairs of summary conditions in conflict and ignores cases where a plan that can come between the clobberer and requirer may also be clobbered. For example consider the partial ordering of plans in Figure 5.2. Plan p with a *must*, $\neg\ell$ summary postcondition is ordered before both p' and p'' with *must* ℓ summary pre-, in-, and postconditions, and p' and p'' are unordered with respect to each other. In this case, the algorithm for must-clobber returns $false$ because either p' or p'' can achieve the summary preconditions before p can clobber either of them. Thus, the algorithm for determining $\neg MSW$ returns false. However, in no histories can any set of plans with these summary conditions execute successfully, so $\neg MSW$ is true, and the algorithm is not complete. In order for the algorithm for $\neg MSW$ to be complete, it must assume that the plans are totally ordered (but potentially overlapping).[4] Therefore, I push the responsibility of detecting $\neg MSW$ for partially ordered plans onto the planner/coordinator that can search the possible synchronizations of the plans. The planner/coordinator can still use the $\neg MSW$ algorithm at each search state to discover unresolvable conflicts and backtrack. Section 6.1 details this approach. The algorithm for determining *CAW* is sound and complete for partially ordered plans because it is sufficient to only search for conflicts among pairs of summary conditions among the plans.

[4] A total ordering is one where each endpoint of a plan's execution interval is constrained to one temporal relation (precedes, follows, or same) with every other endpoint of every other plan's execution.

5.2 Summary Resource Usage Conflicts

Ignoring summary resource usage for a moment, resource conflicts are detected in different ways depending on the type of planner. If the planner reasons about partially ordered actions, it must consider which combinations of actions can overlap and exceed (or fall below) the resource's maximum capacity (or minimum value). A polynomial algorithm for doing this for the IxTeT planner is described in [Laborie and Ghallab, 1995]. Other planners that consider total order plans can more simply project the levels of the resource from the initial state through the plan to see if there are conflicts. This is done by applying the usages summed for concurrent actions for each subinterval between start and end times to the resource level and propagating depletions (or restorations) from the previous subinterval to the following one. ASPEN [Chien *et al.*, 2000b] is an example of a planner that does this.

Finding conflicts involving summarized resource usages can work in the same way. For the partial order planner, the resultant usage of clusters of actions are tested using the PARALLEL-AND algorithm in Section 4.4. For the total order planner, the level of the resource is represented as a summarized usage initially $\langle [x, x], [x, x], [x, x] \rangle$ for a depletable resource with an initial level x and $\langle [x, x], [x, x], [0, 0] \rangle$ for a non-depletable resource. Then, for each subinterval between start and end times of the schedule of tasks, the summary usage for each is computed using the PARALLEL-AND algorithm. Then the level of the resource is computed for each subinterval while propagating persistent usages using the SERIAL-AND algorithm.

5.3 Summary of Foundations

This chapter explained how threats can be detected for summary information using the algorithms developed in Chapter 4. The basic threat detection algorithms for summary conditions determine whether $CanAnyWay(order, P_{sum})$ and $MightSomeWay(order, P_{sum})$ properties hold for the summary information (P_{sum}) for a set of plans under a set of partial temporal ordering constraints (*order*). Informally, *CanAnyWay* (abbreviated *CAW*) is true if the plans can be decomposed and ordered in any way (while respecting the constraints in *order*) successfully. *MightSomeWay* (*MSW*) is true if there might be some successful decomposition and ordering for the tasks. A threat exists if *CAW* is false. The threat is unresolvable under *ordering* if *MSW* is false—in other words, there is no way to decompose and order the tasks consistently. *CAW* is proven sound and complete. The algorithm that determines if *MSW* is false is sound but only complete when *ordering* is a complete ordering (i.e. there are no disjuncts, or partially specified constraints, inferable from *ordering*). The *MSW* algorithm can be used to identify all threats and any unresolvable threats for summary conditions.

For summarized metric resources, we point to the IxTeT partial-order planner that uses graph techniques to efficiently detect and resolve resource conflicts [Laborie and Ghallab, 1995] for refinement-based planning. Our algorithms for determining may-clobber relationships can be integrated into this algorithm in a straightforward way to detect conflicts among abstract tasks. Local search planners such as ASPEN [Chien *et al.*, 2000b] efficiently detect and resolve metric resources by tracking resource levels over a time horizon by computing local and persistent usages of tasks with grounded start and end times. The component algorithms for deriving summarized metric resource usage can be substituted into this computation for determining a summarized state of the resource level over time to similarly detect resource conflicts.

In summary, Part I of this dissertation formalizes the concurrent execution of CHiPs (concurrent hierarchical plans), specifies the properties of summary conditions and summary metric resource usage, and introduces algorithms that derive this summary information, that determine many different interactions (such as clobbering and achieving) between tasks and their constraints, and that identify threats among partially expanded plans that can determine that either

- the plans are threat-free;
- the plans have unresolvable conflicts; or
- the plans may potentially be decomposed and ordered to resolve the threats.

These formalisms and algorithms are the foundations for the rest of this dissertation. The extensive formalization is necessary to ensure that agents reason clearly and correctly about their abstract plans in the context of others' plans while inferring as much about the potential and definite interactions of these tasks as possible.

In doing this, system designers can be assured that the agents will make sound coordination decisions and that they will reach these decisions efficiently at the highest level of abstraction possible without introducing irrelevant details in the decompositions of these tasks. This chapter described how agents can identify threats among their partially expanded plans hierarchies but makes no mention of how they should decompose their hierarchies and resolve their conflicts.

That is the purpose of the next two parts of this dissertation. Part II describes and evaluates a multiagent coordination algorithm based on the foundations of summary conditions built in this first part of the thesis. Part III describes how this algorithm can be adapted for single-agent planning and how summarized state and metric resource information is efficiently used in a local search planner. Part IV summarizes contributions and results and describes future research directions.

PART II

Multiagent Coordination

CHAPTER 6

Coordination Algorithm and Analyses

With the earlier defined algorithms for reasoning about a group of agents' plans at multiple levels of abstraction, I now describe how agents can efficiently coordinate based on summary information. I describe a coordination algorithm that searches for ways to restrict the decomposition and ordering of the collective actions of the agents in order to resolve conflicts while maximizing the utilities of the individual agents or the global utility of the group.

My approach to coordinating agents at multiple levels of abstraction starts by trying to coordinate at the most abstract level and, as needed, decomposes the agents' plans in a top-down fashion. The idea is that agents should not divulge any more information than is needed. Introducing irrelevant details only increases communication and complicates the coordination. After describing the top-down coordination algorithm, this chapter describes search techniques and heuristics that the algorithm can use to further exploit summary information. Then, analyses describe how coordinating at multiple levels can exponentially reduce search and maximize the performance of combined search and execution.

6.1 Coordinating from the Top Down

The formalism of summary conditions in Part I culminated in algorithms determining if a set of plans (abstract or primitive) under a partial set of ordering constraints is definitely conflict-free (*CanAnyWay*) or has unresolvable conflicts ($\neg$*MightSomeWay*). Here I integrate the *CanAnyWay* and *MightSomeWay* procedures into an algorithm that searches for a consistent coordinated plan for a group of agents. The particular algorithm I describe here is sound and complete. The search starts out with the top-level plans of each agent, which together represent the coordinated plan. The algorithm tries to find a solution at this level and then expands the hierarchies deeper and deeper until the optimal solution is found or the search space has been exhausted. A pseudocode description of the algorithm is given in Figures 6.1 and 6.2.

A state of the search is a partially elaborated plan that we represent as a set of *and* plans (one for each agent), a set of temporal constraints, and a set of blocked plans. The subplans of the *and* plans are the leaves of the partially expanded hierarchies of the agents. The set of temporal constraints includes synchronization constraints added during the search in addition to those dictated by the agents' individual hierarchical plans. Blocked subplans keep track of pruned *or* subplans. Decisions made during the search can be made decentrally. The agents can negotiate over ordering constraints to impose, choices of subplans to accomplish higher level plans, and which decompositions to explore first. While the algorithm described here does not comment on specific negotiation techniques, it does provide the mechanisms for identifying the choices over which the agents can negotiate. Although search decisions can be made decentrally, the algorithm given here is described as a centralized process that requests summary information from the agents being coordinated.

The operators of the search are expanding non-primitive plans, blocking *or* subplans, and adding temporal constraints on pairs of plans. When an agent expands one of its plans, the plan's summary conditions are replaced with only the original conditions of the parent plan. Then the subplans with their summary conditions are added to the search state, and the ordering information is updated in the coordinated plan. A subplan of an *or* plan is added only when all other subplans are blocked. Blocking an *or* subplan can be effective in resolving a constraint in which the other *or* subplans are not involved. For example, if the inventory manager plans to only use transport2, the production manager could block subplans using transport2 leaving subplans using transport1 that do not conflict with the inventory manager's plan. This can lead to least commitment abstract solutions that leave the agents flexibility in selecting among the multiple applicable remaining subplans. The agents can take another approach by selecting subplans (effectively blocking all of the others) to investigate choices that are given greater preference or are more likely to resolve conflicts.

In the pseudocode in Figure 6.1, the coordinating agent collects summary information about the other agents' plans as it decomposes them. The *queue* keeps track of expanded search states. If the *CanAnyWay* relation holds for the search state, the `Dominates` function determines if the current solutions are better for every agent than the solution represented by the current search state and keeps it if the solution is not dominated. If *MightSomeWay* is false, then the search space represented by the current search state can be pruned; otherwise, the operators mentioned above are applied to generate new search states (shown in Figure 6.2). Nondeterministic `Choose` functions determine how these operators are applied. The implementation of the algorithm uses heuristics specified in Section 6.2 to determine what choices are made. When a plan is expanded or selected (by the `Decompose` function shown in Figure 6.3), the ordering constraints must be updated for the subplans that are added. The `UpdateOrder` function accomplishes this.

Adding temporal constraints should only generate new search nodes when the ordering is consistent with the other global and local constraints. In essence, this operator performs the work of merging non-hierarchical plans since it is used to find a synchronization of the individual agents' plans that are one level deep. In the pseudocode, the `ChooseConstraint` function nondeterministically investigates all orderings (represented by point algebra constraints over the `Start` and `End` points of action intervals), and inconsistent ordering constraints are pruned using a point algebra table as discussed in Section 4.3.1. However, in the implementation, I only investigate legal ordering constraints that resolve threats that are identified by algorithms determining must/may achieves and clobbers relations among CHiPs. (By choosing only these constraints, fewer search states are generated, and the search more directly resolves conflicts.) In the experiments, the search for synchronizations is separated from the expansion and selection of subplans. An outer search is used to explore the space of plans at different levels of abstraction. For each state in the outer search, an inner search explores the space of plan merges by resolving threats with ordering constraints.

The soundness and completeness of the coordination algorithm depends on the soundness and completeness of identifying solutions and the complete exploration of the search space. Soundness and completeness is not defined with respect to accomplishing particular goals but resolving conflicts in the plan hierarchies. While goals can be represented

```
Concurrent Hierarchical Coordination Algorithm
Input:  set of agents
Output:  set of solutions
begin function
  plans = ∅
  for each agent a_i
     p_i = get summary information for a_i's top-level plan
     plans = plans ∪ {p_i}
  end for
  queue = {(plans, ∅, ∅)}
  solutions = ∅
  loop
     if queue == ∅
        return solutions
     end if
     (plans, order, blocked) = Pop(queue)
     if CanAnyWay(initial_state, plans, order, blocked)
        solution = (plans, order, blocked)
        solutions = solutions ∪ {solution}
        for each sol_1 and sol_2 in solutions
           if Dominates(sol_1, sol_2)
              solutions = solutions - {sol_2}
           end if
        end for
     end if
     if MightSomeWay(initial_state, plans, order, blocked)
        operator = Choose({expand, select, block, constrain})
        ApplyOperator(operator, plans, order, blocked, queue) (see Figure 6.2)
     end if
  end loop
  return solutions
end function
```

Figure 6.1: A concurrent hierarchical coordination algorithm.

```
ApplyOperator Function
Input: operator, plans, order, blocked, queue
Output: queue
begin function
  if operator == expand
     plan = ChooseAndPlan(plans)
     subplans = get summary information for subplans of plan
     Decompose(plans, plan, subplans, order) (Figure 6.3)
  else if operator == select
     plan = ChooseOrPlan(plans)
     plan.subplans = get summary information for subplans of plan
     for each subplan ∈ plan.subplans
        newblocked = blocked ∪ plan.subplans - {subplan}
        neworder = order
        newplans = plans
        Decompose(newplans, plan, {subplan}, neworder) (Figure 6.3)
        InsertStateInQueue(queue, newplans, neworder, newblocked)
     end for
  else if operator == block
     plan = ChooseOrPlan(plans)
     subplans = get summary information for subplans of plan
     for each subplan ∈ subplans where subplan ∉ blocked
        newblocked = blocked ∪ subplan
        neworder = order
        newplans = plans
        if ∃! subplan' ∈ plan.subplans, subplan' ∉ blocked
           Decompose(newplans, plan, {subplan'}, neworder) (Figure 6.3)
        end if
        InsertStateInQueue(queue, newplans, neworder, newblocked)
     end for
  else if operator == constrain
     plan = ChoosePlan(plans)
     plan' = ChoosePlan(plans - {plan})
     constraint = ChooseConstraint({Start, End} × {<, ≤, =, ≥, >} × {Start, End})
     neworder = order ∪ (plan, plan', constraint)
     if Consistent(neworder)
        InsertStateInQueue(queue, plans, neworder, blocked)
     end if
  end if
end function
```

Figure 6.2: ApplyOperator subprocedure for expanding a search state.

```
Decompose Function
Input:  plans, plan, subplans, order
Output:  plans, order
begin function
  Replace summary conditions of plan with original conditions
  plans = plans ∪ subplans
  UpdateOrder(order, plan, subplans, plan.order)
end function
```

Figure 6.3: `Decompose` subprocedure of `ApplyOperator()`.

as abstract CHiPs that decompose into possible plans that accomplish them, the goals of the planner/coordinator may be only to execute a series of actions successfully. Each search state is tested by the *CanAnyWay* procedure to determine whether it is a solution. The *CanAnyWay* procedure is shown to be sound and complete in Appendix D. Although the algorithm for determining $\neg MightSomeWay$ is only complete for a total ordering of CHiPs, it is used to prune invalid branches in the search space, so it is enough that it is sound (proof in Appendix D). In order to explore the search space completely, the coordinator would need to consider all synchronizations of all possible decompositions of each of the agents' top-level plans. We assume that the plan hierarchy of each agent is finite in its decomposition, so when the coordinator nondeterministically expands abstract plans, eventually all abstract plans will be replaced with primitive decompositions. Likewise, eventually all *or* plans will be replaced with subplan choices, and since new search states are generated and added to the queue for each subplan of an *or* plan, all possible combined decompositions of the agents' top-level plans are explored. The `Choose` function for selecting operators nondeterministically explores any synchronization of the expanded plans in conjunction with the `ChooseConstraint` function, so the search is complete. Note that this search assumes that each agent's hierarchical plan is internally consistent. This means that the hierarchies have the *CanAnyWay* property. If they did not, then the agents would also need to do planning to resolve internal conflicts. A planning algorithm is given in Chapter 8 that is a modification of the coordination algorithm presented here.

Consider how the algorithm would find coordinated plans for the manufacturing agents. At the beginning of the search, a coordinating agent gathers the summary information for the top-level plans of the three agents in *plans*. At first, there are no ordering constraints, so *order* is empty in the first search state (shown in Figure 5.1a) popped from the *queue*. *CanAnyWay* is false, and *MightSomeWay* is true for this state as described earlier in this section, so the coordinator chooses an *operator* to apply to the search state. It could choose `constrain` and order the *maintenance* plan before *produce_H* to resolve all conflicts between those two plans. So, $plan = maintenance$, $plan' = produce\_G\&H$, and $constraint =$(`End`, $<$, `Start`). The *order* is updated with the new constraint, and the new search state is inserted into the *queue* by the `InsertStateInQueue` function that determines how search states are ordered in the *queue*. On the next iteration of the `loop`, the only search state in the queue that was just inserted is popped. The coordinator again finds that *CanAnyWay* is false, and *MightSomeWay* is true since *move_parts* can still conflict with other plans over the use of transports. It can choose to constrain *produce_H* before *move_parts* to resolve the remaining conflicts. This is detected on the next cycle of the search `loop` where *CanAnyWay* is found to be true for this search state (shown in Figure 5.1a). The *plans*, the two constraints in *order*, and the empty set of blocked plans are added as a solution since there is no previously found solution that `Dominates` it. The `Dominates` function uses domain specific criteria for determining when a solution has value as an alternative and should be kept or is inferior compared to another and should be dropped. In this manufacturing domain, one solution dominates another if the finish time for at least one agent is earlier and none are later. The search then continues to find alternative or superior solutions, although the agents may wish to terminate the search in the interest of time.

Another coordination solution to this problem from the manufacturing domain is shown in Figure 5.1c. To generate this solution search state, the coordinator could first choose to `expand` *produce_H* (an *and* plan) into *produce_G* and $produce\_H\_from\_G$. These subplans are added to *plans*; *produce_H* is removed from *plans*; and the ordering constraint that *produce_G* is before $produce\_H\_from\_G$ is added to *order*. Then, with this search state, the coordinator can choose *select* on the *or* plan *maintenance* to create two new search states: one where *maintenance* decomposes into $service\_M1\_M2$ and one where *maintenance* decomposes into $service\_M2\_M1$. The first search state is created by adding $service\_M2\_M1$ to the *blocked* set and replacing *maintenance* with $service\_M1\_M2$ in *plans*. When this new search state is popped from the queue, the coordinator can then expand $service\_M1\_M2$ into $service\_M1$ and $service\_M2$. To arrive at the solution, the coordinator then adds the ordering constraints as shown by the arrows in Figure 5.1c in successive iterations through the search `loop`.

6.2 Search Techniques and Heuristics

Although summary information is valuable for finding conflict free or coordinated plans at abstract levels, this information can also be valuable in directing the search to avoid branches in the search space that lead to inconsistent or suboptimal coordinated plans. Inconsistent coordinated plans can be pruned away at the abstract level by doing a quick check to see if *MightSomeWay* is false. In terms of the number of states expanded during the search, employing this technique will always do at least as well as not using it, and pruning the search space at abstract levels greatly reduces search effort. For example, if the search somehow reached the state shown in Figure 4.2b, the coordinator could backtrack before expanding the hierarchies further and avoid reasoning about details of the plans where they must fail.

Another strategy is to first expand plans involved in the most threats. For the sake of completeness, the order of plan expansions does not matter as long as they are all expanded at some point when the search trail cannot be pruned. But, employing this "expand on most threats first" (EMTF) heuristic aims at driving the search down through the hierarchy to find the subplan(s) causing conflicts with others so that they can be resolved more quickly. This is similar to a most-constrained variable heuristic often employed in constraint satisfaction problems. For example, if the facilities and inventory managers wished to execute their plans concurrently as shown in Figure 6.4a, at the most abstract level, the coordinator would find that there are conflicts over the use of transports for moving parts. Instead of decomposing *produce_H* and reasoning about plan details where there are no conflicts, the EMTF heuristic would choose to decompose either *maintenance* or *move_parts* which have an equal number of conflicts. By decomposing *maintenance* the agents can resolve the remaining conflicts and still execute concurrently.

Another heuristic that a coordinator can use in parallel with EMTF is "fewest threats first" (FTF). Here the search orders nodes in the search queue by ascending numbers of threats left to resolve. In effect, this is a least-constraining value heuristic used in constraint satisfaction approaches. As mentioned in Section 5.1, threats are identified

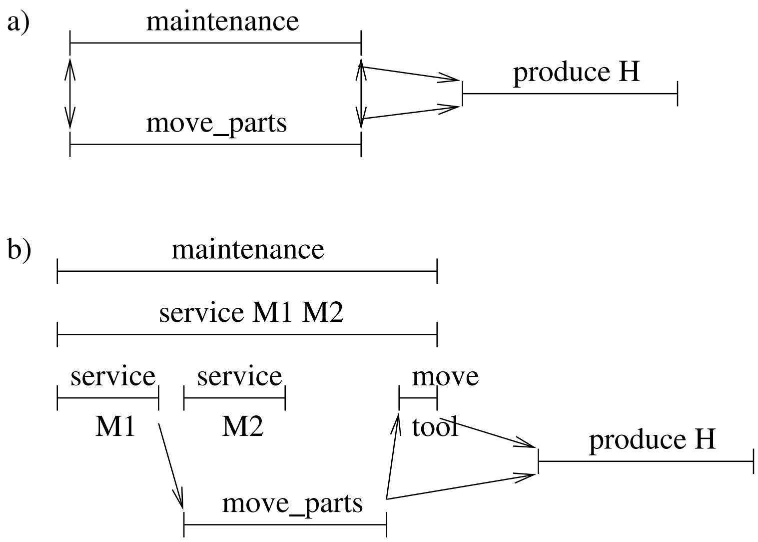


Figure 6.4: EMTF heuristic resolving conflicts by decomposing the *maintenance* plan

by the $\neg$*MightSomeWay* algorithm. By trying to resolve the threats of coordinated plan search states with fewer conflicts, it is hoped that solutions can be found more quickly. So, EMTF is a heuristic for ordering *and* subplans to expand, and FTF, in effect, orders *or* subplan choices. For example, the production manager has a choice of using machine M1 or M2 to produce part G. In decomposing *produce_G*, the coordinator creates two search states for these two choices: one where *produce_G* is replaced by *produce_G_on_M*1 and one that selects *produce_G_on_M*2. If these plans overlap with the inventory manager's *service_M*1 plan, the search state with *produce_G_on_M*2 will have fewer conflicts since there will be no conflicts over the use of machine M1. Therefore, by first searching the state with *produce_G_on_M*2 selected as dictated by the FTF heuristic, the coordinator is likely closer to finding a solution because there are fewer conflicts left to resolve. This heuristic can be applied not only to selecting *or* subplan choices but also to choosing temporal constraints and variable bindings or even the set of all possible search state expansions.

In addition, in trying to find optimal solutions in the style of a branch-and-bound search, I use the cost of abstract solutions to prune away branches of the search space whose minimum cost is greater than the maximum cost of the current best solution. This is the role of the `Dominates` function in the description of the coordination algorithm in Section 6.1. This usually assumes that cost/utility information is decomposable over the hierarchy of actions. This means that the cost of any abstract action is a function of its decompositions.

For example, if the coordinator is trying to minimize the makespan (duration) of each of the agents, each plan's summary information can keep track of the minimum and maximum makespans for its possible decompositions. During the computation of summary information, these ranges can be computed for an abstract plan as follows. If a plan is an *and* plan, the planner can simulate the execution of its subplans (without decomposing them) according to the ordering constraints among them starting each subplan as soon as the constraints allow and keep track of the time passed. Simulating with the subplans' minimum makespans as their durations gives the minimum makespan of the parent, and simulating with the maximum makespans of the subplans computes the maximum makespan of the parent. If it is an *or* plan, its minimum makespan is simply the minimum of its subplans' minimum makespans, and the maximum makespan is the maximum of the subplans' maximums. The coordinator can use the *and* plan simulation to estimate the minimum and maximum makespans of each agent's plan in a search state by considering the state's set of plans and their constraints as the decomposition of a parent representing the coordinated plan. In this way, the coordinator can evaluate the cost of different options during search and prune those whose minimum makespans are greater than those of an already found solution. This computation of cost is used in this work's implementations.

Horty and Pollack have developed a theory for estimating the collective costs of combinations of actions within a context in order to evaluate plan choices [Horty and Pollack, 2001]. A domain expert can apply this research to develop more complex mechanisms for deriving summarized costs for abstract plans and for determining when a search state or solution is dominated by another. The branch-and-bound pruning technique can be used without summarized costs, but then only solutions at the primitive level can be used to prune search states at the primitive level. Again, pruning abstract plans can only help improve the search and can do so greatly. I report experimental results in Chapter 7 that show that the techniques and heuristics reported in this section can greatly improve coordination performance.

6.3 Coordination Performance and Complexity

While the coordinator can use the search techniques described in the Section 6.2 to prune the search space, simply being able to find solutions at multiple levels of abstraction can reduce the combined computation and execution cost. In this section I give an example of this and then analyze the complexity of coordination to characterize this cost reduction and the conditions under which it occurs.

6.3.1 Improving the Performance of Search and Execution

An agent that interleaves execution with planning/coordination often must limit the total computation and execution cost required to achieve its goals. The coordinator described in Section 6.1 is able to search for solutions at different levels of abstraction. While less effort is needed to find a solution at a high level of abstraction, better plans (with lower cost) may be found at lower levels. For example, if the cost of execution depends on the time to complete execution and if an agent must react quickly to the current situation, it may not have much time to deliberate about its plan before acting. In this case, it is important to reduce the total time for the agent to both plan and execute its actions.

For the manufacturing example, the coordinator finds a solution at the top level of the agents' plans as shown in Figure 5.1b and described in Section 6.1. For the implementation, the coordinator takes 1.9 CPU seconds to find this solution. If we define the cost of execution as the makespan (completion time) of the coordinated plan, the cost of this solution is 210 where the makespan of the production manager's plan is 90, the facilities manager's is 90, and the inventory manager's is 30. For the solution in Figure 5.1c, the implementation required 667 CPU seconds, and the makespan of the coordinated plan is 170. Another solution is found at an intermediate level of abstraction, taking 69 CPU seconds and having a makespan of 180. So, with a little more coordination effort, the coordinator expanded the hierarchy to an intermediate level where the cost of the solution was reduced by 30. By digging a little deeper into the hierarchy and spending a lot more effort, the coordinator was able to further reduce the cost of the solution by 10.

Figure 6.5 shows how the total cost of computation and execution (y-axis) for the three solutions varies with different ratios of the computation unit cost with execution unit

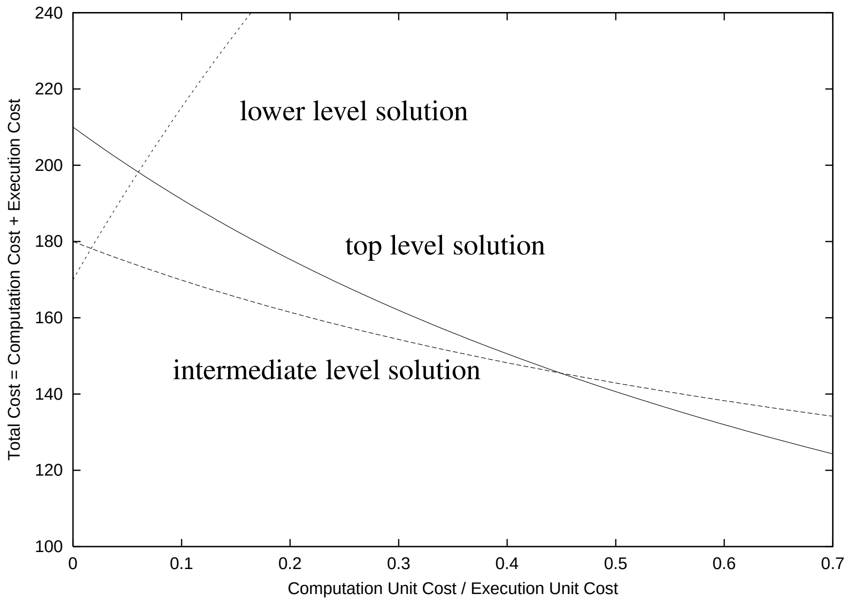


Figure 6.5: Tradeoffs of computation and execution costs

cost (x-axis). The total cost for each solution is plotted as $w_c \times computation\_cost + w_e \times execution\_cost$ where w_c and w_e are relative unit costs such that $w_c + w_e = 1$, and the x-axis is the ratio w_c/w_e. Therefore, when the ratio is zero, the total cost is the computation cost, and as the ratio approaches infinity, the total cost approaches the execution cost. By order of increasing slope the curves correspond to the top-level solution, the intermediate level solution, and the lower level solution. When this ratio is 0.6 (1 unit of execution cost is equal to 0.6 units of computation cost), the total cost is lowest when stopping the search after the top-level solution is found and then executing. If the computation cost is insignificant (the ratio is close to zero), then it is better to allow the coordinator to find the lower cost solution before executing. If the ratio of the costs is 0.3, then using the intermediate solution gives the best performance.

Thus, depending how costly execution is compared to computation, the depth to which coordinator should search varies. Without the ability to find solutions at abstract levels, the coordinator would be forced to search at the bottom (primitive) level of the hierarchies where the problem could be intractable. So, the goal of the coordinator is to search until the cost of finding the next best solution is too expensive to continue. While it is not always possible to predict the execution cost of a solution at any particular level, the next section describes the worst case complexity of finding a solution at a particular level that can be used to estimate computation costs.

6.3.2 Complexity of Summarization and Finding Abstract Solutions

In the previous section, anecdotal evidence was given to show that coordinating at higher levels of abstraction is less costly because there are fewer plan steps. But, even though there are fewer plans at higher levels, those plans may have greater numbers of summary conditions to reason about because they are collected from the much greater set of plans below. Here I argue that even in the worst case where the number of summary conditions per plan increases exponentially up the hierarchy, finding solutions at abstract levels is expected to be exponentially cheaper than at lower levels. This analysis shares some similarity to those of others that show that hierarchical problem solving, under certain restrictions, can reduce the size of the search space by an exponential factor [Korf, 1987; Knoblock, 1991]. I show the potential for even greater speedups without these restrictions. This discussion is saved for the end of the section. I first analyze the complexity of the summarization algorithm to help the reader understand how the summary conditions can collect in greater sets at higher levels.

Consider a hierarchy with n total plans, b subplans for each non-primitive plan, and depth d as shown in Figure 6.6.[1] The procedure for deriving summary conditions works by basically propagating the conditions from the primitives up the hierarchy to the most abstract plans. Because the conditions of any non-primitive plan depend only on those of its immediate subplans, deriving summary conditions can be done quickly. The summary information algorithm mainly involves checking for achieve and undo interactions between subplans (as described in Section 4.1). Checking for one of these relations for one summary condition of one subplan is $O(bs)$ for b subplans, each with s summary conditions (as discussed in Section 4.3.4). Since there are $O(bs)$ conditions that must be checked in the set of subplans, deriving the summary information of one plan from its subplans is $O(b^2s^2)$.

However, the maximum number of summary conditions for a subplan grows exponentially up the hierarchy since, in the worst case, no summary conditions merge during summarization.[2] As shown in the third column of the table in Figure 6.6, a plan at the

[1] We consider the root at depth level 0 and the leaves at level d.

[2] This happens when the conditions of each plan are completely different than those of any other plan.

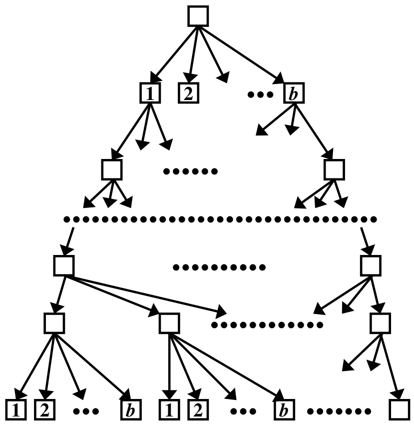
lowest level d has $s = c$ summary conditions derived from its c pre-, in-, and postconditions. A plan at level $d-1$ derives c summary conditions from its own conditions and c from each of its b subplans giving $c + bc$ summary conditions, or $s = O(bc)$. So, in this worst case $s = O(b^{d-i}c)$ for a plan at level i in a hierarchy for which each plan has c (non-summary) conditions. Thus, the complexity of summarizing a plan at level i (with subplans at level $i+1$) is $O(b^2 b^{2(d-(i+1))} c^2) = O(b^{2(d-i)} c^2)$. There are b^i plans at level i (second column in the figure), so the complexity of summarizing the set of plans at level i is $O(b^i b^{2(d-i)} c^2) = O(b^{2d-i} c^2)$ as shown in the fourth column in the figure. Thus, the complexity of summarizing the entire hierarchy of plans would be $O(\sum_{i=0}^{d-1} b^i b^{2(d-i)} c^2)$. In this summation level $i = 0$ dominates, and the complexity can be reduced to $O(b^{2d} c^2)$. If there are $n = O(b^d)$ plans in the hierarchy, we can write this simply as $O(n^2 c^2)$, which is the square of the size of the hierarchy.

In order to resolve conflicts (and potentially arrive at a solution) at a particular level of expansion of the hierarchy, the coordination algorithm checks for threats between the plans under particular ordering constraints at that level. Checking for threats involves finding clobber relations among the plans and their summary conditions. The complexity of finding threats among n plans each with s summary conditions is $O(n^2 s^2)$ as shown in Section 5.1 for the *MightSomeWay* algorithm. For a hierarchy expanded to level i, there are $n = O(b^i)$ plans at the frontier of expansion, and each plan has $s = O(b^{d-i}c)$ summary conditions. So, as shown in the fifth column of the table in Figure 6.6, the complexity of checking for threats for one synchronization of a set of plans at level i is $O(b^2 i (b^{d-i}c)^2) = O(b^{2d} c^2)$. Notice that i drops out of the formula, meaning that the complexity of checking a candidate solution is *independent of the depth level*.

However, the algorithm may check many synchronizations at a particular level before finding a solution or exhausting the search space. In fact this search complexity grows exponentially with the number of plans. As shown in the last column of the table in Figure 6.6, the search space is $k^n = O(k^{b^i})$ for n plans at level i and constant k.[3] Thus, the search space grows doubly exponentially down the hierarchy despite the worst case when the number of conditions grows exponentially up the hierarchy. I formally show that finding

In this case, a separate summary condition will be generated for each summary condition of each subplan.

[3]This is why Georgeff chose to cluster multiple operators into "critical regions" and synchronize the (fewer) regions since there would be many fewer interleavings to check [Georgeff, 1983]. By exploiting the hierarchical structure of plans, I use the "clusters" predefined in the hierarchy to this kind of advantage without needing to cluster from the bottom up.



level	#plans	#conds / plan	#operations to derive summ. info.	#test operations / solution candidate	solution space
0	1	$O(b^d c)$	$O(b^2(b^{d-1}c)^2) = O(b^{2d}c^2)$	$O(1)$	1
1	b	$O(b^{d-1}c)$	$O(bb^2(b^{d-2}c)^2) = O(b^{2d-1}c^2)$	$O(b^2(b^{(d-1)}c)^2) = O(b^{2d}c^2)$	$O(k^b)$
2	b^2	$O(b^{d-2}c)$	$O(b^2b^2(b^{d-3}c)^2) = O(b^{2d-2}c^2)$	$O(b^4(b^{(d-2)}c)^2) = O(b^{2d}c^2)$	$O(k^{b^2})$
d-2	b^{d-2}	$O(b^2c)$	$O(b^{d-2}b^2(bc)^2) = O(b^{d+2}c^2)$	$O(b^{2(d-2)}(b^2c)^2) = O(b^{2d}c^2)$	$O(k^{b^{d-2}})$
d-1	b^{d-1}	$3c+b3c = O(bc)$	$O(b^{d-1}b^2c^2) = O(b^{d+1}c^2)$	$O(b^{2(d-1)}(bc)^2) = O(b^{2d}c^2)$	$O(k^{b^{d-1}})$
d	b^d	$3c$	$O(1)$	$O(b^{2d}c^2)$	$O(k^{b^d})$
i	b^i	$O(b^{d-i}c)$	$O(b^{2d-i}c^2)$	$O(b^{2d}c^2)$	$O(k^{b^i})$

Figure 6.6: Complexity of threat identification and resolution at abstract levels

a valid synchronization is intractable for larger numbers of plan steps by proving that it is actually NP-complete. In Appendix E, I reduce HAMILTONIAN PATH to the THREAT RESOLUTION problem for STRIPS planning and describe how a similar reduction can be done for the problem that allows concurrent execution.

There are only *and* plans in this worst case. In the case that there are *or* plans, by similar argument, being able to prune branches at higher levels based on summary information will greatly improve the search despite the overhead of deriving and using summary conditions. As will be discussed in Section 9.3, the computational savings of using summary information will be even greater when there are conditions common to plans on the same level, and the number of summary conditions does not grow exponentially up the hierarchy. Still, surely there are cases where none of the details of the plan hierarchy can be ignored, and summary information would incur unnecessary overhead, but when the size of problem instances are scaled, dealing with these details will likely be infeasible anyway.

As stated earlier, other complexity analyses have shown that, under certain restrictions, different forms of hierarchical problem solving can reduce the size of the search space by an exponential factor [Korf, 1987; Knoblock, 1991]. These restrictions are basically that the algorithm never needs to backtrack from lower levels to higher levels. Backtracking across abstraction levels occurs within the planner described in 6.1 when the current search state is $\neg MightSomeWay$ and another *or* subplan on the same or higher level can be selected. I demonstrated that the search space grows doubly exponentially down the hierarchy because the number of plans grows exponentially, and resolving conflicts grows exponentially with the number of plans. Thus, as long as the coordinator does not have to fully expand all abstract plans to the primitive level, the search complexity is reduced at least by a factor of $k^{b^d - b^i}$ where i is the level where the search completed, and d is the depth of the hierarchy. In Section 9.3 we show how further improvements can be made when summary information does collapse during summarization.

CHAPTER 7

Performance Experiments and Applications

The mechanisms for reasoning about plans of multiple agents at abstract levels in Part I and the coordination algorithm in Section 6.1 can apply to a wide range of applications involving interacting planning (or plan execution) agents. In support of this, in this chapter I describe the application of multi-level coordination to three different domains: the manufacturing domain described in the Introduction, an evacuation domain, and a military peace-keeping scenario. In these domains, performance is defined in different ways to show a range of benefits offered by abstract reasoning.

In Section 6.3, I analytically explained how agents can coordinate much more quickly at abstract levels than at the most detailed level. But, what if they cannot resolve conflicts at abstract levels? Or, what if the agents need to find optimally coordinated plans that may only exist at lower levels in the agents' hierarchies? In the latter case, performance is then measured more in terms of execution rather than coordination computation time. In the next section I report experiments for an evacuation domain that show how abstract reasoning using summary information can find optimal coordination solutions much more quickly than conventional search strategies. Optimal solutions in the evacuation domain have minimal global execution times because evacuees must be transported to safety as quickly as possible.

In some domains, computation time may be insignificant to communication costs. These costs could be in terms of privacy for self-interested agents, security for sensitive information that could obtained by malicious agents, or simply communication delay. In Section 7.2, I show how multi-level coordination fails to reduce communication delay for the manufacturing domain example but, for other domains, can be expected to reduce communication overhead exponentially.

Then I describe how this technology is wrapped in an agent that continually coordinates coalitions of UN forces in maintaining peace between two warring factions in a fictional state of Africa, called Binni (Section 7.3). Performance for this domain is based on presenting Combat Operations with several options that tradeoff makespans (completion times) of different coalition forces.

7.1 Evacuation Experiments

In this section, I describe experiments that evaluate the use of summary information in coordinating a group of evacuation transports that must together retrieve evacuees from a number of locations with constraints on the routes. In comparing the EMTF and FTF search techniques described in Section 6.2 against conventional HTN approaches, the experiments show that reasoning about summary information finds optimally coordinated plans much more quickly than the prior HTN techniques.

I compare different techniques for ordering the expansion of subplans of both *and* and *or* plans to direct the decomposition of plan hierarchies in the search for optimal solutions. This corresponds to the application of *expand* (for *and* subplans) and *select* (for *or* subplans) operators of the algorithm described in Section 6.1. I compare EMTF's expansion of *and* plans to the ExCon heuristic and to a random selection heuristic. The ExCon heuristic [Tsuneto *et al.*, 1998] first selects plans that can achieve an external precondition, or if there are no such plans, it selects one that threatens the external precondition. In the case that there are neither achieving or threatening plans, it chooses randomly. Note that EMTF will additionally choose to expand plans with only threatened external preconditions but has no preference as to whether the plan achieves, threatens, or is threatened. For the expansion of *or* plans, I compare FTF to a depth-first (DFS) and a random heuristic. I also compare the combination of FTF and EMTF to an FAF heuristic and to the combination of DFS and ExCon. The FAF heuristic does not employ summary information but rather uses an FAF ("fewest alternatives first") heuristic [Currie and Tate, 1991; Tsuneto *et al.*, 1997] to decide both the order in which subplans are expanded and the order in which *or* subplans (and search states) are investigated. This simply means it chooses to expand *and* and select *or* plans that have the fewest subplans. Since no sum-

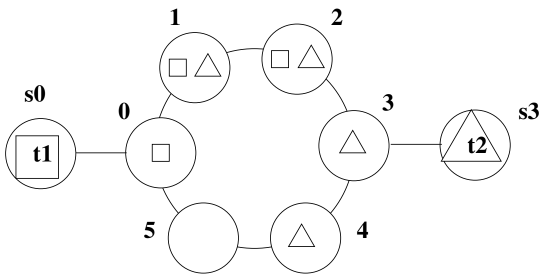


Figure 7.1: Evacuation problem

mary information is used, threats are only resolved at primitive levels. While it has been shown that the FAF heuristic can be effectively used by an HTN planner [Tsuneto *et al.*, 1997], the combination of DFS and ExCon has been shown to make great improvements over FAF in a domain with more task interactions [Tsuneto *et al.*, 1998]. I show in one such domain that the FTF and EMTF heuristics can together outperform combinations of FAF, DFS, and ExCon.

The problems were generated for an evacuation domain where transports are responsible for visiting certain locations along restricted routes to pick up evacuees and bring them back to safety points. To avoid the risk of oncoming danger (from a typhoon or enemy attack), the planner must ensure that transports avoid collisions along the single lane routes and accomplish their goals as quickly as possible. Transports are allowed to be at the same location at the same time.

Suppose there are two transports, $t1$ and $t2$, located at safety points $s0$ and $s3$ respectively, and they must visit the locations 0, 1, and 2 and 2, 3, and 4 respectively and bring evacuees back to safe locations as shown in Figure 7.1. Because of overlap in the locations they must visit, the planner must synchronize their actions in order to avoid collision. The planner's goal network includes two unordered tasks, one for each transport to $evacuate$ the locations for which it is responsible. As shown in Figure 7.2, the high-level task for $t1$ ($evacuate$) decomposes into a primitive action of moving to location 0 on the ring and an abstract plan to traverse the ring (*makerounds*). $t1$ can travel in one direction around the ring without switching directions, or it can switch directions once. $t1$ can then either go clockwise or counterclockwise and, if switching, can switch directions at any location ($first\ route$) and travel to the farthest location it needs to visit from where it switched ($second\ route$). Once it has visited all the locations, it continues around until it reaches the first safety point in its path (*go back* and $goto\ safe\ loc$). (The *no move* plan

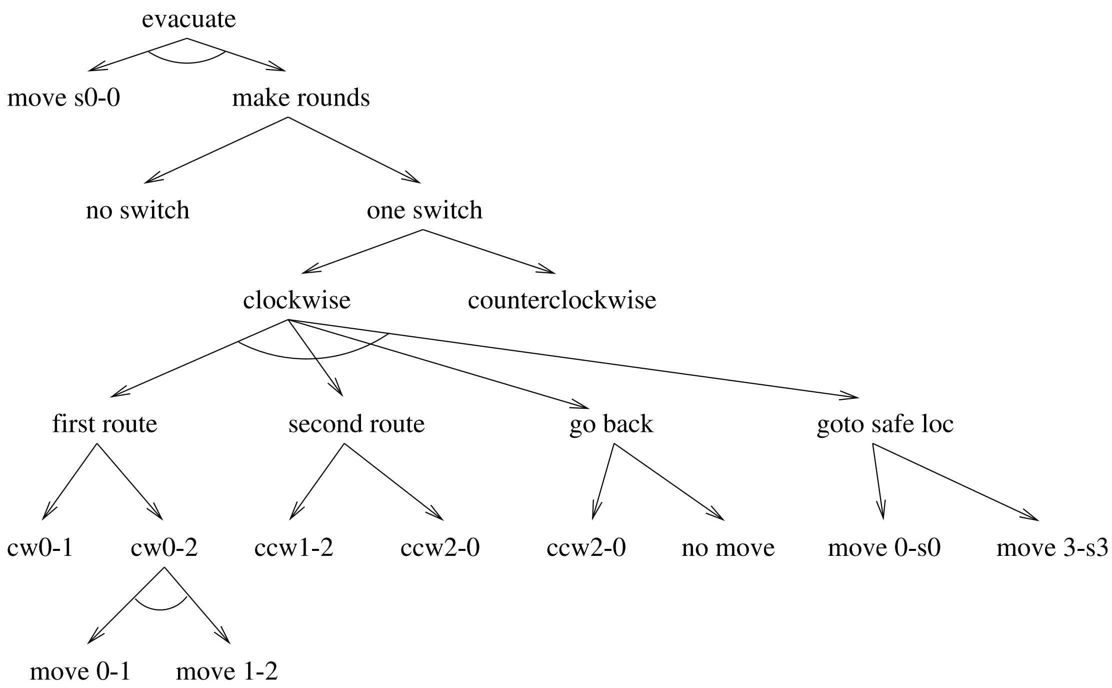


Figure 7.2: The plan hierarchy for transport $t1$

is for the case where $t1$ is already at location 0.) The task for $t2$ can be refined similarly.

Suppose the planner derives summary information for the plan hierarchy and attempts to resolve conflicts. Looking just at the summary information one level from the top, the planner can determine that if $t1$ finishes evacuating before $t2$ even begins, then there will be no conflicts since the external conditions of $t1$'s *evacuate* plan are that none of the routes are being traversed. This solution has a makespan (total completion time) of 16 steps. The optimal solution is a plan of duration seven where $t1$ moves clockwise until it reaches location $s3$, and $t2$ starts out clockwise, switches directions at location 4, and then winds up at $s0$. For this solution $t1$ waits at location 2 for one time step to avoid a collision on the route from location 2 to location 3.

A solution is simply a conflict-free plan. The cost of a solution is the makespan (completion time) of the coordinated plan for the transports where each move has a uniform time cost. Thus, an optimal plan has the minimum makespan. I generated 24 problems with four, six, eight, and twelve locations; with two, three and four transports; and with no, some, and complete overlap in the locations the transports visit. Performance was measured as the number of search nodes expanded to find the optimal solution or to find the *best solution of common quality* to the compared heuristics within memory and time

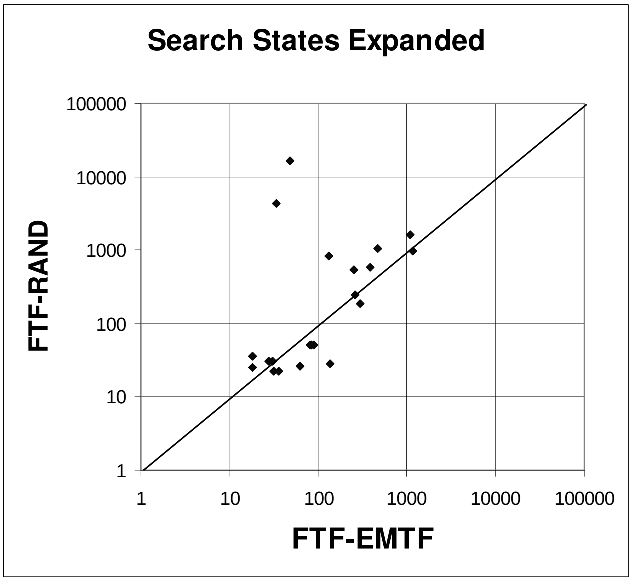


Figure 7.3: FTF-EMTF vs. FTF-RAND in searching for optimal solutions for 24 problems

bounds. We chose this instead of CPU time as the measure of performance in order to avoid fairness issues with respect to implementation details of the various approaches.

The scatter plot in Figure 7.3[1] shows the relative performance of the combination of FTF and EMTF (FTF-EMTF) and the combination of FTF and random *and* expansion (FTF-RAND). While performance is similar for most problems, there are a few cases where FTF-EMTF outperformed FTF-RAND by an order of magnitude or more. Figure 7.4 exhibits a similar effect for FTF-EMTF and FTF-ExCon.[2] While performance is similar for most problems, there are four points along the top[2] where FTF-ExCon finds no solution. Thus, although EMTF does not greatly improve performance for many problems, it rarely performs much worse, and almost always avoids getting stuck in fruitless areas of the search space compared to the ExCon and the random heuristic. This is to be expected since EMTF focuses on resolving conflicts among the most problematic plans first and avoids spending a lot of time reasoning about the details of less problematic plans.

The combination of FTF with EMTF, pruning inconsistent abstract plan spaces, and branch-and-bound pruning of more costly abstract plan spaces (all described in Section

[1]Note that for all scatter plots, the axes are scaled logarithmically.

[2]Runs were terminated after the expansion of 3,500 search states. Data points at 3,500 indicate that no solution was found within memory and time constraints.

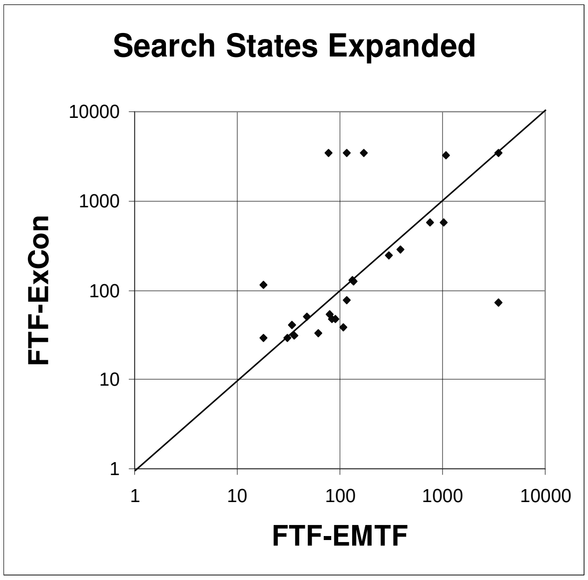


Figure 7.4: FTF-EMTF vs. FTF-ExCon in searching for optimal solutions

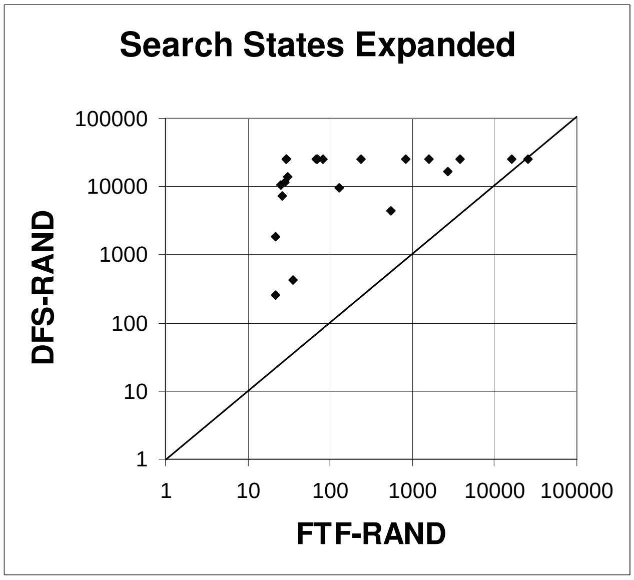


Figure 7.5: FTF-RAND vs. DFS-RAND in searching for optimal solutions

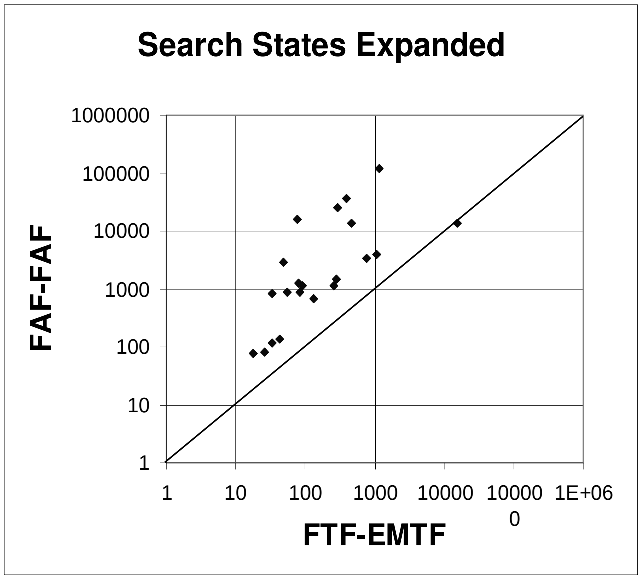


Figure 7.6: FTF-EMTF vs. FAF-FAF in searching for optimal solutions

6.2) outperforms techniques that do not reason at abstract levels much more dramatically.[3] Figure 7.5 shows DFS-RAND expanding between one and three orders of magnitude more states than FTF-RAND.[4] By avoiding search spaces with greater numbers conflicts, FTF finds optimal or near-optimal solutions much more quickly. In Figures 7.6[5] and 7.7[6], FTF-EMTF outperforms FAF-FAF (FAF for both selecting *and* and *or* plans) and DFS-ExCon by one to two orders of magnitude for most problems. These last two comparisons especially emphasize the importance of abstract reasoning for finding optimal solutions. Within a maximum of 3500 expanded search states (the lowest cutoff point in the experiments), FTF-EMTF and FTF-RAND found optimal solutions for 13 of the 24 problems. FTF-ExCon and FAF-FAF found 12; and DFS-ExCon and DFS-Rand only found three.

A surprising result is that FAF-FAF performs much better than DFS-ExCon for the evacuation problems contrary to the results in [Tsuneto *et al.*, 1998] that show DFS-ExCon dominating for problems with more goal interactions. I believe that this result

[3]The planners for the other techniques (DFS-RAND, FAF-FAF, and DFS-ExCon) do prune more costly *primitive-level* plan spaces.

[4]Runs were terminated after the expansion of 25,000 search states. Data points at 25,000 indicate that no solution was found within memory and time constraints.

[5]One of the 24 problems that neither could solve is omitted.

[6]Runs were terminated after the expansion of 3,500 search states. Data points at 3,500 indicate that no solution was found within memory and time constraints.

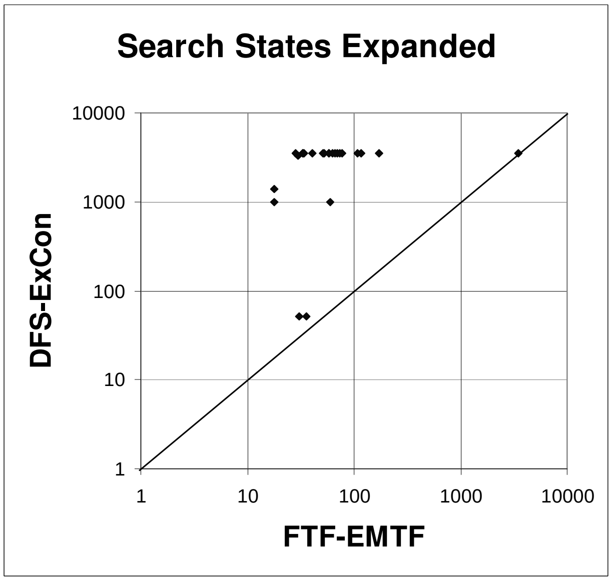


Figure 7.7: FTF-EMTF vs. DFS-ExCon in searching for optimal solutions

was not reproduced here because those experiments involved hierarchies with no *or* plans. The experiments show that the selection of *or* subplans more greatly affects performance than the order of *and* subplans to expand. So, I believe DFS-ExCon performed worse than FAF-FAF not because FAF is better at choosing *and* subplans than ExCon but because FAF is stronger at selecting *or* subplans than DFS.

However, the main point of this section is that each of the heuristic combinations that use summary information to find solutions and prune the search space at abstract levels (FTF-EMTF, FTF-ExCon, and FTF-RAND) greatly outperform all of those that do not (FAF-FAF, DFS-ExCon, and DFS-RAND) when searching for optimal solutions.

7.2 Manufacturing Experiments

Here I show that, depending on bandwidth, latency, and how summary information is communicated among the agents, delays due to communication overhead vary. If only communication costs are a concern, then in one extreme sending the plan hierarchy without summary information makes the most sense. In another extreme it makes sense to send the summary information for each task in a separate message as each is requested. Still, there are cases when sending the summary information for tasks in groups makes the most sense. This section will explain how a system designer can choose an appropriate level of granularity to send summary information that will reduce communication overhead exponentially.

Consider a simple protocol where agents request coordination from a central coordinating agent. During the search for a feasible solution, whenever it decomposes a task, the coordinator requests summary information for the subtasks that it has not yet received. For the manufacturing domain, the coordinator may already have summary information for a task to move a part, but if it encounters a different instantiation of the same task schema, it still must request the parameters for the new task. If a coordinator needs the subplans of an *or* plan, the client agent sends the required information for all subplans specifying its preferences for each. The coordinator then chooses the most preferred subplan, and in the case it must backtrack, it chooses the next most preferred subplan. Once the coordinator finds a feasible solution, modifications are sent to each agent specifying which *or* subplans are blocked and where it must send and wait for synchronization messages. An agent can alternatively send summary information for the whole plan hierarchy up front, for single tasks as they are requested, or for some number of levels of expansion of the requested task's hierarchy.

For the manufacturing problem described in Section 1.2, communication data in terms of numbers of messages and the size of each was collected up to the point that the coordinator found the solution in Figure 5.1c. This data was collected for cases where agents sent summary information for tasks in their hierarchies, one at a time, two levels at a time, and all at once. The two levels include the requested task and its immediate subplans. The following table below summarizes the numbers and total sizes of messages sent for each granularity level of information:

	number of messages	total size (bytes)
one at a time	9	8708
two at a time	4	10525
all at once	3	16268

Assuming that the coordinator must wait for requested information before continuing its search and can request only one task of one agent at a time, the coordination will be delayed for an amount of time depending on the bandwidth and latency of message passing. The total delay can be calculated as $(n-2)\ell + s/b$, where n is the number of

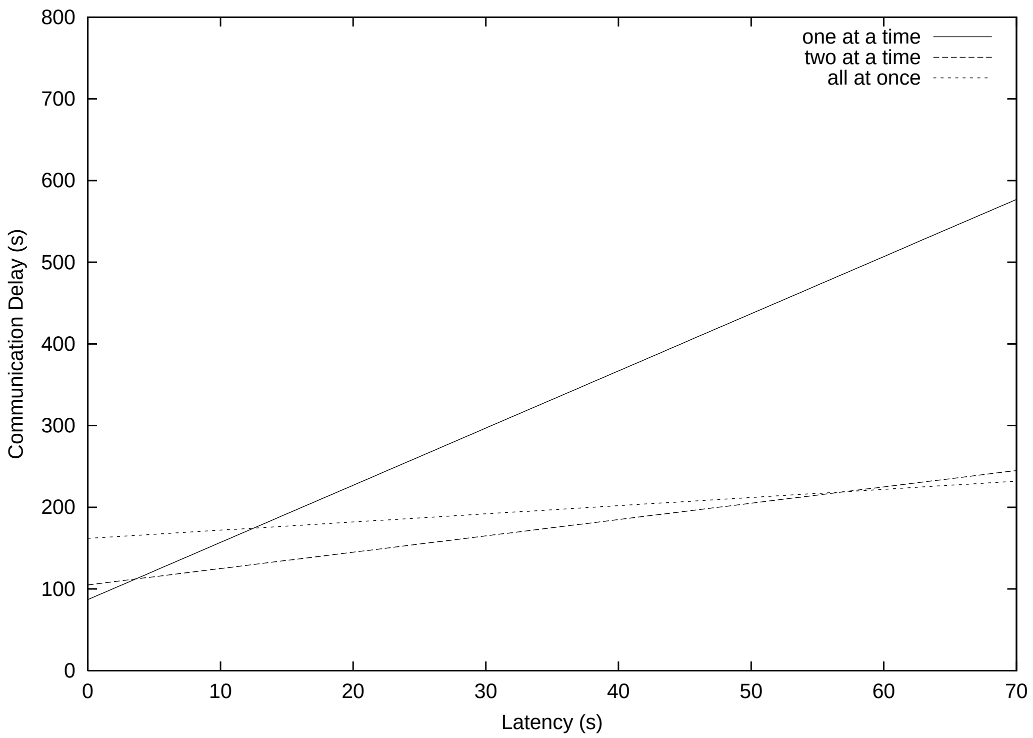


Figure 7.8: Delay of communicating different granularities of summary information with varying latency

messages sent; ℓ is the latency in seconds; s is the total size of all messages; and b is the bandwidth in bytes per second. We use $n-2$ instead of n because we assume that the agents all transmit their first top-level summary information message at the same time, so three messages actually only incur a delay of ℓ instead of 3ℓ.

Figure 7.8 shows how the communication delay varies for the three granularities of information for a fixed bandwidth of 100 bytes/second. When the latency is less than 3 seconds, sending summary information for each task in separate messages results in the smallest communication overhead. For latencies greater than 58 seconds, sending the entire hierarchy is best; and in between sending summary information two levels at a time is best. If the latency is fixed at 100 seconds, then the communication delay varies with bandwidth as shown in Figure 7.9. When the bandwidth is less than 3 bytes/second, sending one at a time is best; sending it all at once is best for bandwidths greater than 60 bytes/second; and sending two levels at a time is best for bandwidths in between.

Admittedly, these values for bandwidth and latency are unrealistic for the manufacturing domain. The manufacturing problem itself is very simple and provided mainly as

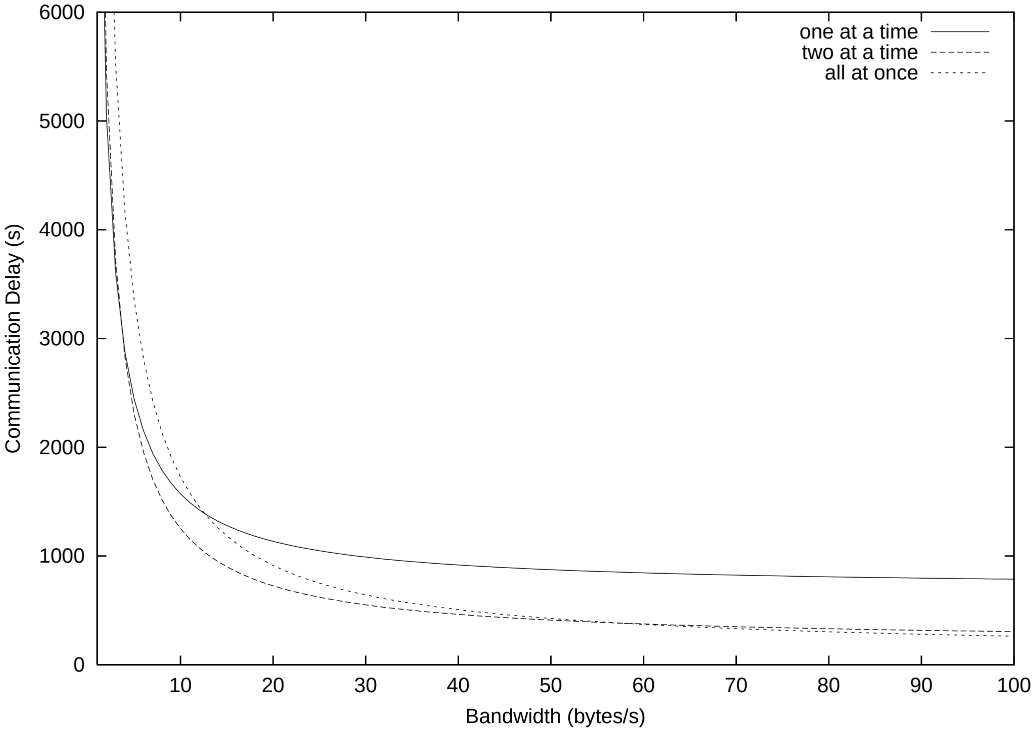


Figure 7.9: Delay of communicating different granularities of summary information with varying bandwidth.

an interesting domain for coordination. More realistic problems involving the manufacturing domain could have much larger hierarchies and require much larger scales of data to be sent. In that case more realistic bandwidth and latency values would exhibit similar tradeoffs.

To see this, suppose that the manufacturing managers' hierarchies had a common branching factor b and depth d. If tasks generally had reservations on similar resources throughout the hierarchies, the amount of summary information derived for the tasks at particular levels would grow exponentially down the hierarchy just as would the number of tasks. If the agents agreed on a feasible solution at depth level i in the hierarchy, then the table for messages and size would appear as follows:

	number of messages	total size
one at a time	$O(b^i)$	$O(b^i)$
two at a time	$3i/2$	$O(b^i)$
all at once	3	$O(b^d)$

Now suppose that the branching factor b is 3; the depth d is 10; the solution is found

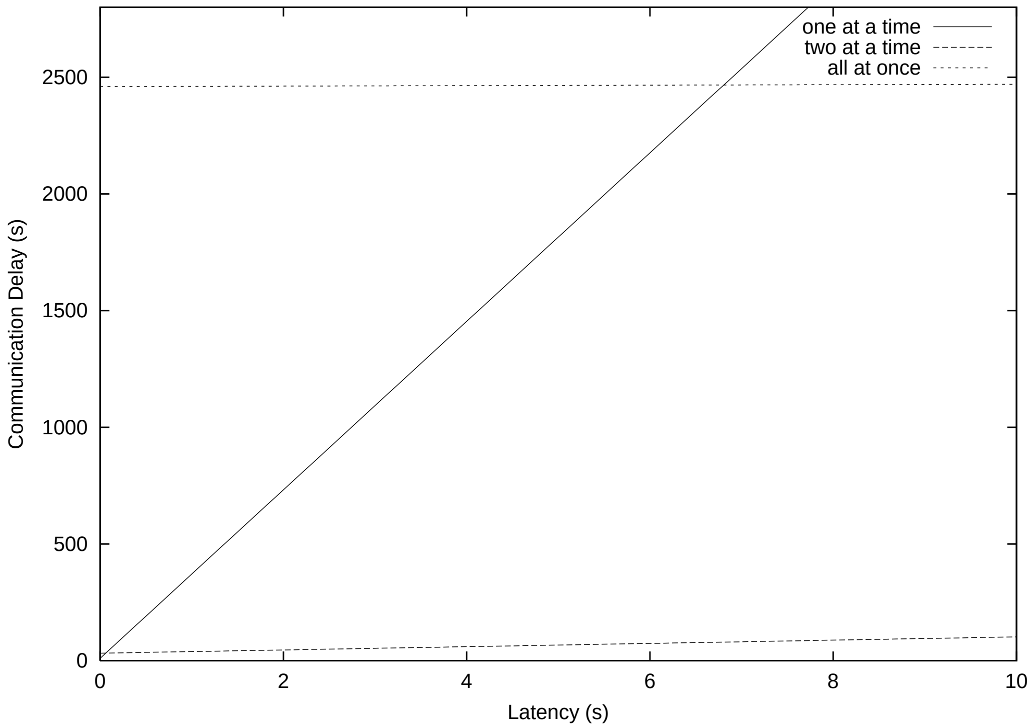


Figure 7.10: Delay with varying latency for hypothetical example

at level $i = 5$; and the summary information for any task is 1 Kbyte. Then the table would look like this:

	number of messages	total size (Kbytes)
one at a time	363	1089
two at a time	9	3276
all at once	3	246033

Now, if we fixed the bandwidth at 100 Kbyte/second and varied the latency, more realistic tradeoffs are seen in Figure 7.10. Here, we see that unless the latency is very small, sending summary information two levels at a time is best. As shown in Figure 7.11, if we fix latency to be one second and vary the bandwidth, for all realistic bandwidths sending summary information two levels at a time is again best.

This simple protocol illustrates how communication can be minimized by sending summary information at particular levels of granularity. If the agents chose not to send summary information but the unsummarized hierarchies instead, they would need to send their entire hierarchies. As the experiment shows, as hierarchies grow large, sending the entire hierarchy ("all at once") cause great communications delay. Thus, using summary

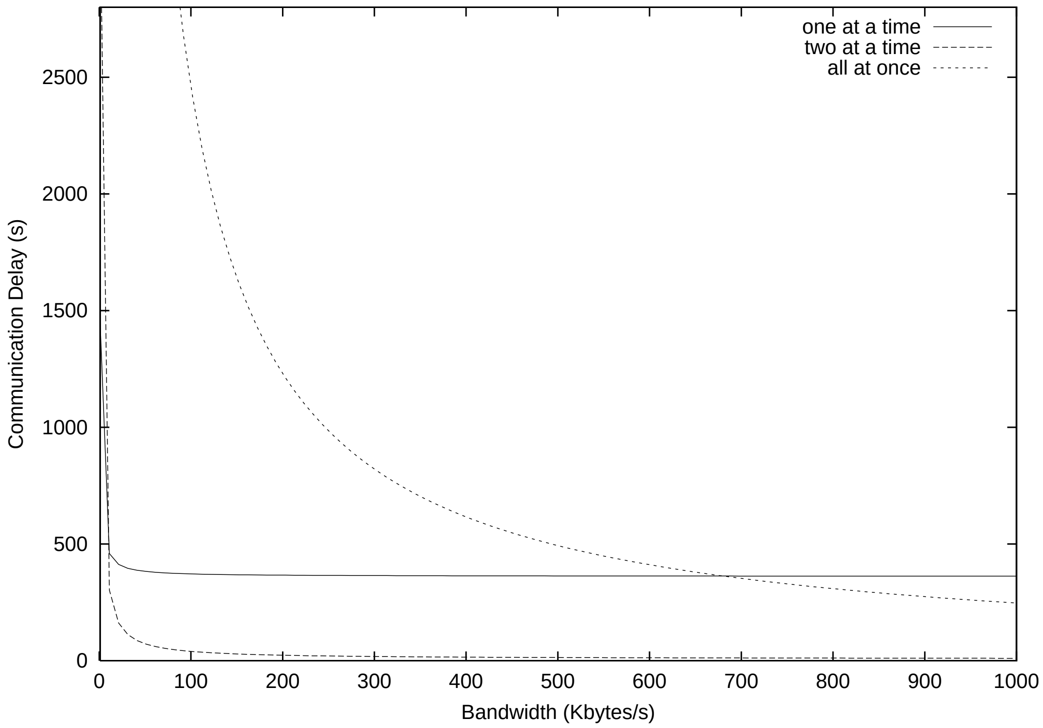


Figure 7.11: Delay with varying bandwidth for hypothetical example

information (as opposed to not using it) can reduce communication exponentially when solutions can be found at abstract levels.

At the other extreme, if the agents sent summary information one task at a time, the latency for sending so many messages can grow large for larger task hierarchies. If solutions could only be found at primitive levels, then sending summary information one task at a time would cause an exponential latency overhead compared to sending the entire hierarchy at once. But, if solutions can be found at intermediate levels, being able to send summary information at different levels of granularity can minimize total delay.

However, this argument assumes that summary information collapses at higher levels in the hierarchy. Otherwise, sending summary information at some intermediate level could be almost as expensive as sending the entire hierarchy and cause unnecessary overhead. For the actual manufacturing domain, tasks in the agents' hierarchies mostly have constraints on different resources, and summarization is not able to reduce summary information significantly because constraints do not collapse. The result is that it is better, in this case, to send the entire hierarchy at once to minimize delay (unless there are unusual bandwidth and latency constraints, as shown in the experiment). Even so, the coordination agent can still summarize the hierarchies itself to take advantage of the computational advantages of abstract reasoning.

This section shows how a domain modeler can minimize communication overhead by communicating summary information at the proper level of granularity. If bandwidth, latency, and a common depth for coordination solutions is known, the domain modeler can perform a hypothetical experiment like the one above for varying granularities of summary information to determine which granularity is optimal. If summary information collapses up the hierarchy, and solutions can be found at intermediate levels, then communication can be exponentially reduced in this manner.

7.3 Multi-Level Coordination of Military Coalitions

In order to motivate potential applications for multi-level coordination, here I describe how I embedded the coordination algorithm (from Section 6.1) in a Multi-level Coordination Agent (MCA) that continually coordinates a group of planning agents with hierarchical plan libraries in successive episodes. I describe how it has been demonstrated in a fictional United Nations (UN) peace-keeping scenario ("Binni Scenario") for the CoABS (Control of Agent Based Systems) DARPA Program.[7]

Initially, I characterize the MCA and its interactions with client task agents. Then I describe two demonstrations of the MCA's capabilities for the Binni Scenario. The first is a stand-alone demonstration of the coordination of coalitions. Then, I explain how the coordinator is integrated into a larger demonstration integrating the software of many other institutions.

7.3.1 Multi-Level Coordination Agent

The MCA centrally coordinates the plans of requesting task agents in episodes. When a task agent requests coordination, the MCA collects the current plans of all agents it knows about. If the collected plans are not pre-summarized, the MCA derives the summary information. Then, the MCA coordinates the summarized plans according to the

[7]The demonstrations discussed here were developed by graduate students under the direction of Edmund Durfee, the Principal Investigator for the Coalition Agents eXperiment (CoAX) TIE at the University of Michigan. Specific names are given in the Acknowledgments.

algorithm in Section 6.1. If the MCA decomposes an abstract plan and has not received the summary information for a subplan, it requests it from the task agent and waits until it has received it. The task agent can refuse to provide more detailed information, in which case the MCA must choose another subplan (if an *or* decomposition) or choose another abstract plan to decompose.

The MCA generates solutions in the form of synchronization and decomposition constraints for each task agent. By default, the MCA evaluates solutions according to the makespan (or completion time) for each agent and prunes the search space where solutions have costs greater than some previously found solution. Because the coordination algorithm is complete, all Pareto-optimal solutions will eventually be found.

Because the agents may not be able to wait long for solutions to be collected, each solution is posted to a list for selection at any time. Because this software was developed for application to the Binni Scenario, the solutions are graphically posted to window for a military commander who has authority to select and issue a coordinated plan.

Solutions at the highest level of abstraction are found very quickly, and more efficient solutions are posted with less frequency as the MCA digs deeper into the hierarchies, giving the coordination the flavor of an anytime algorithm [Zilberstein and Russell, 1992]. Whenever the user selects a solution, the coordinator aborts its search for more solutions and sends the corresponding plan synchronization and decomposition constraints to the agents. Alternatively, the MCA may find no solutions, or the user may decide no presented solutions are feasible, and a *fail* message is sent to the agents in conflict. Once it sends messages back to the task agents, the MCA processes the next coordination request, starting a new episode.

7.3.2 Coordinating Coalitions in Binni

The peace-keeping scenario involves coalitions of UN forces that must try and maintain a Total Exclusion Zone (TEZ) between two warring factions (Gao and Agadez) in a fictional city-state in Africa called Binni. Details of the scenario are given in a comprehensive document developed for the DARPA CoABS (Control of Agent Based Systems) program [Rathmell, 2001].

Figure 7.12 shows the layout of locations and routes through which the UN forces

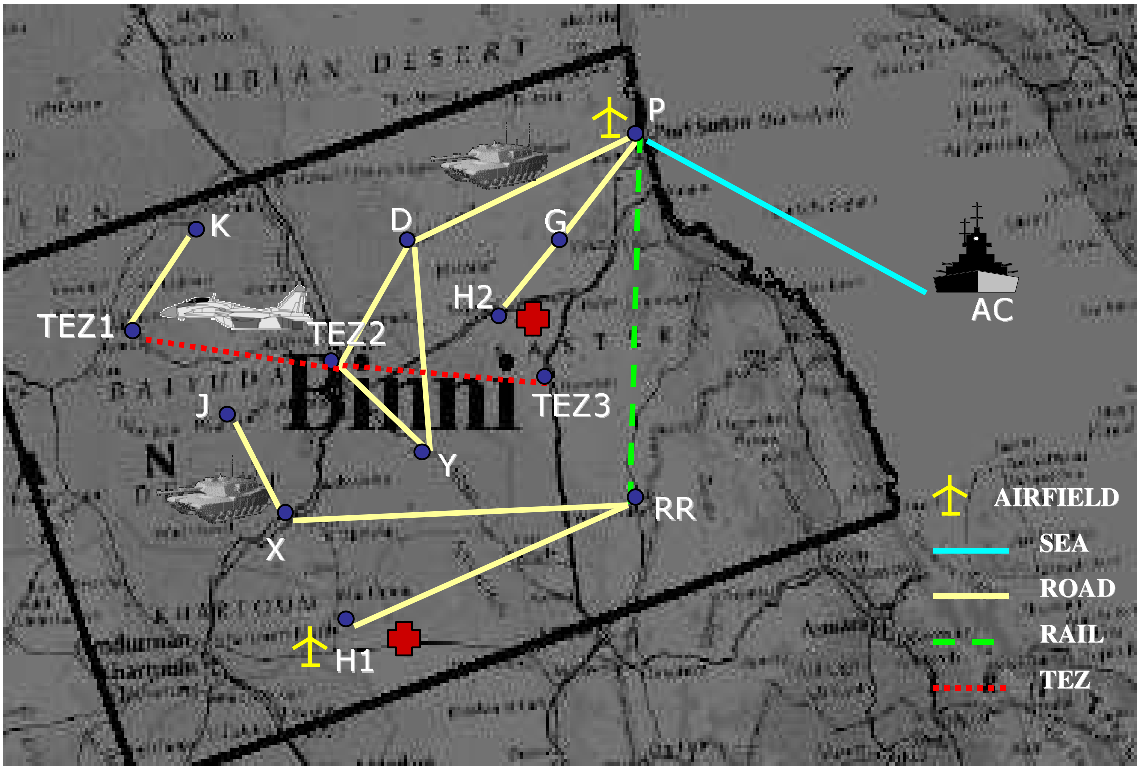


Figure 7.12: UN forces in Binni

travel. An AIRFORCE commanding agent is responsible for making air strikes along the TEZ from an aircraft carrier (AC) to ensure that the opposing forces are separated. An army division, ARMYDIV1, is responsible for reaching the Agadez forces at location X with supplies. Another army division, ARMYDIV2, has a similar task of moving to K where the Gao forces are located. The motivation is to monitor the opposing forces and keep them from crossing the TEZ while providing humanitarian aid. In addition, a LOGISTICS agent has goals to provide humanitarian aid to refugees at locations H1 and H2. All agents are originally on the aircraft carrier (AC), and each has choices of different ways to accomplish its goals. AIRFORCE must avoid bombing the TEZ while ARMYDIV2 passes through. ARMYDIV2 cannot travel through TEZ2 after the bombing which destroys its roads. There is also contention for sea (AC to P) and rail (P to RR) transport among army divisions and logistics.

In the demonstration, task agents (representing coalition partners) each derive summary information for their abstract plans offline and then share the summary information

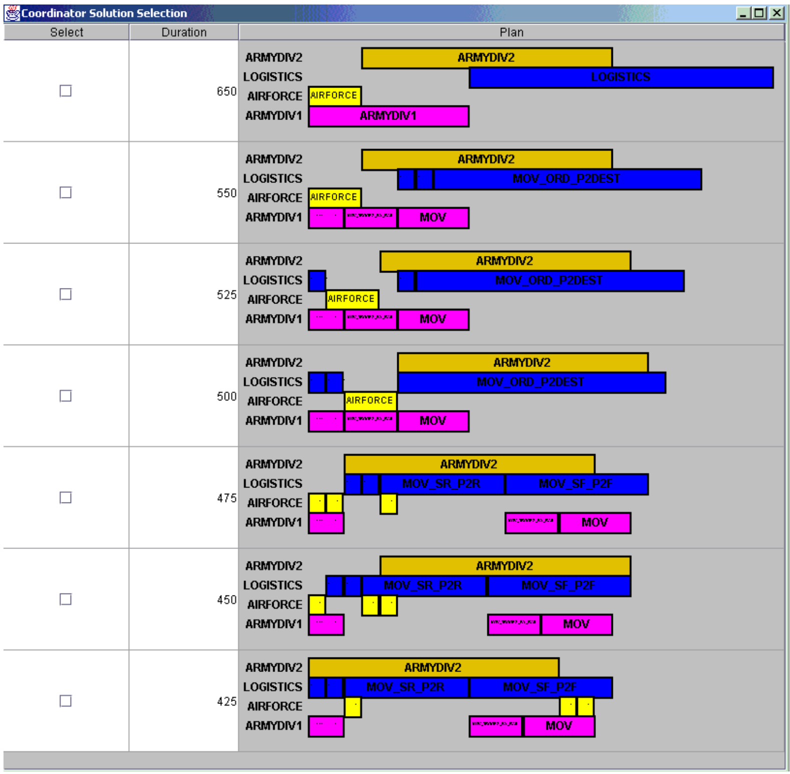


Figure 7.13: Window for selecting coordination solutions

of their current plans with the MCA.[8] The MCA posts several solutions at different levels of abstraction to the selection panel shown in Figure 7.13. The solution at the highest level of abstraction shows that as long as ARMYDIV2 begins its journey after the AIRFORCE makes its strikes, and the LOGISTICS team transports supplies after ARMYDIV1 has reached its destination, the coalitions can execute their plans freely. Successive solutions show improved completion times for the group, and the optimal solution (in terms of overall makespan) is shown at the bottom.

When the commander chooses a solution from the selection window, the MCA sends plan modifications (temporal constraints and blocked *or* subplans) for the selected coordination solution to the task agents. The task agents incorporate these into procedures that they execute using UMPRS (University of Michigan Procedural Reasoning System) [Lee *et al.*, 1994]. They add signal and wait communication primitives to their UMPRS procedures for the synchronization constraints and remove blocked *or* subgoals. The agents then execute their plans.

[8]The communication framework is built upon the CoABS (Control of Agent Based Systems) Grid [Grid, 1999] that serves as an infrastructure for building flexible agent based systems.

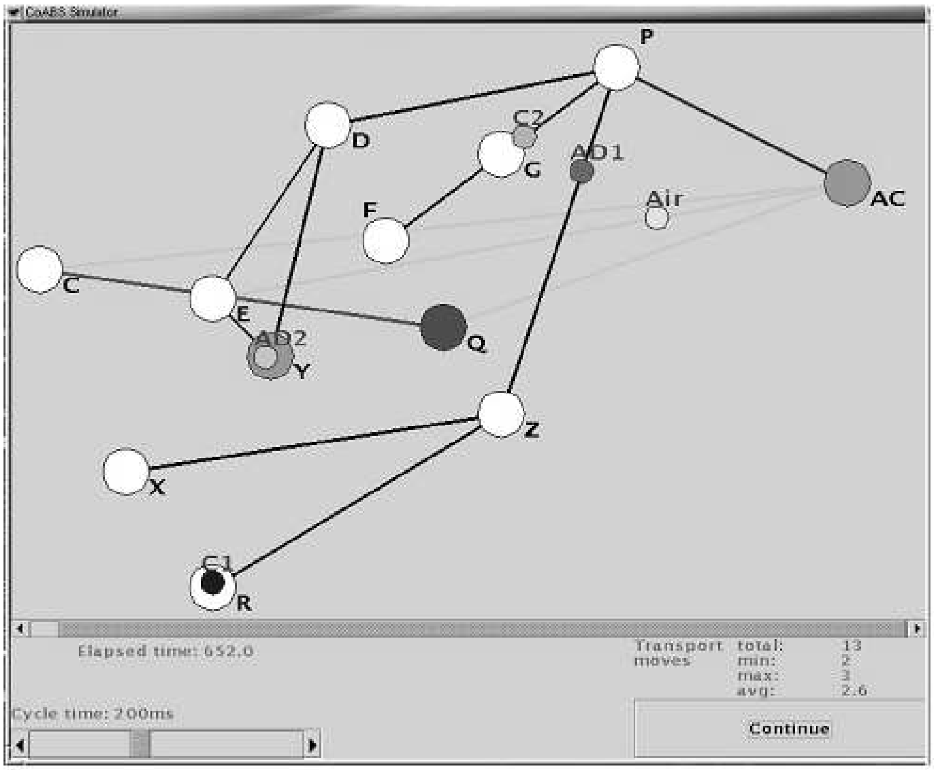


Figure 7.14: Multiagent simulator

A simulator demonstrates execution as it receives commands from UMPRS primitive actions and gives feedback on the effects of each action. The simulator is used to visualize and debug the plan execution and interactions of the agents. Figure 7.14 is a screen shot of the simulator illustrating the coalition agents' execution.

7.3.3 Integrated Binni Demonstration

In another CoAX demonstration of the Binni Scenario, more than twenty types of agents (including the MCA) developed at eighteen different organizations are integrated to show that coalitions can interoperate dynamically and decentrally with the support of current agent software technology in the face of realistic military challenges. These challenges involve commanding military coalitions while taking a number of factors into consideration: political pressures, media coverage, and cultural sensitivities, to name a few.

The MCA's role is to find alterations to the plans of humanitarian aid and medical evacuation teams in response to changes in the flight plans for air squadrons that main-

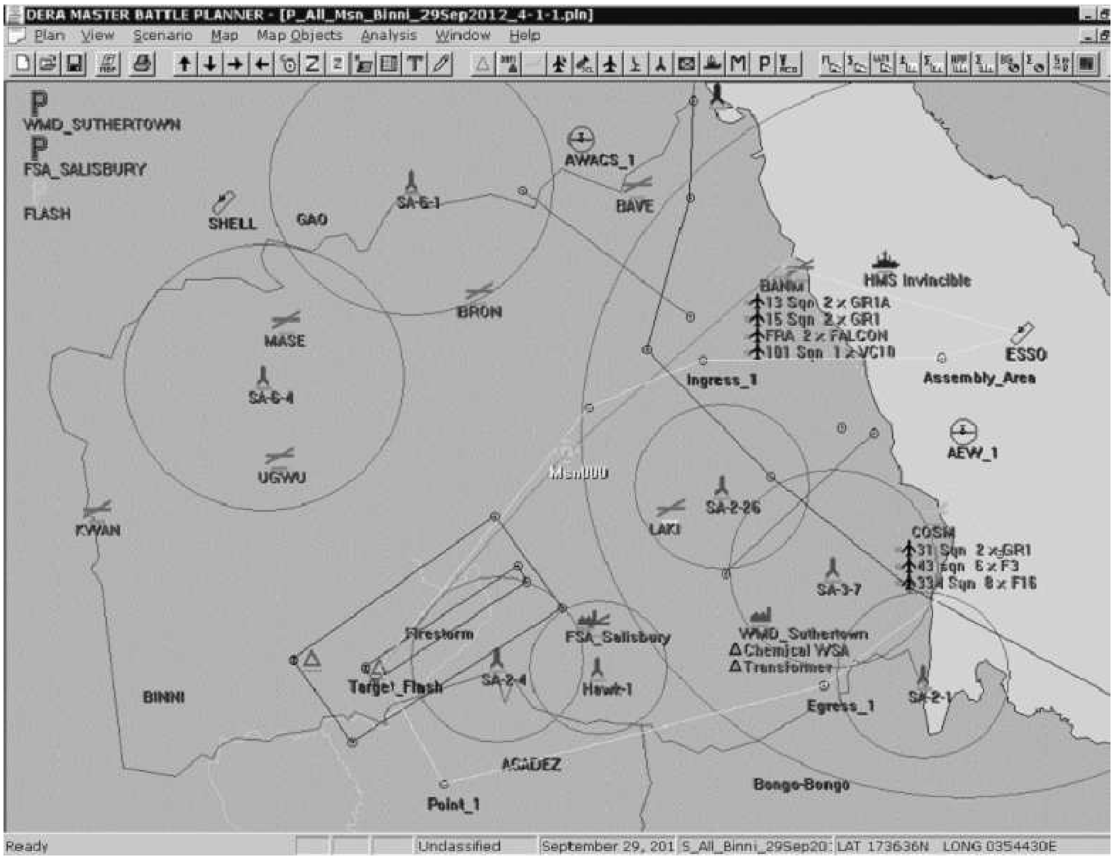


Figure 7.15: Binni, a fictional city-state in Africa

tain the TEZ. In the demonstration, observers detect that the opposing forces have been misleading the UN on their locations and are in position to move around the TEZ. The TEZ must be relocated, but a herd of elephants in a nearby safari park must be protected. Once observers determine the location of the herd and its direction of travel, the Master Battle Planner (MBP) determines how the air squadrons will be deployed to relocate the TEZ.[9] The MBP application is used to visualize and command military forces as shown in Figure 7.15.

The MBP requests coordination from the MCA and sends plans for the air squadrons with hard time constraints. The MCA collects these and the plans of the humanitarian aid and medical evacuation teams and coordinates their movements to avoid potential friendly fire at the new TEZ and air collisions. Solutions in Figure 7.16 show how the MCA finds alternate solutions favoring different agents (in terms of minimizing completion time). The results are passed to the I-DEEL agent (provided by the Artificial

[9]This is an application provided by QinetiQ.

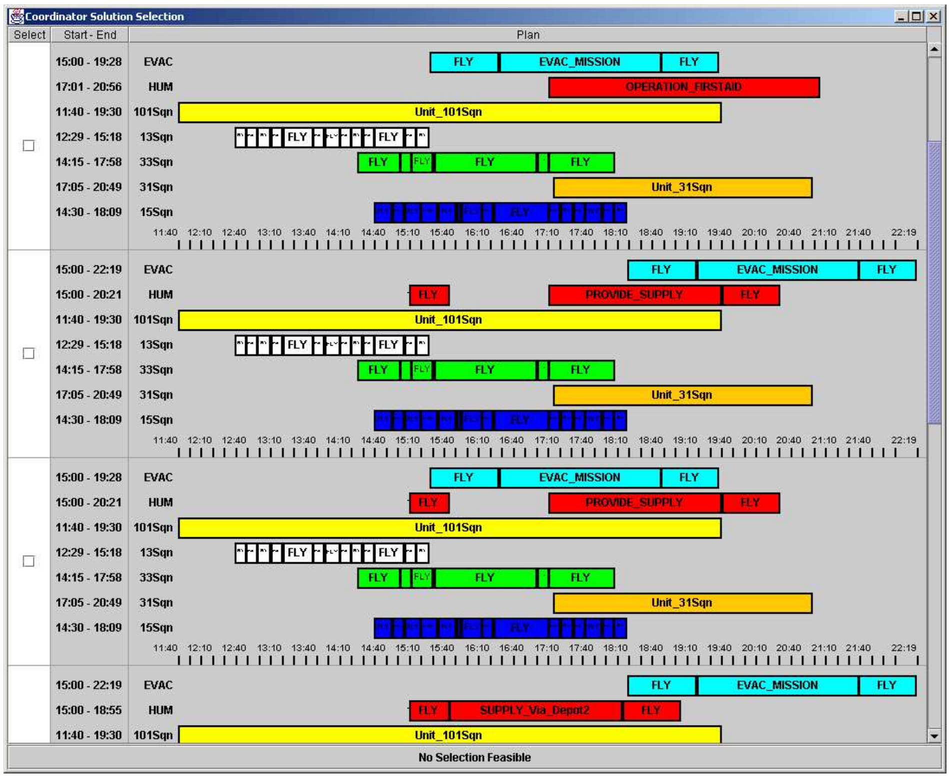


Figure 7.16: Solution selection window

Intelligence Applications Institute) that tracks events and interfaces back to the MBP to plan combat operations. Meanwhile the humanitarian aid and medical evacuation teams are re-tasked based on the selected coordination solution.

The implementation of the MCA and its use in the Binni demonstrations show how multi-level reasoning can be simply used in a realistic application to continually coordinate a group of agents. In the MCA's interactions with the task agents, solutions selected at abstract levels are passed to a robust execution system (UMPRS), taking advantage of the common hierarchical task representation and preserving decomposition choices for flexible execution. In the integrated CoAX demonstration, multi-level coordination is used as a tool to support logistical commanding of coalitions when plans change. This validates the MCA as a viable tool for continual coordination.

PART III

Planning

CHAPTER 8

Concurrent Hierarchical Planning

In Parts I and II, I have described and evaluated a framework for coordinating a group of agents with interacting hierarchical plans. Many of these abstract reasoning techniques and benefits also benefit single-agent planning and scheduling. The coordination algorithm described in Section 6.1 is based on refinement planning similar to that of HTN planning. However, I claim that abstract reasoning techniques can be applied to a wide range of single-agent planning and scheduling approaches, which coordinating agents can also exploit.

In Section 2.3.1, I explained how refinement and local search planners differ. In this chapter, I describe a concurrent hierarchical refinement planning algorithm as well as the use of abstraction in ASPEN [Chien *et al.*, 2000b], a local search planner with representations for concurrent action and abstract tasks. I also explain how decomposition heuristics and complexity advantages of using summary information differ for local search planners like ASPEN.

8.1 A Concurrent Hierarchical Refinement Algorithm

The coordination algorithm described in Figure 6.1 assumes that each agent's plan is internally consistent—no conflicts can occur in the execution of any task in any agent's plan without a task of another agent causing them. The coordination algorithm can easily be modified to work as a concurrent hierarchical single-agent planning algorithm by dropping the assumption that a task is internally consistent.

In order to do this, the *CanAnyWay* test must be extended to make sure that each *plan* in *plans* is internally consistent. This is determined offline when summary conditions are derived. If any summary condition is may-clobbered, then the parent task is not internally consistent and labeled as such. Then, during planning, tasks that are not internally consistent are decomposed until the conflicts at lower levels can be resolved. If a condition is discovered to be must-clobbered, then the parent task is inconsistent and $\neg$*MightSomeWay* is true causing the planning algorithm to backtrack and choose another *or* subplan to avoid the inconsistent branch. If no such *or* subplan is available, then the planner soundly recognizes that there are no solutions to the problem.

With this modification, the algorithm can be used for hierarchical refinement planning for a single agent or also for interleaving single agent planning with coordination. As the plans of the agents are expanded, the algorithm detects and resolves internal conflicts and inter-agent conflicts.

Since the single-agent hierarchical refinement planning algorithm only differs from the coordination algorithm (Section 6.1) in that it may need to resolve conflicts internal to an abstract task, it can use the same search techniques and heuristics described in Section 6.2. Inconsistent search spaces are pruned by detecting $\neg$*MightSomeWay*, and previously found solutions are used to prune spaces where the solution cost must be worse. In addition, the EMTF and FTF heuristics can guide the decomposition to find solutions more quickly.

For the same reason, the complexity advantages of the coordination algorithm also transfer to the planning algorithm. Summarizing Section 6.3.2, these advantages occur when a solution is a partially elaborated hierarchy that has not been fully expanded. The partially expanded hierarchy has exponentially fewer tasks, which serve as input to a threat resolution algorithm whose complexity is exponential with the number of tasks. Thus, the worst case complexity of finding the abstract solution is a factor of a composite of two exponentials less than finding the fully expanded solution.

8.2 Abstraction in Iterative Repair Planning

In this section, I describe techniques for using summary information in local search planners to reason at abstract levels effectively. In iterative repair planning, a technique called *aggregation*, which involves scheduling hierarchies of tasks, outperforms the movement of tasks individually [Knight *et al.*, 2000]. But, can summary information be used in an iterative repair planner to improve performance when aggregation is already used to exploit hierarchy? I demonstrate that summarized state and resource constraints makes exponential improvements by collapsing constraints at abstract levels. First, I describe how I use aggregation and summary information to schedule tasks within an iterative repair planner. Next, I analyze the complexity of moving abstract and detailed tasks using aggregation and summary information. Then I describe how a heuristic iterative repair planner can exploit summary information.

Moving tasks is a central scheduling operation in iterative repair planners. A planner can more effectively schedule tasks by moving related groups of tasks to preserve constraints among them. Hierarchical task representations are a common way of representing these groups and their constraints. Aggregation involves moving a fully detailed abstract task hierarchy while preserving the temporal ordering constraints among the subtasks. Moving individual tasks independent of their parent, siblings, and subtasks is shown to be much less efficient [Knight *et al.*, 2000]. Valid placements of the task hierarchy in the schedule are computed from the state and resource usage profiles for the hierarchy and for the other tasks in the context of the movement. A hierarchy's profile represents one instantiation of the decomposition and temporal ordering of the most abstract task in the hierarchy.

The approach taken here involves reasoning about summarized constraints in order to schedule abstract tasks before they are decomposed. As I will show in Section 8.2.2, scheduling an abstract task is computationally cheaper than scheduling the task's hierarchy using aggregation when the summarized constraints more compactly represent the constraint profiles of the hierarchy. This improves the overall performance when the planner/scheduler resolves conflicts and finds solutions at abstract levels before fully decomposing tasks. However, because these summarized constraints abstract away information about the timing of constraints and choices of decomposition, solutions may not be found until tasks are refined to a lower level of abstraction. In this case, aggregation can be still be used to move partially elaborated hierarchies based on their summarized constraints. The heuristics in the next section describe how to efficiently find solutions at abstract levels.

8.2.1 Decomposition Heuristics for Iterative Repair

Reasoning about summarized constraints only translates to better performance if the movement of summarized tasks resolves conflicts and advances the search toward a solution. There may be no way to resolve conflicts among abstract tasks without decomposing them into more detailed ones. So when should summary information be used to reason about abstract tasks, and when and how should they be decomposed? Here, I describe techniques for reasoning about summary information as abstract tasks are detailed.

I explored two approaches that reason about tasks from the top-level of abstraction down in the manner described in Section 6.2. Initially, the planner only reasons about the summary information of fully abstracted tasks. As the planner manipulates the schedule, tasks are gradually decomposed to open up new opportunities for resolving conflicts using the more detailed child tasks. One strategy (that I will refer to as *level-decomposition*) is to interleave repair with decomposition as separate steps. Step 1) The planner repairs the current schedule until the number of conflicts cannot be reduced. Step 2) It decomposes all abstract tasks one level down and returns to Step 1. By only spending enough time at a particular level of expansion that appears effective, the planner attempts to find the highest decomposition level where solutions exist without wasting time at any level. The time spent searching for a solution at any level of expansion is controlled by the rate at which abstract tasks are decomposed.

Another approach is to use decomposition as one of the repair methods that can be applied to a conflict so that the planner gradually decomposes conflicting tasks. This strategy tends to decompose the tasks involved in more conflicts since any task involved in a conflict is potentially expanded when the conflict is repaired. The idea is that the scheduler can break overconstrained tasks into smaller pieces to offer more flexibility in rooting out the conflicts. This resembles the EMTF (expand-most-threats-first) heuristic for the refinement planner that expands (decomposes) tasks involved in more conflicts before others. (Thus, I will refer to this heuristic as EMTF also for local search planners.) This heuristic avoids unnecessary reasoning about the details of non-conflicting tasks. Tasks that are not involved in conflicts are rarely expanded because they are less likely chosen for repair.

A local search planner can also use the FTF heuristic described in Section 6.2. Using summary information, the planner can test each child task by decomposing to the child and replacing the parent's summarized constraints that summarize the children with the particular child's summarized constraints. For each child, the number of conflicts in the schedule are counted, and the child creating the fewest conflicts is chosen with greater probability. The experiments in Chapter 9 report that using FTF can find solutions much more quickly when decomposition choices cause significantly varying numbers of conflicts.

Note that the techniques to prune spaces of inconsistent and higher cost plans described for refinement planning and coordination (Section 6.2) do not apply to local search planning. These techniques rely on the planner to use backtracking to avoid or focus the search on particular plan spaces. While a local search planner does not typically keep any memory of previous search states in order to backtrack, heuristics can guide the search away from unfruitful plan spaces. This thesis does not investigate this topic. An open question for iterative repair planners is how to appropriately interleave conflict repair with optimization. A common practice is to optimize the plan after resolving conflicts [Chien *et al.*, 2000b].

8.2.2 Scheduling Complexity

A local search planner (as described in Section 2.3.1) does not backtrack, but the problem to be solved is the same, so one might expect that complexity advantages are the same as for the refinement planner. However, the search operations for the local search planner can be very different. As mentioned earlier in this section, a previous study of a technique called *aggregation* uses hierarchy to its advantage to eliminate search inefficiencies at lower levels of detail in task hierarchies [Knight *et al.*, 2000]. Thus, it is not immediately clear what additional improvements can be obtained using summary information.

Consider a schedule of n task hierarchies with a maximum branching factor b expanded to a maximum depth of d as shown in Figure 8.1. Suppose each hierarchy has c constraints on each of v variables (states or metric resources). To move a hierarchy of tasks using aggregation, valid intervals must be computed for each resource variable affected by the hierarchy.[1] These valid intervals are intersected for the valid placements

[1]The analysis also applies to state constraints, but I restrict the discussion to resource usage constraints for simplicity.

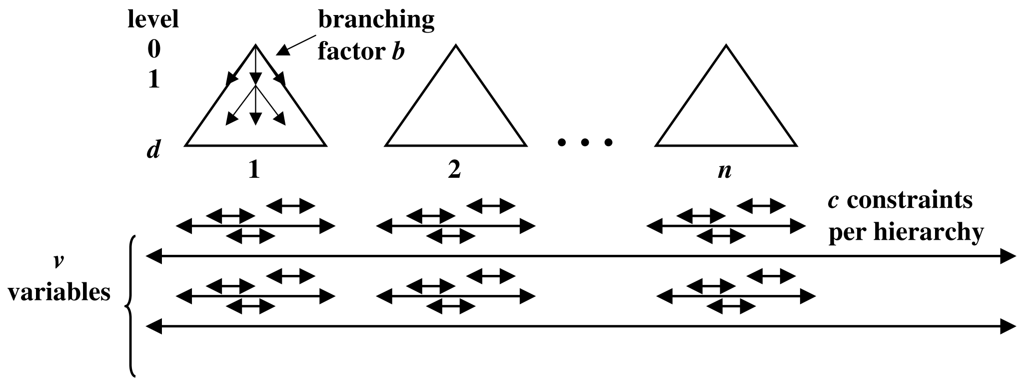


Figure 8.1: Schedule of n task hierarchies each with c constraints on v variables

for the abstract tasks and their children. The complexity of computing the set of valid intervals for a resource is $O(cC)$ where c is the number of constraints (usages) an abstract task has with its children for the variable, and C is the number of constraints of other tasks in the schedule on the variable [Knight *et al.*, 2000]. With n similar task hierarchies in the entire schedule, then $C = (n-1)c$, and the complexity of computing valid intervals is $O(nc^2)$. But this computation is done for each of v resource variables (often constant for a domain), so moving a task will have a complexity of $O(vnc^2)$. The intersection of valid intervals across variables does not increase the complexity. Its complexity is $O(tnr)$ because there can be at most nr valid intervals for each timeline; intersecting intervals for a pair of timelines is linear with the number of intervals; and only $t-1$ pairs of timelines need to be intersected to get the intersection of the set.

The summary information of an abstract task represents all of the constraints of its children, but if the children share constraints over the same resource, this information is collapsed into a single *summary* resource usage in the abstract task. Therefore, when moving an abstract task, the number of different constraints involved may be far fewer depending on the domain. If the scheduler is trying to place a summarized abstract task among other summarized tasks, the computation of valid placement intervals can be greatly reduced because the c in $O(vnc^2)$ is smaller. I now consider two extreme cases where constraints can be fully collapsed and where they cannot be collapsed at all.

In the case that all tasks in a hierarchy have constraints on the same variable, the number of constraints in a hierarchy is $O(b^d)$ for a hierarchy of depth d and branching factor (number of child tasks per parent) b. In aggregation, where hierarchies are fully detailed first, this means that the complexity of moving an task is $O(vnb^{2d})$ because

$c = O(b^d)$. Now consider using aggregation for moving a partially expanded hierarchy where the leaves are summarized abstract tasks. If all hierarchies in the schedule are decomposed to level i, there are $O(b^i)$ tasks in a hierarchy, each with one summarized constraint representing those of all of the yet undetailed subtasks beneath it for each constraint variable. So $c = O(b^i)$, and the complexity of moving the task is $O(vnb^{2i})$. Thus, moving an abstract task using summary information can be a factor of $O(b^{2(d-i)})$ times faster than for aggregation. In Appendix E, I show that resolving conflicts among state state variable constraints is NP-complete in the number of plans. The addition of metric resources cannot make the problem easier in the worst case, so the worst case number of moves to resolve conflicts at level i is $O(k^{b^i})$. This is a factor of $O(k^{b^{d-i}})$ smaller than at the primitive level d. Thus using summary information can make speedups of $O(k^{b^{d-i}}b^{2(d-i)})$ when summary information fully collapses.

The other extreme is when all of the tasks place constraints on different variables. In this case, $c = 1$ because any hierarchy can only have one constraint per variable. Fully detailed hierarchies contain $v = O(b^d)$ different variables, so the complexity of moving a task in this case is $O(nb^d)$. If moving a summarized abstract task where all tasks in the schedule are decomposed to level i, v is the same because the abstract task summarizes all constraints for each subtask in the hierarchy beneath it, and each of those constraints are on different variables such that no constraints combine when summarized. Thus, the complexity for moving a partially expanded hierarchy is the same as for a fully expanded one. However, finding solutions at an abstract level i can still give speedups of $O(k^{b^d - b^i})$ because the number of moves grows exponentially in the worst case (as described earlier). Experiments in Chapter 9 exhibit great improvement for cases when tasks have constraints over common resource variables, but when solutions cannot be found at abstract levels, and summary information does not collapse, reasoning at abstract levels can cause unnecessary overhead.

Along another dimension, scheduling summarized tasks is exponentially faster because there are fewer *temporal* constraints among higher level tasks. When task hierarchies are moved using aggregation, all of the local temporal constraints are preserved. However, there are not always valid intervals to move the entire hierarchy. Even so, the scheduler may be able to move less constraining lower level tasks to resolve the conflict. In this case, temporal constraints may be violated among the moved task's parent and siblings. The scheduler can then move and/or adjust the durations of the parent and siblings to resolve the conflicts, but these movements can affect higher level temporal constraints or even produce other conflicts. At a depth level i in a hierarchy with decompositions branching with a factor b, the task movement can affect b^i siblings in the worst case and produce a number of conflicts exponential to the depth of the task. Thus, if all conflicts can be resolved at an abstract level i, a factor of $O(b^{d-i})$ scheduling operations may be avoided. In Chapter 9, empirical data shows the exponential growth of computation with respect to the depth at which ASPEN finds solutions.

Other complexity analyses have shown that under certain restrictions different forms of hierarchical problem solving can reduce the size of the search space by an exponential factor [Korf, 1987; Knoblock, 1991]. Basically, these restrictions are that an algorithm never needs to backtrack from lower levels to higher levels in the problem. In other words, subproblems introduced in different branches of the hierarchy do not interact. This assumption is unnecessary when using summary information. However, the speedup described above does assume that the hierarchies need not be fully expanded to find solutions.

CHAPTER 9

Mars Rovers Experiments

The experiments I describe here show that summary information improves performance significantly when tasks within the same hierarchy have constraints over the same resource, and solutions are found at some level of abstraction. At the same time, there are cases where abstract reasoning incurs significant overhead when solutions are only found at deeper levels. However, in domains where decomposition choices are critical, I show that this overhead is insignificant when the FTF heuristic quickly guides the search to solutions at deeper levels. These experiments also show that the EMTF heuristic outperforms level-decomposition for certain decomposition rates, raising new research questions. In addition, I show that the time to find a solution increases dramatically with the depth where solutions are found, supporting the analysis at the end of Section 8.2.2, which claims that more constraints at deeper levels exponentially complicate the scheduling problem. At the end of this chapter, I compare the results for the refinement-based coordination and local search planning algorithms.

9.1 Problem Domains

The domain involves a team of rovers that must resolve conflicts over shared resources. I generate two classes of maps within which the rovers move. For one, I randomly generate a map of triangulated waypoints (Figure 9.1). For the other, I generate corridor paths from a circle of locations with three paths from the center to points on the circle to represent narrow paths around obstacles (Figure 9.2). This "corridor" map is used only for an experiment evaluating the FTF heuristic. I then select a subset of the

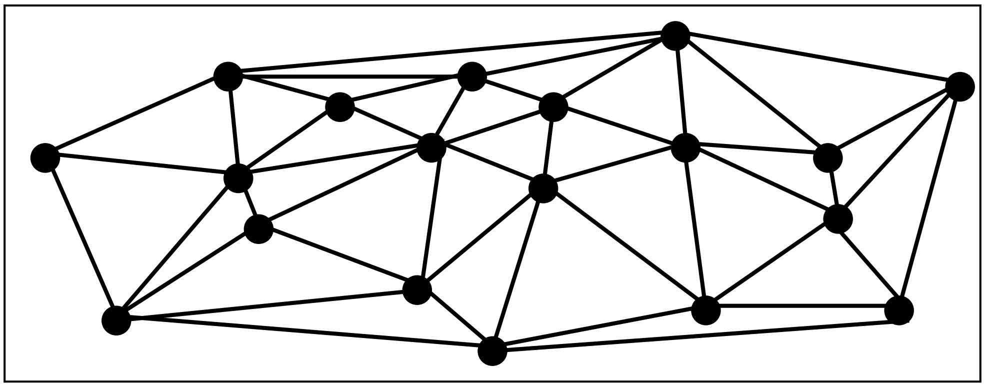


Figure 9.1: Randomly generated rectangular field of waypoints

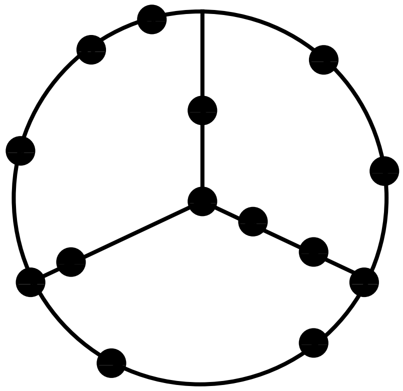


Figure 9.2: Randomly generated waypoints along corridors

points as science locations (where the rovers study rocks/soil) and use a simple multiple traveling salesman algorithm to assign routes for the rovers to traverse and perform experiments. The idea is that a map of the area around a lander is constructed from an image taken upon landing on Mars.

Paths between waypoints are assigned random capacities such that either one, two, or three rovers can traverse a path simultaneously; only one rover can be at any waypoint; and rovers may not traverse paths in opposite directions. These are metric resources. State variables are also used to ensure rovers are at locations from which they leave. In addition, rovers must communicate with the lander for telemetry using a shared channel of fixed bandwidth (metric resource). Depending on the terrain between waypoints the required bandwidth varies. 80 problems were generated for two to five rovers, three to six science locations per rover, and 9 to 105 waypoints. In general, problems that contain fewer waypoints and more science goals are more difficult because there are more interactions among the rovers. Schedules ranged from 180 to 1300 tasks. Note that I use a prototype interface for summary information, and some of ASPEN's optimized scheduling techniques could not be used because the interface is not yet fully implemented.

Schedules consist of an abstract task for each rover that decomposes into tasks for visiting each assigned science location. Those tasks decompose into the three shortest paths through the waypoints to the target science location. The paths decompose into movements between waypoints. Additional levels of hierarchy were introduced for longer paths in order to keep the offline resource summarization tractable.

9.2 Empirical Results

I compare ASPEN using aggregation with and without summarization for three variations of the domain. When using summary information, ASPEN also uses the EMTF and FTF decomposition heuristics. One domain excludes the communications channel resource (*no channel*); one excludes the path capacity restrictions (*channel only*); and the other includes all mentioned resources (*mixed*). Since all of the movement tasks reserve the channel resource, greater improvement in performance is expected when using summary information according to the complexity analyses in Section 8.2.2. This is because constraints on the channel resource collapse in the summary information derived at higher levels such that any task in a hierarchy only has one constraint on the resource. When ASPEN uses aggregation without summary information, the hierarchies must be fully expanded, and the number of constraints on the channel resource is equivalent to the number of leaf movement tasks.

However, tasks within a rover's hierarchy rarely place constraints on the other path variables more than once, so the *no channel* domain corresponds to the case where summarization collapses no constraints. Here the complexity of moving an abstract task is the same for aggregation without summary information for the fully expanded hierarchy as it is for aggregation with summary information for a partially expanded hierarchy.

Figure 9.3 (top) exhibits two distributions of problems for the *no channel* domain. In most of the cases (points along the y-axis), ASPEN with summary information finds a solution quickly at some level of abstraction. However, in many cases, summary information performs notably worse (points along the x-axis). I discovered that for these

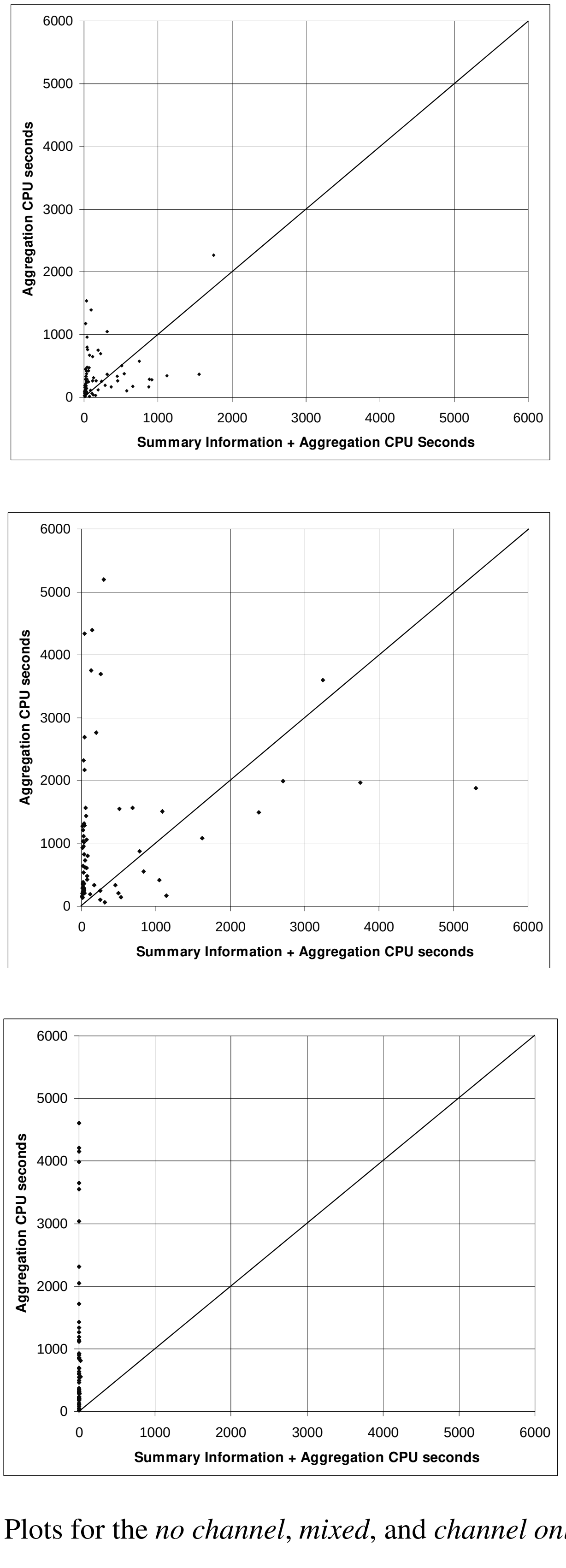


Figure 9.3: Plots for the *no channel*, *mixed*, and *channel only* domains

problems finding a solution requires the planner to dig deep into the rovers' hierarchies, and once it decomposes the hierarchies to the level of the solution, the difference in the additional time to find a solution between the two approaches is negligible (unless the use of summary information found a solution at a slightly higher level of abstraction more quickly). Thus, the time spent reasoning about summary information at higher levels incurred unnecessary overhead. This overhead is rarely significant in backtracking planners because summary information can prune inconsistent search spaces at abstract levels. However, in non-backtracking planners like ASPEN, the only opportunity I found to prune the search space at abstract levels was using the FTF heuristic to avoid greater numbers of conflicts in particular branches. But, for these problems, the FTF did not give summary information an advantage. Later, I will explain why FTF is not helpful for the field problems but very effective for the corridor problems.

Figure 9.3 (middle) shows significant improvement for summary information in the *mixed* domain compared to the *no channel* domain. Adding the channel resource rarely affects the use of summary information because the collapse in summary constraints incurs insignificant additional complexity. However, the channel resource makes the scheduling task noticeably more difficult for ASPEN when not using summary information. In the *channel only* domain (Figure 9.3 bottom), summary information finds solutions at the abstract level almost immediately, but the problems are still complicated when ASPEN does not use summary information. These results support the complexity analysis in the previous section that argues that summary information exponentially improves performance when tasks within the same hierarchy have constraints over the same resource and when solutions are found at some level of abstraction.

Because summary information is generated offline, the domain modeler knows up front whether or not constraints are significantly collapsed. Thus, an obvious approach to avoiding cases where reasoning about summary information causes unnecessary overhead is to fully expand the hierarchies of tasks where summary information does not collapse at the start of scheduling. Because the complexity of moving a task hierarchy is the same in this case whether fully expanded or not, ASPEN does not waste time by duplicating its efforts at each level of expansion before reaching the level at which it finds a solution. Evaluating this approach is a subject of future work.

Figure 9.4 shows the CPU time required for ASPEN using summary information

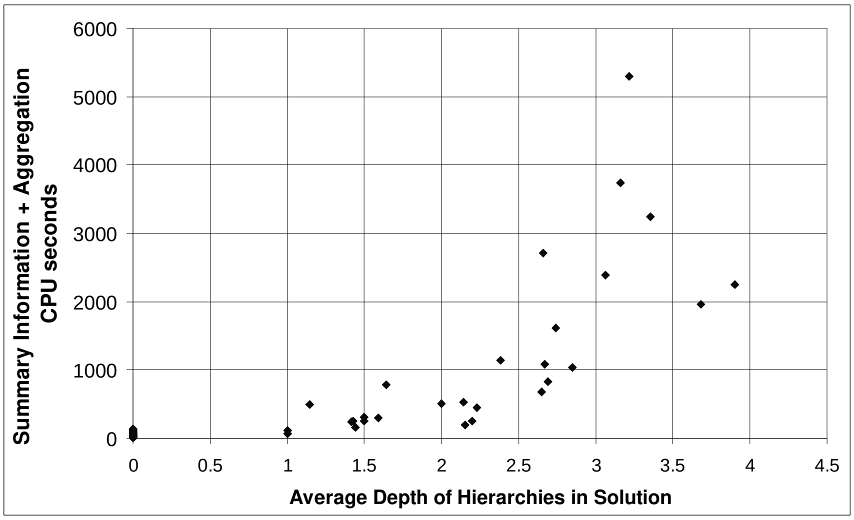


Figure 9.4: CPU time for solutions found at varying depths

for the *mixed* domain for the depths at which the solutions are found. The depths are average depths of leaf tasks in partially expanded hierarchies. The CPU time increases dramatically for solutions found at greater depths. This supports the claim in the previous section that finding a solution at more abstract levels is exponentially easier as a result of an exponential increase in the number of constraints at lower levels. This also explains why there were cases for the *no channel* domain where ASPEN performed better using summary information to find solutions higher up in the hierarchy. Although moving the most abstract tasks using aggregation would have enabled ASPEN to find solutions quickly for fully expanded hierarchies, ASPEN must choose to move lower level tasks independently of their parents and siblings with a small probability of causing temporal constraint violations. Using summary information to find a solution at higher levels of abstraction protects ASPEN against any adverse effects from this.

Earlier I mentioned that the FTF heuristic is not effective for the rectangular field problems. This is because the choice among different paths to a science location usually does not make a significant difference in the number of conflicts encountered—if the rovers cross paths, all path choices usually still lead to conflict. For the set of corridor problems, path choices always lead down a different corridor to get to the target location, so there is usually a path that avoids a conflict and a path that causes one depending on the path choices of the other rovers. When ASPEN uses the FTF heuristic, the performance dominates that of when it chooses decompositions randomly for all but two problems

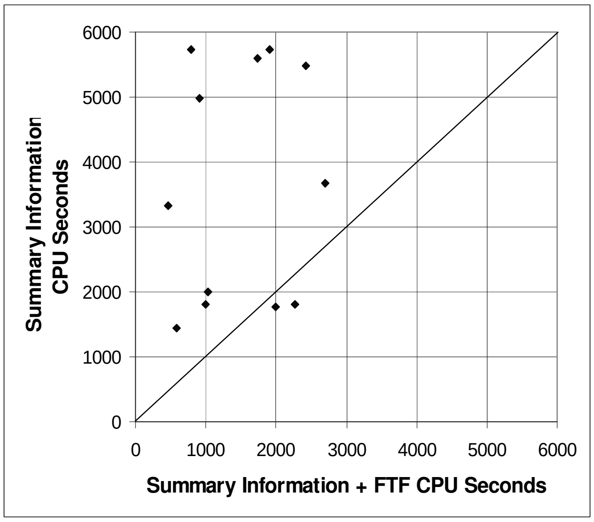


Figure 9.5: Performance using the FTF heuristic

(Figure 9.5). This reflects experiments for the coordination algorithm in Chapter 7 that show that FTF is crucial for reducing the search time required to find solutions.

Figure 9.6 shows the performance of EMTF vs. level decomposition for different rates of decomposition for three problems selected from the set. The plotted points are averages over ten runs for each problem. Depending on the choice of rate of decomposition (the probability that a task will decompose when a conflict is encountered), performance varies significantly. However, the best decomposition rate can vary from problem to problem making it potentially difficult for the domain expert to choose. For example, for problem A in the figure, all tested decomposition rates for EMTF outperformed the use of level decomposition. At the same time, for problem C using either decomposition technique did not make a significant difference while for problem B choosing the rate for EMTF made a big difference in whether to use EMTF or level decomposition. Although these examples show varied performance, results for most other problems tested showed that a decomposition rate of around 15% was most successful. This suggests that a domain modeler may be able to choose a generally successful decomposition rate by running performance experiments for a set of example problems.[1]

A better strategy may be to combine these task expansion heuristics. If the planner repairs the schedule to the point where it cannot make further progress easily, instead of decomposing all tasks, the planner can expand the task that the EMTF heuristic chooses.

[1] For other experiments, I used a decomposition rate of 20%.

Future work will include investigating this approach and the relation of decomposition rates to performance based on problem structure.

In summary, the complexity analyses and experiments show that summary information can offer speedups in single-agent iterative repair planning unless the problem (or domain) has the following characteristics:

- solutions cannot be found at abstract levels;
- summarization does not reduce the constraints derived at higher levels of abstraction; and
- choices of decompositions (*or* branches) lead to similar numbers of conflicts.

If solutions are found at abstract levels, temporal constraint violations are exponentially fewer resulting in exponentially fewer scheduling operations. If summarization collapses constraints on common variables, scheduling operations are exponentially less complex. Even if the planner only finds solutions found at lower levels, the search at higher levels will be insignificant compared to that of lower levels where the schedule's state, resource, and temporal constraints exponentially explode. If neither of these conditions hold, but the number of conflicts in choosing different decompositions of tasks significantly vary, choosing branches with fewer conflicts (the FTF heuristic) will direct the search away from greater numbers of conflicts and make it easier for the planner to find solutions at whatever depth the solutions exist.

In addition to showing speedups in planning and scheduling, the experiments reveal tradeoffs in using the EMTF and level-decomposition heuristics for choosing tasks to expand. Results suggest that if an appropriate decomposition rate is chosen, EMTF will outperform level-decomposition. As mentioned earlier, this investigation raises research questions in how to choose a decomposition rate and how to construct other heuristics based on these.

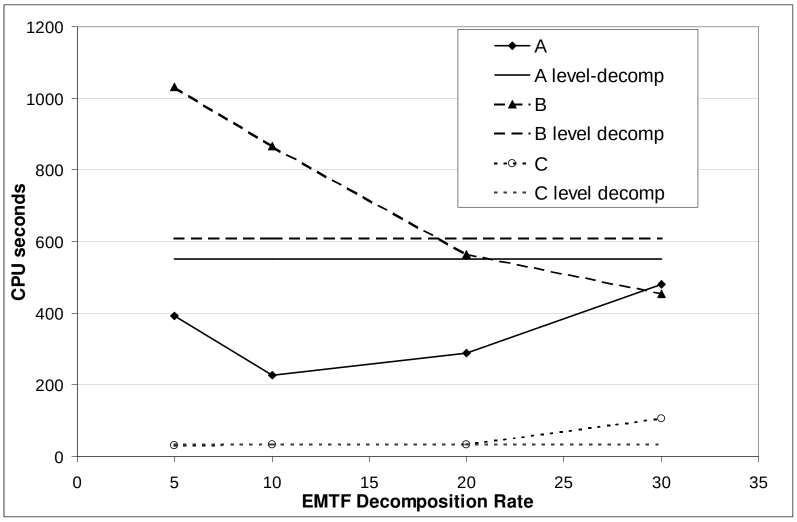


Figure 9.6: Performance of EMTF vs. level-decomposition heuristics

9.3 Comparing Refinement and Iterative Repair Planning

As stated in Section 8.1, the complexity analysis for the coordination algorithm in Section 6.3 applies directly to the refinement planner based on the coordination algorithm. In this section, I compare the complexity analyses and experimental results for these two planning approaches to explain the performance advantages of abstract reasoning that can be expected for different classes of planners and schedulers.

The complexity analyses and experiments show that abstract reasoning in the iterative repair planner is sensitive to the size of summary information derived for abstract tasks. If the size is too large (because of no collapse in summary information), using summary information at abstract levels can be a waste of effort. But, if it sufficiently collapses in size up the hierarchy, then performance improves exponentially. In contrast, the evaluation of refinement planning reveals no sensitivity to the size of summary information, but the number of expanded tasks greatly affects performance. The evaluation of local search planning reveals no sensitivity to the number of tasks. Are these planners really this different?

The answer is no. The sensitivity to the size of summary information did not appear in the evaluation of refinement-based coordination because it was not varied. Section 8.2.2 analyzes the worst case, where no summary information collapses. In that case, for a hierarchy of depth d, branching factor b, and c conditions per task, the number of summary conditions for a task at depth level i is $C = O(b^{d-i}c)$, and the complexity of detecting conflicts in an ordering of plans (at level i) is $O(b^{2i}C^2) = O(b^{2d}c^2)$. If the summary conditions were to completely collapse because all plans in the hierarchy had conditions on the same c variables, the number of summary conditions per task would be simply c, and the complexity of testing would be $O(b^{2i}c^2)$. Thus, the complexity differs by a factor of $O(b^{2(d-i)})$. In our refinement coordination and planning algorithm, this conflict detection is a basic operation that is done for resolving conflicts. So, by collapsing summary information, resolving conflicts at abstract levels gains another exponential performance improvement. This makes the complexity of resolving conflicts $O(k^{b^i}b^{2d}c^2)$ when summary information does not collapse, $O(k^{b^i}b^{2i}c^2)$ when summary information collapses, and $O(k^{b^d}b^{2d}c^2)$ when reasoning at the primitive level (no summary information). Thus, planning at level i can be as much as a factor of $O(k^{b^d-b^i}b^{2(d-i)})$ times faster than planning at level d. This is the same result found for the iterative repair planner. Although both planners described here experience this exponential speedup, without more analysis it is uncertain whether there is some planning algorithm that would *not* experience similar performance gains. Thus, future work is needed to analyze the complexity of planning based on the number of constraints (conditions) instead of just the number of plan operators.

But how could summary information cause unnecessary overhead for the iterative repair planner and not for the refinement coordinator? In the experiments for the evacuation domain (Chapter 7), summary information does not collapse up the hierarchy because each task usually has travel constraints on different locations/paths. This is the worst case for using summary information. However, in finding optimal solutions, the FTF-EMTF coordinator completely dominates the FAF-FAF planner, that represents the state-of-the-art HTN planning heuristic that does not reason about conditions at abstract levels. Other experiments suggest that the FTF heuristic makes the difference in directing the search away from costly search. This, in conjunction with branch-and-bound pruning, eliminates unfruitful search spaces, explaining why using summary information never introduced unnecessary overhead. It is possible that if the FTF heuristic was ineffective, and abstract solutions could not be found at higher levels and used to prune costlier search spaces, reasoning at abstract levels would be duplicating effort at lower

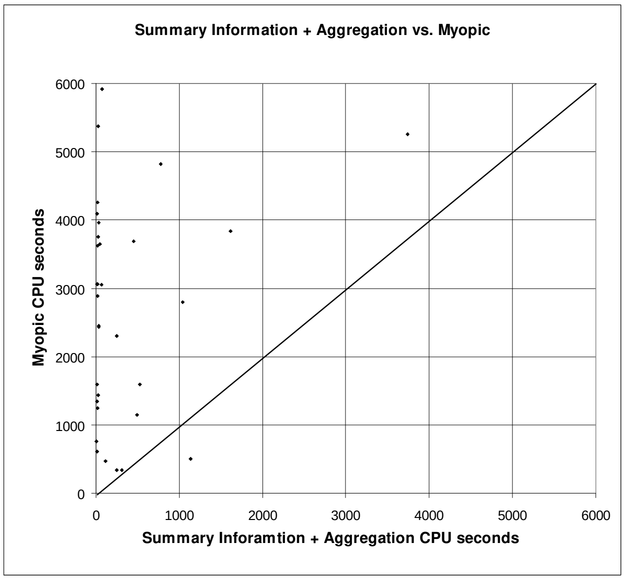


Figure 9.7: Performance of summary information with aggregation vs. myopic

levels causing unnecessary overhead. However, if the domain modeler is aware of these characteristics of the domain, the planner can skip reasoning at abstract levels and use summary information at a level where it can find solutions.

There is also another explanation for why summary information did not appear as effective in the iterative repair planner. The comparisons were made against the planner that still used aggregation. As mentioned before, aggregation already exploits hierarchy to avoid many potential temporal constraints, and previous work evaluating aggregation shows that the manipulation of tasks outside the context of their hierarchy (*myopic* scheduling) results in very poor performance [Knight *et al.*, 2000]. Figure 9.7 shows that this myopic scheduling indeed performs much worse than the combination of summary information and aggregation for the *mixed* domain.

Thus, the performance advantages of using summary information do not differ greatly between the refinement and iterative repair planners described in this thesis. These planners are plan-based in that they manipulate plans (or schedules). This evaluation does not directly apply to state-based planners, but it suggests that reasoning at abstract levels can generally reduce the complexity of search in planning. Planning problems in general are intractable in the number of steps in the generated solution plan [Bylander, 1994]. Summary information enables a planner to reason about fewer abstract tasks that represent all potential (or desirable) action combinations. For example, forward-expanding state-space planners could search through expansions of top-level tasks first using summary information. Then, if no feasible solution is found, the subtasks in the decompositions of one (or all) of the high-level tasks can be added (or substituted), and the search can be repeated. The search space reductions due to fewer operators and collapsed summary information for plan-space planners can also apply to state-space planning.

PART IV

Conclusions and Future Directions

CHAPTER 10

Contributions and Results

This dissertation presents a collection of algorithms to efficiently coordinate a group of agents with hierarchical plans, generate a plan for a single-agent, and interleave planning and coordination. The formalization of summary information and the lower-level algorithms that reason about them also facilitate the construction of other coordination and planning systems that efficiently reason about plans at multiple levels of abstraction. The complexity analyses and experiments in a variety of problem domains quantify the benefits of using summary information in different classes of coordination, planning, and scheduling algorithms. My approach also supports flexible execution of coordinated plans. In summary, using techniques provided by this dissertation for reasoning efficiently about plans at multiple levels of abstraction enables researchers and system designers to scale the capabilities of interleaved coordination, planning, and execution.

10.1 Summary Information

The first contribution of this thesis is the algorithms for deriving and reasoning about summary information for the propositional state and metric resource constraints of abstract tasks. The formalism of the concurrent execution of CHiPs serves as a foundation upon which sound and complete algorithms are constructed for deriving summary conditions and determining whether one task must or may achieve, clobber, or undo a condition of another task. Based on these algorithms, others determine whether a group of agents can decompose and execute a set of abstract tasks under particular ordering constraints in any way (*CanAnyWay*), might decompose and execute them in some way

(*MightSomeWay*), or cannot execute them consistently in any way ($\neg$*MightSomeWay*).

Like the algorithms for summary conditions, the components of the metric resource summarization algorithm allow an agent to determine whether resource usages of abstract tasks must or may conflict. Local search algorithms can directly use these component algorithms to calculate resource profiles for fixed orderings of tasks. Existing refinement planners offer techniques for determining whether a set of actions under partial ordering constraints have resource usage conflicts. Those planners can also use these summarization algorithms for efficiently detecting and resolving conflicts at multiple levels of abstraction.

A domain modeler can run the summarization algorithms offline for a library of plan hierarchies so that summary information is available for the coordination and planning of any set of goal tasks supported by the library. Using algorithms for reasoning about summary information, agents can discover with whom they should coordinate and over which states and resources they must coordinate/negotiate. These formalisms and algorithms serve as a toolbox for building coordination and planning algorithms that can effectively reason about concurrent action at multiple levels of abstraction.

10.2 Coordination and Planning Algorithms

Another contribution is a sound and complete coordination algorithm built on the algorithms for summary conditions. Given a separately constructed plan for each of a group of agents, this algorithm is able to potentially find solutions or determine that there are no solutions at multiple levels of abstraction. The ability to find solutions at abstract levels can preserve decomposition choices for agents for more flexible execution using many existing robust execution systems that exploit similar representations of task hierarchies. A small modification to this algorithm results in a concurrent hierarchical planner that efficiently refines a single agent's top level goals and tasks into consistent plans at multiple levels of abstraction. This planner extends existing HTN planners to reason more efficiently at abstract levels about activities that may execute in parallel with interval temporal constraints. This enables these kinds of refinement-based coordination and planning algorithms to handle a wider class of domains where several tasks may execute during the interval of another. The algorithm is applied to an evacuation domain, a military coalition scenario, and a manufacturing domain to demonstrate its effectiveness in coordinating a group of concurrently executing agents.

The use of summary information in a local search planner is another contribution of this thesis. The strength of local search algorithms is their ability to efficiently reason about large numbers of tasks with constraints on metric resources, state variables, and other complex resource classes. By integrating algorithms for reasoning about summarized propositional state and metric resource constraints into ASPEN (a heuristic iterative repair, local search planner/scheduler), we enable a powerful system to scale to even larger problem domains by reasoning about tasks at multiple levels of abstraction, as shown in its application to planning the actions for a team of Mars rovers. This use of summary information in a different style of planner demonstrates the applicability of abstract reasoning in improving the performance of different kinds of planning (and plan coordination) systems. I also explain how complexity advantages can be extended to other classes of planners.

10.3 Decomposition Search Techniques and Heuristics

Coordination and planning algorithms that use summary information can find solutions at higher levels of abstraction more quickly, but a system may require optimal solutions that only exist at lower levels. For these systems, making decisions about how to decompose task hierarchies is crucial. Another contribution of this thesis is a search strategy involving several techniques for decomposing task hierarchies based on summary information. One search strategy is to find abstract solutions and use them to prune away (in a branch-and-bound style) search spaces that are worse (according to utility or cost) than those found. Another is to prune away search spaces where there are unresolvable conflicts. The EMTF heuristic decomposes tasks involved in greater numbers of conflicts first in order to "root out" conflicts more quickly without introducing unnecessary details of non-conflicting tasks. When decomposing an *or* task, the FTF heuristic prefers tasks involved in fewer conflicts. In selecting subtasks causing fewer conflicts (or blocking tasks involved in more conflicts), conflicts are easily avoided to enable the coordinator/planner to find solutions more quickly. The combination of these effectively avoids unfruitful search spaces, guides the search to solutions more quickly, and finds optimal solutions much more quickly than existing HTN heuristics as shown in experiments. These heuristics are adapted for both the refinement coordination/planning algorithm and iterative repair in ASPEN with similar experimental results.

10.4 Complexity Analyses and Experiments

The complexity analyses and experiments in this dissertation comprise another contribution in their quantitative explanation of the benefits of using summary information in different coordination and planning systems. One analysis shows that finding a solution at a higher level of abstraction is easier by a factor of $O(k^{b^d - b^i})$ where k is some constant; b is the branching factor of the hierarchy; i is the level at which a solution is found; and d is the depth of the hierarchy, where previous hierarchical planning approaches can only find solutions by fully decomposing plans to the primitive level. This complexity reduction occurs in the worst case where the summary information for a plan grows exponentially up the hierarchy because there is no collapse of information during summarization. If summary information fully collapses, our refinement coordination and planning algorithm is sped up by another factor of $O(b^{2(d-i)})$, resulting in a total improvement factor of $O(k^{b^d - b^i} b^{2(d-i)})$.

For iterative repair (local search), another analysis shows that scheduling operations are exponentially cheaper at higher levels of abstraction when using summary information. Resolving a conflict by moving a task at level i in the schedule is as much as a factor of $O(b^{2(d-i)})$ times faster than moving a task in fully decomposed schedule, even when the scheduler uses aggregation to efficiently schedule task hierarchies. However, this best case occurs when summary information fully collapses in the hierarchy during summarization because all tasks have constraints on the same state or resource variable. If there is no collapse in summary information because tasks in each hierarchy have constraints on different variables, then summary information offers no speedup. But, scheduling at abstract levels achieves a speedup along another dimension. Because temporal constraints between tasks grow exponentially as a hierarchy is decomposed, rescheduling a task at the lowest level d, creates a factor of $O(b^{d-i})$ more temporal conflicts than at an abstract level i. But, just as with the refinement-based search, the number of scheduling operations could grow exponentially with the number of tasks that grow exponentially down the hierarchy as well, making an even more dramatic improvement ($O(k^{b^d - b^i})$). Thus, the combined complexity speedup of using summary information is potentially $O(k^{b^d - b^i} b^{2(d-i)})$ for both planners.

These results quantify the advantages of abstract reasoning over non-hierarchical plan merging approaches, such as [Georgeff, 1983]. This extends previous results [Korf, 1987; Knoblock, 1991] that demonstrate exponential search space reductions for hierarchical problem solving when assuming hierarchies do not have interacting subgoals. My analyses make no such harsh assumptions about the problem structure.

I also show that coordination at multiple levels of abstraction minimizes the overall performance in terms of combined computation and execution cost. Moreover, communicating summary information at different levels of abstraction reduces communication costs exponentially during coordination when summary information collapses up the hierarchy and solutions can be found at abstract levels. This is shown in a experiment with the manufacturing domain where summary information is transmitted at different levels of granularity, and bandwidth and latency vary.

Experiments reflect the complexity analyses in this thesis, showing where the use of summarization dominates previous approaches that do not reason at abstract levels about the constraints found in the decompositions of tasks. This is shown for the refinement-based coordinator in the evacuation domain where full decomposition using the FAF heuristic performs worse by orders of magnitude. Experiments for the iterative repair planning (ASPEN) in the Mars rovers domain show even greater improvements when summary information collapses up the hierarchy. However, these experiments also show that scheduling at abstract levels using summary information creates unnecessary overhead resulting in poorer performance in the worst case when summary information does not collapse. This occurs when solutions can only be found at lower levels of decomposition because scheduling at higher levels is just as costly at lower levels and does not lead to a solution.

However, other experiments show that this overhead is insignificant when decomposition search strategies prune the search space. The result is that reasoning about summary information finds solutions even at lower levels much faster than previous approaches. In the evacuation experiments, the combination of the FTF and EMTF heuristics with pruning of costlier and inconsistent search spaces leads to optimal solutions more quickly than previous heuristics. Local search planning cannot take advantage of pruning techniques, but in planning for the team of Mars rovers, ASPEN finds solutions much more quickly when using summary information in combination with the FTF heuristic for problems where *or* branches lead to varied numbers of conflicts.

The refinement and local search coordinators and planners that I evaluated search in the space of plans. However, state-space planning and coordination systems can also reap the benefits of abstraction using summary information in similar ways. My evaluation of summary information explains the potential and expected benefits of its use in the coordination and planning systems presented in this thesis based on domain characteristics. It argues for the use of abstraction in all classes of planning and plan coordination systems and provides analyses that can be applied to predict their potential and expected performance improvements in terms of computation and communication.

CHAPTER 11

Future Directions

This work raises a number of new research problems. In general, future work is needed to develop techniques for applying summary information to wider classes of coordination, planning, and scheduling problems; for summarizing other kinds of information; for handling uncertainty and risk; for interleaving coordination, planning, and execution; for scaling the numbers of agents that can be coordinated; for exploiting synergistic actions; and for learning from previous episodes of coordination, planning, and execution.

There are many ways in which the techniques here can be applied to other classes of problems. One way of doing this is to integrate summary information into other kinds of planners and schedulers and to use these planning techniques in plan coordination. Recently developed planning algorithms have shown much greater performance and more sophisticated task and resource representations compared to planners developed in previous years [Weld, 1999]. Most of these planners do not use hierarchical representations, and none can reason at abstract levels to the extent of the work presented in this thesis. While some of these planners provide ways for a user to infuse domain knowledge to guide the search, hierarchical representations provide qualitatively different domain knowledge that is natural for humans to specify; abstract tasks are like procedures in a programming language that specify the physics of some application. The integration of summary information with more recent planning algorithms can improve the performance of planning and coordination as well as apply to more expressive plan execution systems.

One aspect of this integration with other planners and schedulers involves expanding summary information to include more complex resources. Metric resource usage is currently commonly represented as an instantaneous depletion, but for many applications, representing this usage as some function over time may be necessary. In addition, the usage of one resource may have interactions with other resources. For example, a solar array on a spacecraft may recharge a battery, and actions that deplete all of the solar power will use battery energy as well. Other complex representations of resources include geometrical constraints, variable capacity resources, and volatile objects (e.g. a file system). ASPEN provides a *generalized timeline* interface for a domain expert to specify the physics of arbitrary resource representations that can use iterative repair for planning and scheduling [Knight *et al.*, 2001]. Many of the resource representations just mentioned have been implemented in ASPEN using this interface. How to summarize constraints and effects on these resources to exploit the benefits of abstract reasoning is an open research question.

Another parameter of expressiveness in planning domain descriptions is the temporal model. My approach is based on point-based interval temporal reasoning. This has the same expressiveness as Allen's temporal relations [Allen, 1983]. As mentioned in the related work on plan merging (Section 2.2), simple temporal networks (STNs) represent time ranges within which endpoints of task intervals must execute. For example, task A must begin executing between five and ten seconds before task B ends. Adapting the algorithms here to handle this representation only involves substituting the algorithms for updating and checking the consistency of temporal constraints [Meiri, 1992]. An extension to this representation involves differentiating between constraints that are controllable and uncontrollable. For example, a planner may determine that A start 5 to 10 seconds before B ends, but the duration of B may uncontrollably vary between 20 and 30 seconds, making it impossible for the planner to determine an appropriate time point to start A. Handling these constraints efficiently remains an open problem.

It is also important to consider other ways of summarizing task information to further exploit abstract reasoning. For example, I showed that abstract reasoning with summary information can cause unnecessary overhead in a local search planner when tasks in a hierarchy have constraints on different variables, and summarization does not collapse information at higher levels. If variables could be grouped in class hierarchies, then constraints over different variables in the same class may be summarized as a single constraint on the set of variables. Thus, a coordination or planning algorithm could interleave the decomposition of the class of variables involved in a constraint with the decomposition of the task hierarchy. By further reducing the information at abstract levels with this different kind of summary information, algorithms should be able to further reduce the size of the search space at higher levels of abstraction since the complexity of resolving conflicts at any level of abstraction grows with the number of constraints. Exploring new ways to reason at abstract levels like this one promises better performance for coordination, planning, and scheduling.

Specific to coordination, more work is needed to understand the effects of communicating summary information for different coordination protocols. My study of one protocol in the manufacturing domain only provides preliminary results. Communication can be constrained with respect to bandwidth, latency, windows of availability, and privacy. For example, coordinating Mars surface explorers and orbiters depends greatly on these variables—communication on the surface, between the surface and orbiters, and between both to Earth varies greatly for all these parameters except there are no privacy concerns. More efficient protocols may not be possible without communicating at multiple levels of abstraction.

As mentioned in related work (Chapter 2), other research has provided theories based on models of agents that have joint-intentions, beliefs, and goals [Grosz and Kraus, 1996; Rao and Georgeff, 1995; Fagin *et al.*, 1995]. This dissertation models agents based on more traditional models of planning and makes few references to these more sophisticated mental states. Making a tie with this other work can bring the benefits of hierarchy to higher level multiagent reasoning techniques.

Summary information offers a natural representation of uncertainty. Instead of assuming that primitive level tasks have only *must*, $first$, *always*, and *last* conditions, the domain expert may encode *may* and *sometimes* conditions in tasks to represent uncertain constraints and effects. If information about the likelihood of these constraints and effects is available, probabilities can be included in the summary information representation of tasks and states. Similarly, metric resources could be specified as probability distributions of values over local and persistent ranges. At abstract levels, summary information can then represent the likelihood of achieving goals, the expected cost or utility of abstract plans, and the risk of failure. At the same time, durations of abstract and primitive actions can be represented as distributions over a range and calculated during summarization and coordination/planning, taking into account execution failures. Investigating how coordination, planning, and execution can exploit summary information to better handle uncertainty and risk of failure can enable autonomous systems to better evaluate choices of accomplishing goals and subgoals in an uncertain environment.

Integrating planning and execution systems for particular domains has long been a subject of research for robotics. A three-tier approach of interfacing planning at the decision-level, a reactive execution system, and a reactive control system is a common approach [Gat, 1998]. General approaches to this integration has been the subject of more recent research. This involves providing state updates to the planner and passing schedules from the planner to the execution system that must interact with the control system to sense and manipulate effectors. The capability is needed to also integrate coordination with execution. CASPER (Continuous Activity Scheduling Planning Execution and Replanning) is a continuous planner/scheduler built on top of ASPEN that constantly reacts to state updates sent from the execution system underneath to replan activities in real time to react to changes in the environment and failure [Chien *et al.*, 2000a]. A moving commit window determines which activities have been passed to the execution layer and which activities can still be modified. The use of abstraction can enable continual planners such as this to focus on the details of more important near-term tasks while reasoning at higher levels about the ramifications on long-term activities. Future work should also investigate how to continuously coordinate planning agents using summary information to reason at multiple levels of abstraction appropriately.

With respect to plan execution, the use of summary information can preserve decomposition choices in the hierarchy that can be exploited by robust plan execution systems. This is one tie in the deliberative level to the execution level. The interactions with a continuous coordination or planning system (mentioned in the preceding paragraph) is another. However, more interactions promise more streamlined interleaved planning and execution. Recent research has developed robust execution for multiagent teams in TEAMCORE [Tambe, 1997; Pynadath *et al.*, 1999] based on a joint-intention model. This work provides execution monitoring and failure recovery for a group of agents collaborating in a common team plan including the possibility of agent role re-assignment. This dissertation discusses an approach to generating coordinated plans that agents can expect to execute successfully. However, future work is needed to evaluate and adapt the task representations here to better exploit the capabilities of multiagent execution systems like TEAMCORE.

The coordination examples in this thesis involve handfuls of agents. Being able to scale to numbers of agents of higher orders of magnitude may be essential for some applications, such as coordinating a large constellation of space probes. Certainly, if the plans of each agent are complex and have tight interactions with many others, finding a consistent coordination solution would be intractable for an arbitrarily large set of agents. However, for agents that have limited interactions with others summary information can be used to determine which agents must coordinate and over which constraint variables they must coordinate. One approach would be to divide the agents into interaction groups or resource groups to localize coordination problems. Of course, interactions may extend across groups, so groups may be organized hierarchically with coordination agents handling groups at different levels. Understanding the tradeoffs of alternative organizations of agents and alternative protocols among agents and their groups can provide insight in scaling coordination to applications involving greater numbers of agents. Work in distributed constraint satisfaction problems (DCSP) provides insights on how this might be done [Yokoo and Hirayama, 1998].

This dissertation provides algorithms and search techniques for resolving conflicts among agents, but agents may have synergistic tasks where one agent achieves a subgoal of another. In addition, an agent may depend on others to achieve some state or produce a resource required for its actions to be executed successfully. My algorithms can support this if *phantomization* is included in the task hierarchies. This would involve introducing additional *or* branches with subtasks that only require a precondition that the task is achieved (by another agent or other task in the local plan hierarchy) with no actions. So, if the subgoal is achieved by some other task, the agent can choose that branch and do nothing. Otherwise, the preconditions fail for that branch and the agent must choose another subtask that achieves the goal. However, introducing such branches trades computation time for plan quality. Future work is needed to study this tradeoff in hierarchical planning and coordination and understanding implications for coordination/negotiation protocols.

Given that agents can coordinate single instances of their plans, remembering the results can improve the performance of coordination and execution for future coordination problems. Agents could store solutions to coordinating tasks throughout the hierarchy and retrieve them when coordinating similar tasks in new problem instance. Thus, they can learn from their previous experiences to efficiently handle future conflicts. However, it may be unlikely that the same instances of tasks will occur in the future. Therefore, being able to generalize coordinated tasks to apply to variations of the coordination problem promises greater success in applying the learned knowledge. This has been studied in case-based planning in systems like CHEF [Hammond, 1986]. Extending this work to apply to coordination within this framework is needed in future work.

APPENDICES

APPENDIX A

Summary Conditions for Selected CHiPs

Here I give the derived summary information for tasks from the manufacturing domain. This supplements summary information given as examples to explain the summarization algorithm in Section 4.1.

Production manager's *produce_G_from_H* **plan:**

Summary preconditions:

`at(`$\$srcG$`, G)MuF, available(G)MuF, free(transp2)MuF, ¬full(M2_tray1)MuF, available(M2)MuS`

Summary inconditions:

`at(M2_tray1, G)MaS, available(G)MuS, full(M2_tray1)MuS, ¬at(`$\$srcG$`, G)MuS, ¬full(`$\$srcG$`)MuS, free(transp2)MuS, ¬available(G)MuS, ¬full(M2_tray1)MuS, ¬free(transp2)MuS, available(M2)MuS, at(M2_tray1, H)MaS, available(H)MuS, ¬at(M2_tray1, G)MuS, ¬available(M2)MuS, ¬at(M2_tray1, H)MuS, ¬available(H)MuS, full(`$\$srcG$`)MuS`

Summary postconditions:

`¬at(`$\$srcG$`, G)MuS, ¬available(G)MuS, ¬at(M2_tray1, G)MuS, available(M2)MuS, at(`$\$srcG$`, H)MuL, available(H)MuL, full(`$\$srcG$`)MuL, ¬at(M2_tray1, H)MuL, ¬full(M2_tray1)MuL, free(transp2)MuL`

Production manager's *produce_G_on_M1* **plan:**

Summary preconditions:

`at(bin1, A)MuF, at(bin2, B)MuF, available(A)MuF, free(transp1)MuF, ¬full(`$\$srcG$`)MuF, available(B)MuS, ¬full(M1_tray2)MaS, available(M1)MuS`

Summary inconditions:

at(bin2, B)MuS, available(B)MuS, ¬full(M1_tray2)MuS, at($*srcG*, A)MuS, available(A)MuS, full($*srcG*)MuS, ¬at(bin1, A)MuS, ¬full(bin1)MuS, at(M1_tray2, B)MaS, full(M1_tray2)MuS, ¬at(bin2, B)MuS, ¬full(bin2)MuS, free(transp1)MuS, ¬available(A)MuS, ¬full($*srcG*)MuS, ¬free(transp1)MuS, ¬available(B)MuS, available(M1)MuS, ¬available(M1)MuS

Summary postconditions:

full($*srcG*)MaS, ¬at(bin1, A)MuS, ¬full(bin1)MuS, ¬at(bin2, B)MuS, ¬full(bin2)MuS, free(transp1)MuS, at($*srcG*, G)MuL, available(G)MuL, ¬available(A)MuL, ¬available(B)MuL, ¬at($*srcG*, A)MuL, ¬at(M1_tray2, B)MuL, ¬full(M1_tray2)MuL, available(M1)MuL

Production manager's *produce_G_on_M2* **plan:**

Summary preconditions:

at(bin1, A)MuF, at(bin2, B)MuF, available(A)MuF, free(transp2)MuF, ¬full($*srcG*)MuF, available(B)MuS, ¬full(M2_tray2)MaS, available(M2)MuS

Summary inconditions:

at(bin2, B)MuS, available(B)MuS, ¬full(M2_tray2)MuS, at($*srcG*, A)MuS, available(A)MuS, full($*srcG*)MuS, ¬at(bin1, A)MuS, ¬full(bin1)MuS, at(M2_tray2, B)MaS, full(M2_tray2)MuS, ¬at(bin2, B)MuS, ¬full(bin2)MuS, free(transp2)MuS, ¬available(A)MuS, ¬full($*srcG*)MuS, ¬free(transp2)MuS, ¬available(B)MuS, available(M2)MuS, ¬available(M2)MuS

Summary postconditions:

full($*srcG*)MaS, ¬at(bin1, A)MuS, ¬full(bin1)MuS, ¬at(bin2, B)MuS, ¬full(bin2)MuS, free(transp2)MuS, at($*srcG*, G)MuL, available(G)MuL, ¬available(A)MuL, ¬available(B)MuL, ¬at($*srcG*, A)MuL, ¬at(M2_tray2, B)MuL, ¬full(M2_tray2)MuL, available(M2)MuL

Facility manager's *maintenance* **plan:**

Summary preconditions:

at(bin4, tool)MuF, available(tool)MuF, free(transp1)MaF, ¬full(M1_tray2)MuF, available(M1)MuS, ¬full(M2_tray2)MuF, available(M2)MuS, ¬full(dock5)MuS, free(transp2)MaF

Summary inconditions:

available(M1)MuS, at(M1_tray2, tool)MuS, full(M1_tray2)MuS,

¬at(bin4, tool)MuS,¬full(bin4)MuS, free(transp1)MaS,

available(tool)MuS, ¬available(tool)MuS, ¬full(M1_tray2)MuS,

¬free(transp1)MaS, ¬available(M1)MuS, ¬full(M2_tray2)MuS,

available(M2)MuS, at(M2_tray2, tool)MuS, full(M2_tray2)MuS,

¬at(M1_tray2, tool)MuS, ¬available(M2)MuS, ¬full(dock5)MuS,

¬at(M2_tray2, tool)MuS, full(dock5)MuS, free(transp2)MaS,

¬free(transp2)MaS

Summary postconditions:

¬at(bin4, tool)MuS, ¬full(bin4)MuS, available(M1)MuS,

¬at(M1_tray2, tool)MuL, ¬full(M1_tray2)MuL, available(M2)MuS,

at(dock5, tool)MuL, available(tool)MuL, full(dock5)MuL,

¬at(M2_tray2, tool)MuL, ¬full(M2_tray2)MuL, free(transp1)MaL,

free(transp2)MaL

Inventory manager's *move_parts* plan:

Summary preconditions:

at(bin3, C)MuF, available(C)MuF, free(transp1)MuF, ¬full(dock4)MuF,

at(dock1, D)MuS, available(D)MuS, free($*imtransp*)MaS, at(dock2, E)MuS,

available(E)MuS, ¬full(bin4)MuS

Summary inconditions:

at(dock4, C)MuS, available(C)MuS, full(dock4)MuS, ¬at(bin3, C)MuS,

¬full(bin3)MuS, free(transp1)MuS, ¬available(C)MuS, ¬full(dock4)MuS,

¬free(transp1)MuS, at(dock1, D)MuS, available(D)MuS, free($*imtransp*)MuS,

at(dock2, E)MuS, available(E)MuS, ¬full(bin4)MuS, at(bin3, D)MuS,

full(bin3)MuS, ¬at(dock1, D)MuS, ¬full(dock1)MuS, ¬available(D)MuS,

¬free($*imtransp*)MuS, ¬at(dock2, E)MuS, ¬available(E)MuS, full(bin4)MuS,

¬full(dock2)MuS

Summary postconditions:

at(dock4, C)MuS, available(C)MuS, full(dock4)MuS, ¬at(bin3, C)MuS,

free(transp1)MaS, at(bin3, D)MuS, available(D)MuS, full(bin3)MuS,

¬at(dock1, D)MuS,¬full(dock1)MuS, at(bin4, E)MuL, available(E)MuL,

full(bin4)MuL, ¬at(dock2, E)MuL, ¬full(dock2)MuL, free($*imtransp*)MuL

APPENDIX B

Soundness and Completeness Proofs for Must/May Assert, Clobber, Achieve, and Undo

These proofs show the soundness and completeness of algorithms given in Section 4.3 that determine potential (*may*) and definite (*must*) interactions among summarized tasks.

Lemma The algorithm for determining that $p' \in P$ *must-assert* c' [*by*, *before*] c is sound; i.e. if we determine that p' *must-assert* c' [*by*, *before*] c, then such is the case.

Proof by contradiction. If it were not the case that $p' \in P$ must-assert c' [by, before] c, then there must be a history where for all t' where e' attempts to assert c' at t', there is a t where e requires c to be met at t and [$t' > t$, $t' \geq t$] since we assume that there exists a plan $p \in P$ for which there is a summary condition c; thus, there must be some t where e requires c to be met at t. We need to show that this is false for any c and c'. We claim that the table describing the constraints checked by the algorithm does this as it describes all cases of assertions by summary inconditions and postconditions and all cases of required summary conditions. For rows 1-4 in the table, the algorithm finds that the constraint [${p'}^{+} \leq p^{-}$, ${p'}^{+} < p^{-}$] can be derived from *order*. Since all histories must have to meet the constraints in *order*, [$t_f(e') \leq t_s(e)$, $t_f(e') < t_s(e)$] must be true. Because the summary postconditions of p' are derived from its own postconditions or the summary postconditions of its subplans; and the summary preconditions of p' are derived from its own preconditions or the summary preconditions of its subplans, these summary conditions are ultimately derived from postconditions of plans executed within e' and preconditions of plans executed within e according to the semantics of subplan executions. This means that $t_s(e') < t' \leq t_f(e')$ and $t_s(e) \leq t < t_f(e)$, and $[t' > t,\, t' \geq t]$ is, thus, false. Therefore the rules for rows 1-4 are sound in determining must-assert for their respective cases. [1] For row 5 by similar argument, $t_s(e') < t_s(e)$. As explained at the beginning of this section on Supporting Mechanisms, we assume that the summary conditions of p, p', and all other subplans in P have their intended properties, so since $c' \in in_{sum}(p')$, $always(c)$, and e' attempts to assert c' at t', there is such t' where $t_s(e') < t'$, and t' is less than all $t_s > t_s(e')$ and all $t_f > t_s(e')$ for any execution in $E(h)$ including e by the semantics of subplan executions and *always*. Similarly, $t_s(e) \leq t < t_f(e)$ since $\ell(c)$ is an external precondition of p, and e requires c to be met at t. Thus, $[t' > t, t' \geq t]$ is false. In this manner, all rows of the table used by the algorithm have been verified to show that $[t' > t, t' \geq t]$ is false for all cases and are, thus, sound. □

Lemma The algorithm for determining that $p' \in P$ *must-assert* c' [*by*, *before*] c is complete; i.e. if p' must-assert c' [by, before] c, then the algorithm determines this to be true.

Proof by contradiction. If the algorithm did not determine that $p' \in P$ must-assert c' [by, before] c, then it must have found the constraints in the must-assert table to differ from those in the point algebra table. So, we need to show that that could not be the case if $p' \in P$ actually must-assert c' [by, before] c for any c' and c. For rows 1-4 in the table, the algorithm finds that the constraint $[p'^{+} \leq p^{-},\, p'^{+} < p^{-}]$ cannot be proven from *order*. Since all histories have to meet the constraints in *order*, $[t_f(e') > t_s(e),\, t_f(e') \geq t_s(e)]$ must be true for some histories. However, e' attempts to assert c' at t' such that $t' = t_f(e')$, and e requires c to be met at t where $t = t_s(e)$, so $[t' > t,\, t' \geq t]$ in such histories. This contradicts the assumption that p' must-assert c' [by, before] c, so the rules in rows 1-4 in the table are complete in determining must-assert for their respective cases. For row 5 by similar argument, $t_s(e') \geq t_s(e)$ for some histories. However, e' attempts to assert c' at t' such that $t_s(e') < t'$, and e requires c to be met at $t = t_s(e)$. Thus, $t < t'$; the definition of must-assert is again contradicted; and the algorithm is complete for this case. In rows 7, 11, 18, and 19, c' is *sometimes* incondition, which could be summarizing only some precondition of a plan in p''s decomposition. In this case no plan can attempt to assert c',

[1] Notice that whether c and c' are *must* or *may* does not make a difference since must-assert describes a temporal relation among executions only for cases when the conditions must be met in the executions of their respective plans.

so must-assert is false, contradicting the assumption that it was true. [2] In this manner, all rows of the table used by the algorithm have been verified to contradict the assumption for all cases and are, thus, complete. □

Lemma The algorithm for determining that $p' \in P$ *may-assert* c' [*by*, *before*] c is sound; i.e. if we determine that p' may-assert c' [by, before] c, then such is the case.

Proof by contradiction. If it were not the case that $p' \in P$ may-assert c' [by, before] c, then there is no history where e is the top-level execution of p in $E(h)$; e' is top-level execution of p' in $E(h)$; e' attempts to assert c' at t'; e requires c to be met at t; and [$t' \leq t$, $t' < t$]. Since our algorithm assumes that there exists a plan $p \in P$ for which there is a summary condition c, we know that in all histories there is a top-level execution of p, e, in $E(h)$ and a top-level execution of p', e', in $E(h)$, and we know that in some history for some set of plans with summary information P_{sum}, e' attempts to assert c' at t', and e attempts to assert c at t. Thus, [$t' > t, t' \geq t$] for any such history. So, we must find a contradiction by showing that [$t' \leq t, t' < t$] for any c and c' in some history using the corresponding cases in the table used by the algorithm. For row 1 in the table, the algorithm would find that the constraint [${p'}^{+} > p^{-}$, ${p'}^{+} \geq p^{-}$] cannot be derived from *order*. Since all histories must have to meet the constraints in *order*, [$t_f(e') \leq t_s(e), t_f(e') < t_s(e)$] must be true for some history. By the same argument in the soundness proof for must-assert, $t' = t_f(e')$ and $t = t_s(e)$. However, this means that in [$t' \leq t$, $t' < t$] in h, so we have a contradiction for this case. For row 7 by similar argument, [$t_s(e') \leq t_s(e)$, $t_s(e') < t_s(e)$] for some history h. Because $c' \in in_{sum}(p')$, $c \in in_{sum}(p)$, and $always(c)$, $t_s(e') < t' < t_f(e')$, $t_s(e) < t$, and t is less than $t_s > t_s(e)$ and $t_f > t_s(e)$ for any execution in E. Thus, there must be a history where $t' < t$ for any such c and c'. In this manner, all rows of the table used by the algorithm have been verified to show that [$t' \leq t, t' < t$] for all cases and are, thus, sound. □

Lemma The algorithm for determining that $p' \in P$ *may-assert* c' [*by*, *before*] c is complete; i.e. if p' may-assert c' [by, before] c, then the algorithm determines this to be true.

Proof by contradiction. If the algorithm did not determine that $p' \in P$ may-assert c' [by, before] c, then it must have found the constraints in the may-assert table to differ from

[2] It would be easy to add a flag to the summary information indicating whether c' summarizes an in- or postcondition or not. Then we could alter the $false$ entries in Table 4.1 to specify the ordering constraints under which must-assert is true for the cases when p' does attempt to assert c'. These constraints would be ${p'}^{+} \leq p^{-}$ for rows 7, 11, and 19 and ${p'}^{+} \leq p^{+}$ for row 18.

those in the point algebra table. So, we need to show that that could not be the case if $p' \in P$ actually may-assert c' [by, before] c. For row 1 in the table, the algorithm finds that the constraint $[p'^+ > p^-, p'^+ \geq p^-]$ is actually proved by *order*. Since all histories must have to meet the constraints in *order*, $[t_f(e') > t_s(e), t_f(e') \geq t_s(e)]$ must be true for all histories. But, since $last(c')$ and $first(c)$, $t' = t_f(e')$ and $t = t_s(e)$, so $[t' > t, t' \geq t]$ for all histories, contradicting the definition of may-assert [by, before]. Thus, the rule in row 1 in the table is complete in determining may-assert [by, before] for this case. For row 7 by similar argument, $[t_s(e') > t_s(e), t_s(e') \geq t_s(e)]$ for all histories. But, $t_s(e') < t' < t_f(e')$, $t_s(e) < t$, and t is less than $t_s > t_s(e)$ and $t_f > t_s(e)$ for any execution in E. Thus, again $[t' > t, t' \geq t]$ for all histories, and we have a contradiction. In this manner, all rows of the table used by the algorithm have been verified to contradict the assumption for all cases and are, thus, complete. □

Lemma The algorithm for determining that $p' \in P$ *must-assert* c' *in* c is sound; i.e. if we determine that p' must-assert c' in c, then such is the case.

Proof by contradiction. If it were not the case that $p' \in P$ must-assert c' in c, then there must be a history where for all t' where e' attempts to assert c' at t', there is a t where e requires c to be met at t and either $t' \leq t_s(e)$ or $t' \geq t_f(e)$. We need to show that this is false for any c and c'. For row 1 in the table, the algorithm finds that the constraints $p'^+ > p^-$ and $p'^+ < p^+$ can be derived from *order*. Thus, $t_f(e') > t_s(e)$, and $t_f(e') < t_f(e)$ for all histories. But, $t' = t_f(e')$, so $t_s(e) < t' < t_f(e)$, contradicting the assumption that must-assert is false. Therefore, the rule in the table for this case is sound. Similarly, for row 5 $t_s(e') \geq t_s(e)$, and $t_s(e') < t_f(e)$ for all histories. But, since $always(c')$, p' attempts to assert c' just after $t_s(e')$, so $t_s(e) < t' < t_f(e)$, again contradicting the assumption. In this manner, all rows of the table used by the algorithm have been verified to show that $t' \leq t_s(e)$ and $t' \geq t_f(e)$ are false for all cases and are, thus, sound. □

Lemma The algorithm for determining that $p' \in P$ *must-assert* c' *in* c is complete; i.e. if p' must-assert c' in c, then the algorithm determines this to be true.

Proof by contradiction. If the algorithm did not determine that $p' \in P$ must-assert c' in c, then it must have found the constraints in the must-assert table to differ from those in the point algebra table. So, we need to show that that could not be the case if $p' \in P$ actually must-assert c' in c for any c' and c. For row 1 the algorithm finds that the constraints $p'^+ > p^-$ and $p'^+ < p^+$ cannot be proven from *order*. Thus, $t_f(e') \leq t_s(e)$, or

$t_f(e') \geq t_f(e)$ for some histories. But, $t' = t_f(e')$, so it is not the case that $t_s(e) < t' < t_f(e)$ for such histories, contradicting the definition of must-assert. For row 5 $t_s(e') < t_s(e)$ or $t_s(e') \geq t_f(e)$ for some histories. But, p' attempts to assert c' just after $t_s(e')$, so there are some histories where $t' < t_s(e)$ and some where $t' > t_f(e)$, again contradicting the assumption. As explained in the *must-assert by* completeness proof, c' for row 7 could be summarizing only a precondition, so must-assert is false for this case. In this manner, all rows of the table used by the algorithm have been verified to show that $t_s(e) < t' < t_f(e)$ is false for some histories for all cases and are, thus, complete. □

Lemma The algorithm for determining that $p' \in P$ *may-assert* c' *in* c is sound; i.e. if we determine that p' may-assert c' in c, then such is the case.

Proof by contradiction. If it were not the case that $p' \in P$ may-assert c' in c, then there is no history where e is the top-level execution of p in $E(h)$; e' is top-level execution of p' in $E(h)$; e' attempts to assert c' at t'; and $t_s(e) < t' < t_f(e)$. Thus, in any history where e' attempts to assert c' at t', $t' \leq t_s(e)$ or $t' \geq t_f(e)$. So, we must find a contradiction by showing that $t_s(e) < t' < t_f(e)$ for any c and c' in some history using the corresponding cases in the table used by the algorithm. For row 1 in the table, the algorithm would find that the constraints $p'^+ \leq p^-$ and $p'^+ \geq p^+]$ can neither be derived from *order*. Thus, $t_f(e') > t_s(e)$ and $t_f(e') < t_f(e)$ must be true for some histories. But, $t' = t_f(e')$ for such histories, so $t_s(e) < t' < t_f(e)$, contradicting the definition of may-assert. Similarly, for rows 5-8 $t_f(e') > t_s(e)$ and $t_s(e') < t_f(e)$ must be true for some histories. But, $t_s(e') < t' < t_f(e')$, so again $t_s(e) < t' < t_f(e)$, and we have a contradiction. In this manner, all rows of the table used by the algorithm have been verified to show that $t_s(e) < t' < t_f(e)$ for some histories for all cases and are, thus, sound. □

Lemma The algorithm for determining that $p' \in P$ *may-assert* c' *in* c is complete; i.e. if p' may-assert c' in c, then the algorithm determines this to be true.

Proof by contradiction. If the algorithm did not determine that $p' \in P$ may-assert c' in c, then it must have found the constraints in the may-assert table to differ from those in the point algebra table. So, we need to show that that could not be the case if $p' \in P$ actually may-assert c' in c. For row 1 in the table, the algorithm finds that either the constraint $p'^+ \leq p^-$ or $p'^+ \geq p^+$ is actually proved by *order*. Thus, $t_f(e') \leq t_s(e)$ or $t_f(e') \geq t_f(e)$ must be true for all histories. But, since $last(c')$, $t' = t_f(e')$ so in no history could $t_s(e) < t' < t_f(e)$ be true, contradicting the definition of may-assert. Thus, the rule in row 1 in the table is complete in determining may-assert for this case. For rows 5-8 by similar argument, $t_f(e') \leq t_s(e)$ or $t_s(e') \geq t_f(e)$ for all histories. But, $t_s(e') < t' < t_f(e')$, so again $t_s(e) < t' < t_f(e)$ is false for all histories, and we have a contradiction. In this manner, all rows of the table used by the algorithm have been verified to contradict the assumption for all cases and are, thus, complete. □

Lemma The algorithm for determining that $p' \in P$ *must-assert* c' *when* c is sound; i.e. if we determine that p' must-assert c' when c, then such is the case.

Proof by contradiction. If it were not the case that $p' \in P$ must-assert c' in c, then there must be a history where for all t' where e' attempts to assert c' at t', there is a t where e requires c to be met at t, and $t \neq t'$. We need to show that this is false for any c and c'. For row 1 in the table, the algorithm finds that the constraint $p'^+ = p^+$ can be derived from *order*. Thus, $t_f(e') = t_f(e)$ for all histories. But, $t = t_f(e)$, and $t' = t_f(e')$, so $t = t'$, contradicting the assumption that must-assert is false. The other rows are sound because the "if" part of the lemma is false. Therefore, all rows of the table used by the algorithm have been verified for all cases to be sound. □

Lemma The algorithm for determining that $p' \in P$ *must-assert* c' *when* c is complete; i.e. if p' must-assert c' when c, then the algorithm determines this to be true.

Proof by contradiction. If the algorithm did not determine that $p' \in P$ must-assert c' when c, then it must have found the constraints in the must-assert table to differ from those in the point algebra table. So, we need to show that that could not be the case if $p' \in P$ actually must-assert c' in c for any c' and c. For row 1 the algorithm finds that the constraint $p'^+ = p^+$ cannot be proven from *order*. Thus, $t_f(e') \neq t_f(e)$ for some histories. But, $t = t_f(e)$, and $t' = t_f(e')$, so it is not the case that $t = t'$ for such histories, contradicting the definition of must-assert. For row 2 there are no constraints on the executions. But, since t and t' could be at any point within e and e' respectively, so certainly there are histories where $t \neq t'$, again contradicting the assumption. In this manner, all rows of the table used by the algorithm have been verified to show that $t = t'$ is false for some histories for all cases and are, thus, complete. □

Lemma The algorithm for determining that $p' \in P$ *may-assert* c' *when* c is sound; i.e. if we determine that p' may-assert c' when c, then such is the case.

Proof by contradiction. If it were not the case that $p' \in P$ may-assert c' when c, then there is no history where e is the top-level execution of p in $E(h)$; e' is top-level execution of p' in $E(h)$; e attempts to assert c at t; e' attempts to assert c' at t'; and $t = t'$. Thus, in any history where e attempts to assert c at t, and e' attempts to assert c' at t', $t \neq t'$. So, we must find a contradiction by showing that $t = t'$ for any c and c' in some history using the corresponding cases in the table used by the algorithm. For row 1 in the table, the algorithm would find that the constraint $p'^{+} \neq p^{+}$ cannot be derived from *order*. Thus, $t_f(e') = t_f(e)$ must be true for some histories. But, $t = t_f(e)$, and $t' = t_f(e')$, for such histories, so $t = t'$, contradicting the assumption that may-assert is false. Similarly, for row 2 $t_f(e') > t_s(e)$ and $t_f(e') < t_f(e)$ must be true for some histories. But, $t' = t_f(e')$, and $t_s(e) < t < t_f(e)$, so again $t = t'$ for some histories, and we have a contradiction. In this manner, all rows of the table used by the algorithm have been verified to show that $t = t'$ for some histories for all cases and are, thus, sound. □

Lemma The algorithm for determining that $p' \in P$ *may-assert* c' *when* c is complete; i.e. if p' may-assert c' when c, then the algorithm determines this to be true.

Proof by contradiction. If the algorithm did not determine that $p' \in P$ may-assert c' when c, then it must have found the constraints in the may-assert table to differ from those in the point algebra table. So, we need to show that that could not be the case if $p' \in P$ actually may-assert c' when c. For row 1 in the table, the algorithm finds that the constraint $p'^{+} \neq p^{+}$ is actually proved by *order*. Thus, $t_f(e') \neq t_f(e)$ must be true for all histories. But, since $t' = t_f(e')$, and $t' = t_f(e')$, $t \neq t'$ for all histories, contradicting the definition of may-assert. Thus, the rule in row 1 in the table is complete in determining may-assert for this case. For row 2 by similar argument, $t_f(e') \leq t_s(e)$ or $t_f(e') \geq t_f(e)$ for all histories. But, $t_s(e) < t < t_f(e)$, and $t' = t_f(e')$, so again $t = t'$ is false for all histories, and we have a contradiction. In this manner, all rows of the table used by the algorithm have been verified to contradict the assumption for all cases and are, thus, complete. □

Lemma The algorithm for determining that $p' \in P$ *must-*[*achieve*, *clobber*] *c in* $pre_{sum}(p)$ is sound; i.e. if p' must-[achieve, clobber] $c \in pre_{sum}(p)$, then the algorithm determines this to be true.

Proof by contradiction. First, note that as explained in the beginning of this section on Supporting Mechanisms, we assume that the summary conditions in P_{sum} have their intended properties. If the algorithm determines that it is not the case that p' must-[achieve, clobber] $c \in pre_{sum}(p)$, then either p' does not must-assert c' by c; c' is not *must*; $\ell(c') \Leftrightarrow$

$[\neg\ell(c), \ell(c)]$; or there is a p'' and c'' that has the properties described in the algorithm. p' must-assert c' by c because otherwise there would be no execution of p' to attempt to assert c' before or at the time c is required to be met. c' has to be *must* since e' attempts to assert c' in all $h \in H$. There must be a c' such that $\ell(c') \Leftrightarrow [\ell(c), \neg\ell(c)]$ because otherwise there would not be an e' that attempts to assert c'. Now we must show that there can be no p'' and c''. If there were a p'' and c'' such that p' may-assert c' before c''; p'' may-assert c'' by c; and $\ell(c'') \Leftrightarrow [\neg\ell(c), \ell(c)]$, then there would have to be an execution to attempt to assert c'' (where $\ell(c'') \Leftrightarrow [\neg\ell(c), \ell(c)]$) after e' and before or at the time e requires c to be met. There can also be no p'' and c'' such that p' must-assert c' before c''; p'' must-assert c'' by c; and $\ell(c'') \Leftrightarrow [\ell(c), \neg\ell(c)]$ since that would entail having an e'' in $E(h'')$ for all $h'' \in H$ where e' attempts to assert c' before e'' attempts to assert some c'' (where $\ell(c'') \Leftrightarrow [\ell(c), \neg\ell(c)]$), and e'' attempts to assert c'' before or at the time e requires c to be met. The truth of all of the above statements rests on the proofs of the lemmas that priorly established the soundness and completeness of must/may assert relations. Thus, the assumption is false, and the algorithm for must-[achieve, clobber] is sound. $\Box$

APPENDIX C

Proof of Summary Information Properties

This proof shows that the algorithm for deriving summary conditions is sound and complete with respect to the specified propoerties of summary conditions given in Section 4.1. This proof is co-dependent on the proofs in Appendix B that determine interactions between the conditions of summarized tasks.

Theorem The following properties of summary information are met by using the procedure for deriving summary conditions for plans. The set of external [pre, post] conditions for a plan is equivalent to the set of all literals in the plan's summary [pre, post] conditions. The set of conditions that must hold within (in the interval (t_s, t_f) of) some execution of a plan in order for it to be successful is equivalent to the set of all literals in the plan's summary inconditions. A summary [pre, post] condition is *must* iff it is a *must* external [pre, post] condition of the plan. Otherwise, it is *may*.[1] A summary incondition c is *must* iff $\ell(c)$ must hold within all executions of the plan for it to be successful; otherwise, it is *may*. A summary [pre, post] condition c is $[first, last]$ iff $\ell(c)$ must hold at $[t_s, t_f]$ of some execution of the plan for it to be successful; otherwise, $sometimes(c)$. A summary incondition c is $always$ iff $\ell(c)$ must hold throughout (t_s, t_f) for all executions of the plan in order for them to be successful; otherwise, $sometimes(c)$.[2]

[1]This means that a *may* condition is one that must hold for some but not all executions of the plans that have this summary condition. However, a particular plan may have a decomposition such that the *may* condition holds for all of its executions. In this case, this fact is lost in the summary information but can be discovered by looking deeper into the plan hierarchy.

[2]This requires that *always* summary inconditions are also $must$. Summary inconditions could have been derived so that *always* conditions could also be *may*, and mechanisms for determining legal temporal relations among plans would be constructed differently. We have not analyzed the advantages and disadvantages of these options.

Proof by induction over the maximum subplan depth. The base case is a primitive plan p (subplan depth zero). According to the first step of the procedure, the summary [pre, post] conditions include a condition $c = \langle \ell, must, [first, last] \rangle$ for every [pre, post] condition ℓ of p, which must be an external [pre, post] condition of p. According to the semantics for successful execution, these conditions have to hold at [t_s, t_f]. Likewise, the summary inconditions include a condition $c = \langle \ell, must, always \rangle$ for every incondition ℓ of p, and the inconditions must hold throughout (t_s, t_f) for all executions of p. Thus, the base case is satisfied.

Assume that the theorem is true for all plans of maximum depth $\leq k$. Any plan p of maximum depth $k+1$ must have subplans with maximum depths $\leq k$. According to the first step of the procedure, summary conditions for non-primitive p are added in the same way as in the base case for primitive plans based on p's pre, in, and postconditions. The semantics for a successful execution based on these conditions is the same as for primitive plans, so the properties for these summary conditions are met. Additional summary conditions are derived from p's subplans based on whether $type(p)$ is *and* or *or*.

Case 1—$type(p) = and$:

In the step of the procedure for adding summary [pre, post] conditions for an *and* plan, summary conditions of the p's subplans are added if they are not [*must-achieved*, *must-undone*] or *must-clobbered* by some other subplan or have already been added in the first step of the procedure. Any newly added condition c is an external [pre, post] condition because it could not be [achieved, undone] or clobbered by a subplan of p's subplan since c is an external [pre, post] condition according to the inductive hypothesis, and if c were [achieved, undone] or clobbered by another subplan of p, the procedures for determining [*must-achieve*, *must-undo*] and *must-clobber* would have identified such a subplan, and c would not have been added. The *existence* of c is set to *must* by the procedure iff it is a *must* external [pre, post] condition of p because the procedure ensures that it is an external [pre, post] condition of all executions of p by making sure it is a *must* summary [pre, post] condition (thus, a *must* external [pre, post] condition according to the inductive hypothesis) in a subplan and ensuring that there is no other subplan of p that [*may-achieve*, *may-undo*] or *may-clobber* c. Otherwise, the procedure sets $existence(c)$ to *may*. The *timing* of c is set to $[first, last]$ by the procedure iff $\ell(c)$ must hold at [$t_s(e)$, $t_f(e)$] of some execution e of p by identifying a subplan p' that is potentially [least, greatest] temporally ordered according to $order(p)$ with $\ell(c)$ as the literal of a [$first$, $last$] summary [pre, post] condition c'. By the inductive hypothesis, c' must hold at [$t_s(e'), t_f(e')$] of some execution e' of p', and if p' is potentially ordered [least, greatest], then there must be some execution e of p where [$t_s(e) = t_s(e'), t_f(e) = t_f(e')$]. Otherwise, the procedure sets $timing$ to $sometimes$.

In the step of the procedure that adds summary inconditions for an *and* plan, a condition c is added iff it is a condition that must hold within (in the interval $(t_s(e), t_f(e))$ of) some execution e of p in order for e to be successful. This is because the procedure adds summary inconditions to p for all conditions of all of p's subplans except those that can only hold at $t_s(e)$ or $t_f(e)$. It does this in three substeps. First, the procedure adds a summary condition for each summary incondition c' of one of p's subplans p'. $\ell(c')$ must hold within some execution e' of p' by the inductive hypothesis, and the inconditions of p' must hold within e of p according to the semantics of subplan executions. In the [second, third] substep, an incondition c is added for each summary [pre, post] condition c' for each subplan p' that is not always a [$first$, $last$] summary [pre, post] condition of p. Therefore, there is some execution e' of p' where $\ell(c')$ holds within e' and, thus, must also hold within some execution e of p. So, the only conditions of the subplans that are not inconditions of p are those that hold at $t_s(e)$ or $t_f(e)$ for all executions e of p. The $existence$ of c is set to *must* iff $\ell(c)$ must hold at some point within all executions e of p since either $\ell(c)$ is specified as an incondition of p; c is derived from a *must* summary incondition of some subplan p' of p (in which case it must hold in every execution of p' by the inductive hypothesis and, thus, in every execution e of p because $d(e)$ contains an execution of every subplan of p); or c is derived from a *must* summary [pre, post] condition c' of a subplan p' that is always not [$first$, $last$]–thus, $\ell(c')$ must hold within all executions of p' and p. Otherwise, the $existence$ of c is set to *may*. The procedure sets $timing(c)$ to $always$ iff $\ell(c)$ must hold throughout all executions of p since it requires that $\ell(c)$ is in an $always$ summary incondition of each subplan; $\ell(c)$ must hold throughout the executions of each subplan by the inductive hypothesis; the intervals within the executions of the subplans together must cover the interval within the execution of p according to $order(p)$; and there are no intervals in any execution of p where there are no subplans executing. Otherwise, $timing$ is set to $sometimes$.

Case 2—$type(p) = or$:

The step of the procedure for adding summary [pre, post] conditions for an *or* plan only adds conditions for the external [pre, post] conditions of p because there is a one-to-one matching of the summary [pre, post] conditions of every subplan p' to the set external [pre, post] conditions of p' by the inductive hypothesis, and there are no other subplans that could [achieve, undo] or clobber p''s summary [pre, post] conditions since the execution of an *or* plan p is only decomposed into an execution of one of p's subplans. The procedure sets the *existence* of the added summary [pre, post] condition c to *must* iff $\ell(c)$ is a *must* external [pre, post] condition of p because $\ell(c)$ must hold for all executions of every subplan for them to be successful by the inductive hypothesis, and an execution of p is only decomposed into an execution of one of p's subplans. Otherwise, the *existence* of c is set to *may*. The *timing* of c is set to $[first, last]$ iff $\ell(c)$ must hold at $[t_s, t_f]$ of some execution of the plan for it to be successful because there must be an execution e' of a subplan p' and a $[first, last]$ summary [pre, post] condition c' where $\ell(c') \Leftrightarrow \ell(c)$ and $\ell(c)$ must hold at $[t_s(e'), t_f(e')]$ and, thus, at $[t_s(e), t_f(e)]$ of some execution e of p by the inductive hypothesis and the semantics of *or* subplan executions. Otherwise, *timing* is set to *sometimes*.

The step of the procedure for adding summary inconditions for an *or* plan only adds conditions that must hold within some execution e of p for it to be successful since all of the conditions that must hold within the executions of the subplans are captured by their summary inconditions by the inductive hypothesis, and no other conditions of the subplans can hold within e because they are $first$ or $last$ and would only hold at $t_s(e)$ or $t_f(e)$ by the inductive hypothesis and semantics of *or* subplan executions. The procedure sets the *existence* of the added summary incondition c to *must* iff $\ell(c)$ must hold within all executions of p because it is a *must* summary incondition of all subplans and, therefore, must hold within the all of the subplans' executions by the inductive hypothesis and, thus, within all executions of p. Otherwise, the *existence* of c is set to *may*. The *timing* of c is set to *always* iff $\ell(c)$ must hold for any execution e of p throughout the interval $(t_s(e), t_f(e))$ since $\ell(c)$ must hold within all executions e' of each subplan throughout $(t_s(e'), t_f(e'))$ by the inductive hypothesis and, thus, throughout $(t_s(e), t_f(e))$ by the semantics of *or* subplan executions. Otherwise, *timing* is set to *sometimes*. $\Box$

APPENDIX D

Soundness and Completeness Proofs for *CanAnyWay* and *MightSomeWay*

These proofs show that the algorithm for determining that a set of summarized tasks under specified partial temporal constraints have the *CanAnyWay* property is sound and complete. The also show the algorithm determining that a set of totally temporally ordered tasks have the $\neg$*MightSomeWay* property is sound and complete. However, these proofs depend on others (given first in this appendix) that prove the circumstances under which summarized conditions will be met depending on whether summary conditions are must- or may-clobbered. These proofs are reference in Section 4.3.4 and Chapter 5.

Lemma Any precondition summarized by $c \in pre_{sum}(p)$ will be met in *no* histories $h \in H$ if there is a plan $p' \in P$ that must-clobber c, and all of the $\neg\ell(c)$ conditions summarized by the c' referred to in the definition of must-clobber are met in all $h \in H$.

Proof If p' must-clobber c, and the conditions summarized by c' are all met, then p' asserts $\neg\ell(c)$ before c is required to be met by the top-level execution e of p. Thus, any precondition summarized by c will be threatened by p'. We also know by the definition of must-clobber that there can be no other plan in P that achieves the precondition since $\ell(c)$ cannot be asserted between the assertion of $\neg\ell(c)$ and the time that c is required. Thus, any precondition summarized by c must be clobbered. $\square$

Lemma Assuming that the initial state of any $h \in H$ is such that it does not conflict with any preconditions external to the plans represented by P_{sum}, if any precondition summarized by $c \in pre_{sum}(p)$ will be met in *no* histories $h \in H$, then there is a plan $p' \in P$ that must-clobber c.

Proof Based on the assumption about the initial state, if all preconditions summarized by $c \in pre_{sum}(p)$ are not met, then they must be clobbered. The definition of must-clobber requires one or more plans to be responsible for asserting $\neg\ell(c)$ (summarized by c') in all histories. It could not be the case that some set of plans assert $\neg\ell(c)$ in only some histories and only together clobber c in all histories. If that were true, then the c' in each plan would not be a *must* condition, and there would be some set of plans P represented by P_{sum} and some history in H where no plan asserts $\neg\ell(c)$. Thus, there is a plan p' represented in P_{sum} that attempts to assert c' in all $h \in H$. The definition of must-clobber also requires that no plan attempt to assert some c'' where $\ell(c'') \Leftrightarrow \ell(c)$ between when c' is asserted and when c is required in any history. It could not be the case that there are multiple plans (p'_1 and p'_2) asserting $\neg\ell(c)$ in all histories; some plan p'' asserts c'' after p'_1 and before p'_2 asserts c' in some histories; and in other histories p'' asserts c'' after p'_2 and before p'_1 asserts c'. If this were true, then the plans would not have specific ordering constraints given in the *order* over P. This would mean that in some $h \in H$, p'' asserts c'' after both p'_1 and p'_2 assert c', and c is achieved and not clobbered. This contradicts our assumption that preconditions summarized by c are never met. Thus, there can be no such p'', and there must be a p' that must-clobber c. $\Box$

Lemma Assuming that the initial state of any history $h \in H$ is such that it does not conflict with any preconditions external to the plans represented by P_{sum}, any precondition summarized by $c \in pre_{sum}(p)$ will be met in *all* $h \in H$ if there is *no* plan $p' \in P$ that may-clobber c, or there is such a p', and the $\neg\ell(c)$ conditions summarized by c' referred to in the definition of may-clobber are clobbered in all $h \in H$.

Proof If there is no plan p' that may-clobber c, or the $\neg\ell(c)$ condition summarized by c' that the top-level execution of p' attempts to assert is clobbered, then either there is no plan p' that asserts $\neg\ell(c)$, or there is such a p' and also a p'' that achieves $\ell(c)$ by asserting it after p' asserts $\neg\ell(c)$ and before c is required. Thus, there can be no plan that clobbers a precondition summarized by c, and all such preconditions will be met in all $h \in H$. $\Box$

Lemma If any precondition summarized by $c \in pre_{sum}(p)$ is met in *all* $h \in H$, then there is *no* plan $p' \in P$ that may-clobber c, or there is such a p', and the $\neg\ell(c)$ conditions summarized by c' referred to in the definition of may-clobber are clobbered in all $h \in H$.

Proof If all preconditions summarized by c are met, then there could be no plan that clobbered any one of them. If it the case that a plan p' attempted to assert $\neg\ell(c)$, but the assertion failed in every history, then the $\neg\ell(c)$ condition was clobbered. Thus, if there is any history h where a precondition summarized by c is not met, there must be a p' whose top-level execution in $E(h)$ attempts to assert a c' where $\ell(c') \Leftrightarrow \neg\ell(c)$, and there is a $\neg\ell(c)$ condition summarized by c' that is not clobbered. Therefore, if all preconditions summarized by c are met in all $h \in H$, there can be no p' that may-clobber c, or the $\neg\ell(c)$ conditions summarized by c' are clobbered in all $h \in H$. □

Lemma An incondition summarized by $c \in in_{sum}(p)$ will *not* be met in all histories $h \in H$ if there is a plan $p' \in P$ that must-clobber c, and all of the $\neg\ell(c)$ conditions summarized by the c' referred to in the definition of must-clobber are met in all $h \in H$.

Proof If p' must-clobber c, and the conditions summarized by c' are all met, then p' asserts $\neg\ell(c)$ within e when c is required to be met since $always(c)$. Thus, in any $h \in H$ there must be some incondition summarized by c that is clobbered since c can only be *always* when the open interval of e is covered by plan executions with $\ell(c)$ inconditions that are summarized by c. □

Lemma If an incondition summarized by $c \in in_{sum}(p)$ is *not* met in all histories $h \in H$, then there is a plan $p' \in P$ that must-clobber c.

Proof If some incondition summarized by c is not met, then it must be clobbered. It could not be clobbered internal to p in all histories because, otherwise, there would be no summary condition added. c cannot be *sometimes* because there must be some history for some set of plans having the summary information P_{sum} where $\neg\ell(c)$ is not asserted within the interval where $\ell(c)$ is required. By similar argument to that presented in the previous lemma for must-clobbering summary preconditions, there must be a plan p' with execution e' that asserts $\neg\ell(c)$ within e in all $h \in H$. Thus, there must be a c' summarizing $\neg\ell(c)$ that e' attempts to assert within e in all $h \in H$, so p' must-clobber c. □

Lemma Any incondition summarized by $c \in in_{sum}(p)$ will be met in *all* $h \in H$ if there is *no* plan $p' \in P$ that may-clobber c.

Proof If there is no plan p' that may-clobber c, then there is no plan p' that asserts $\neg\ell(c)$ within e. Thus, there can be no plan that clobbers a incondition summarized by c, and all such inconditions will be met in all $h \in H$. □

Lemma If any incondition summarized by $c \in in_{sum}(p)$ is met in *all* $h \in H$, then either there is *no* plan $p' \in P$ that may-clobber c, or there is such a p', and the $\neg\ell(c)$ conditions summarized by c' referred to in the definition of may-clobber are clobbered in all $h \in H$.

Proof If all inconditions summarized by c are met, then there could be no plan that clobbered any one of them. If it the case that a plan p' attempted to assert $\neg\ell(c)$, then the assertion failed in every history because the $\neg\ell(c)$ condition was clobbered. Therefore, if all preconditions summarized by c are met in all $h \in H$, there can be no p' that may-clobber c, or the $\neg\ell(c)$ conditions summarized by c' are clobbered in all $h \in H$. □

Lemma A postcondition summarized by $c \in post_{sum}(p)$ will *not* be met in all histories $h \in H$ if there is a plan $p' \in P$ that must-clobber c, and all of the $\neg\ell(c)$ conditions summarized by the c' referred to in the definition of must-clobber are met in all $h \in H$.

Proof If p' must-clobber c, and the conditions summarized by c' are all met, then p' asserts $\neg\ell(c)$ when e attempts to assert c. Thus, in any $h \in H$ there must be some post-condition summarized by c that is clobbered and, thus, not met since $\neg\ell(c)$ is met at the same time the postcondition is required. □

Lemma If a postcondition summarized by $c \in post_{sum}(p)$ is *not* met in all histories $h \in H$, then there is a plan $p' \in P$ that must-clobber c.

Proof If some postcondition summarized by c is not met, then it must be clobbered. It could not be clobbered internal to p in all histories because, otherwise, there would be no summary condition added. By similar argument to that presented in the lemma for must-clobbering summary preconditions, there must be a plan p' with execution e' that asserts $\neg\ell(c)$ at the same time e attempts to assert c in all $h \in H$. Thus, there must be a c' summarizing $\neg\ell(c)$ that e' attempts to assert at the same time e attempts to assert c in all $h \in H$, so p' must-clobber c. □

Lemma Any postcondition summarized by $c \in post_{sum}(p)$ will be met in *all* $h \in H$ if there is *no* plan $p' \in P$ that may-clobber c.

Proof If there is no plan p' that may-clobber c, then there is no plan p' that asserts $\neg\ell(c)$ when e attempts to assert c. Thus, there can be no plan that clobbers a postcondition summarized by c, and all such postconditions will be met in all $h \in H$. □

Lemma If any postcondition summarized by $c \in post_{sum}(p)$ is met in *all* $h \in H$, then either there is *no* plan $p' \in P$ that may-clobber c, or there is such a p', and the $\neg\ell(c)$ conditions summarized by c' referred to in the definition of may-clobber are clobbered in all $h \in H$.

Proof If all postconditions summarized by c are met, then either there could be no plan that clobbered any one of them, or there is a plan p' that attempted to assert $\neg\ell(c)$, and the assertion failed in every history because the $\neg\ell(c)$ condition was clobbered. Therefore, if all postconditions summarized by c are met in all $h \in H$, there can be no p' that may-clobber c, or the $\neg\ell(c)$ conditions summarized by c' are clobbered in all $h \in H$. □

Theorem The algorithm for $CAW(order, P_{sum})$ is sound and complete.

Proof If the algorithm returns $true$, then there is no p' that may-clobber some c. According to the lemmas above, all preconditions summarized by the summary preconditions, inconditions summarized by summary inconditions, and postconditions summarized by summary postconditions must be met in all $h \in H$ for all sets of plans P whose summary information is P_{sum}. This means that all of the external preconditions, external postconditions, and inconditions of all executions and subexecutions are met. Since we assume that the plans in P cannot clobber their own conditions, we also know that the internal (non-external) pre- and postconditions are also met. Thus, all of the conditions in all of the executions and subexecutions in $E(h)$ are met, so all executions succeed.

If all executions succeed for all $h \in H$ for all sets of plans P whose summary information is P_{sum}, then no conditions were clobbered. According to the lemmas above, there can be no p' that may-clobber some summary condition c unless the execution of p' fails when it attempts to assert a c' that summarizes conditions that would otherwise clobber $\ell(c)$. But, such a p' cannot exist because then there would be some plan that clobbered a condition of p', and we assume that no conditions were clobbered. □

Theorem The algorithm for $\neg MSW(order, P_{sum})$ is sound.

Proof If the algorithm returns $true$, then there is a p' that must-clobber some summary condition c of a plan $p \in P$ whose summary information is $p_{sum} \in P_{sum}$. According to the lemmas above, any conditions summarized by c will be met in *no* histories $h \in H$. Thus, all executions of p will fail in all $h \in H$. □

Theorem The algorithm for $\neg MSW(order, P_{sum})$ is complete when *order* specifies a total order on P_{sum}.[1]

[1] A total ordering is one where each endpoint of a plan's execution interval is constrained to one temporal relation (precedes, follows, or same) with every other endpoint of every other plan's execution.

Proof If some condition ℓ of the execution of p is clobbered in some histories but not in others, then it is because the condition is not required to be met in all histories, or a clobbering condition is not asserted in all histories. If the plans in P are totally ordered, there is no temporal uncertainty, so the summary condition c of p summarizing ℓ is *may*, or any c' of p' summarizing a condition clobbering ℓ is *may*. Thus, p' may-clobber c of p, but must-clobber is false.

Now, if in each $h \in H$ some execution of some plan in P fails, but no particular plan's execution fails in all histories, then each failing plan has a condition that is clobbered in some histories and not in others, as just described. So, for each of these failing plans, the summary condition c summarizing the failed condition is *may*, or the clobbering plans have *may* summary conditions c' that are involved in the clobbering. But given only the summary information for these plans, there are other sets of plans with identical summary information where there is some history h where none of the conditions summarized by c are clobbered since either c or c' is *may*, meaning that they may not be required or asserted in h. Thus, there is no set of summarized plans P_{sum} such that in each $h \in H$ some execution of some plan in P fails, but no particular plan's execution fails in all histories for all sets of plans with summary information P_{sum}.

Therefore, if in each $h \in H$ some execution of some plan in P fails, there does exist a particular plan $\in P$ whose execution fails for all $h \in H$. This means that there must be a p' that must-clobber some c. Since $\neg MSW$ returns true if the algorithm for must-clobber returns true for any such p' and c, and the algorithms for must-clobber are complete, the algorithm for $\neg MSW$ is complete. □

APPENDIX E

THREAT RESOLUTION is NP-complete

The following proof is provided to clarify complexity results in Section 6.3.2.

Theorem THREAT RESOLUTION is NP-complete. This is the problem of determining whether there is a set of ordering constraints that can be added to a partially ordered STRIPS plan such that no operator's preconditions are threatened by another operator's effects.

Proof If there is a set of ordering constraints that will resolve all threats, then there is at least one corresponding total order where there are no threats. Thus, the problem is in NP since orderings of operators can be chosen non-deterministically, and threats can be identified in polynomial time.

For a directed graph $G = (V, E)$ with nodes $v_1, v_2, \ldots, v_n \in V$ and edges $e_1, e_2, \ldots, e_m \in E$ (a set of ordered pairs of nodes), HAMILTONIAN PATH is the problem that asks if there is a path that visits each node exactly once. Build an instance of THREAT RESOLUTION (a partial order plan) by creating an operator for each node v_i. The only precondition of the operator is $A(i)$, representing the *accessibility* of the node. There is a postcondition $A(j)$ for each edge $e_k = (v_i, v_j)$, and a postcondition $\overline{A(l)}$ for all other nodes for which there is no edge from v_i. All operators are unordered, and the initial state and goal state is empty.

If there is a Hamiltonian path for the graph, then the operators for the nodes can be ordered the same as the nodes in the path because the accessibility preconditions of each operator will be satisfied by the previous operator. If there is no Hamiltonian path for the graph, then there is no consistent ordering of the operators. We know this because there is a one-to-one mapping from an ordering of nodes to an ordering of operators. If the ordering of the nodes is such that there is no edge from one to a succeeding node, then the accessibility precondition of the corresponding operator will be clobbered. In addition, for any walk through the graph, there eventually will be an unvisited node for which there is no edge from the last node visited. In this case, the unvisited node will be clobbered because its accessibility precondition will not be met. Thus, THREAT RESOLUTION is NP-hard, and since it was shown to be in NP, it is NP-complete. □

In order to show that resolving threats among CHiPs is also NP-complete, one only needs to add inconditions to each operator that prevent concurrent action. This is done by adding $A(i)$ for v_i and $\overline{A(j)}$ for every other $v_j \in V$ to the inconditions of the operator corresponding to v_i for each $v_i \in V$. This ensures that the only temporal relations that can hold between any pair of operators are $before$, $after$, $meets$, or $imeets$, and the one-to-one mapping from paths in the graph to sequences of operators is preserved.

BIBLIOGRAPHY

BIBLIOGRAPHY

[Allen and Koomen, 1983] J. F. Allen and J. A. Koomen. Planning using a temporal world model. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 741–747, 1983.

[Allen *et al.*, 1991] J. Allen, H. Kautz, R. Pelavin, and J. Tenenberg. *Reasoning about plans*. Morgan Kaufmann, 1991.

[Allen, 1983] J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, November 1983.

[Bylander, 1994] T. Bylander. The computational complexity of STRIPS planning. *Artificial Intelligence*, 69:165–204, 1994.

[Chien *et al.*, 2000a] S. Chien, R. Knight, A. Stechert, R. Sherwood, and G. Rabideau. Using iterative repair to improve the responsiveness of planning and scheduling. In *Proceedings of the International Conference on AI Planning and Scheduling*, pages 300–307, 2000.

[Chien *et al.*, 2000b] S. Chien, G. Rabideu, R. Knight, R. Sherwood, B Engelhardt, D. Mutz, T. Estlin, B. Smith, F. Fisher, T. Barrett, G. Stebbins, and D. Tran. Automating space mission operations using automated planning and scheduling. In *Proc. SpaceOps*, 2000.

[Corkill, 1979] D. Corkill. Hierarchical planning in a distributed environment. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 168–175, 1979.

[Currie and Tate, 1991] K. Currie and A. Tate. O-Plan: The open planning architecture. *Artificial Intelligence*, 52:49–86, 1991.

[Decker, 1995] K. Decker. *Environment centered analysis and design of coordination mechanisms*. PhD thesis, University of Massachusetts, 1995.

[Durfee and Montgomery, 1991] E. H. Durfee and T. A. Montgomery. Coordination as distributed search in a hierarchical behavior space. *IEEE Transactions of Systems, Man and Cybernetics*, 21(6):1363–1378, November 1991.

[Ephrati and Rosenschein, 1994] E. Ephrati and J. Rosenschein. Divide and conquer in multi-agent planning. In *Proceedings of the National Conference on Artificial Intelligence*, pages 375–380, July 1994.

[Erol *et al.*, 1994a] K. Erol, J. Hendler, and D. Nau. Semantics for hierarchical task-network planning. Technical Report CS-TR-3239, University of Maryland, 1994.

[Erol *et al.*, 1994b] K. Erol, D. Nau, and J. Hendler. UMCP: A sound and complete planning procedure for hierarchical task-network planning. In *Proceedings of the International Conference on AI Planning and Scheduling*, June 1994.

[Fagin *et al.*, 1995] R. Fagin, J. Halpern, Y. Moses, and M. Vardi. *Reasoning about knowledge*. MIT Press, 1995.

[Fikes and Nilsson, 1971] R. Fikes and N. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[Firby, 1989] J. Firby. *Adaptive Execution in Complex Dynamic Domains*. PhD thesis, Yale University, 1989.

[Gat, 1998] E. Gat. On three-layer architectures. *Artificial Intelligence and Mobile Robots, AAAI Press*, 1998.

[Georgeff and Lansky, 1986] M. P. Georgeff and A. Lansky. Procedural knowledge. *Proceedings of IEEE*, 74(10):1383–1398, October 1986.

[Georgeff, 1983] M. P. Georgeff. Communication and interaction in multiagent planning. In *Proceedings of the National Conference on Artificial Intelligence*, pages 125–129, 1983.

[Georgeff, 1984] M. P. Georgeff. A theory of action for multiagent planning. In *Proceedings of the National Conference on Artificial Intelligence*, pages 121–125, 1984.

[Grid, 1999] DARPA control of agent based systems program. http://coabs.globalinfotek.com, 1999.

[Grosz and Kraus, 1996] B. Grosz and S. Kraus. Collaborative plans for complex group action. *Artificial Intelligence*, 86:269–358, 1996.

[Hammond, 1986] K. Hammond. CHEF: A model of case-based planning. In *Proceedings of the National Conference on Artificial Intelligence*, pages 267–271, 1986.

[Horty and Pollack, 2001] J. Horty and M. Pollack. Evaluating new options in the context of existing plans. *Artificial Intelligence*, 127(2):199–220, 2001.

[Huber, 1999] M. Huber. JAM: A BDI-theoretic mobile agent architecture. In *Proceedings Internation Conference on Autonomous Agents*, pages 236–243, 1999.

[Knight *et al.*, 2000] R. Knight, G. Rabideau, and S. Chien. Computing valid intervals for collections of activities with shared states and resources. In *Proceedings of the International Conference on AI Planning and Scheduling*, pages 600–610, 2000.

[Knight *et al.*, 2001] R. Knight, G. Rabideau, and S. Chien. Extending the representational power of model-based systems using generalized timelines. In *Proceedings of the Sixth International Symposium on Artificial Intelligence, Robotics, and Automation in Space*, 2001.

[Knoblock, 1991] C. Knoblock. Search reduction in hierarchical problem solving. In *Proceedings of the National Conference on Artificial Intelligence*, pages 686–691, 1991.

[Korf, 1987] R. Korf. Planning as search: A quantitative approach. *Artificial Intelligence*, 33:65–88, 1987.

[Laborie and Ghallab, 1995] P. Laborie and M. Ghallab. Planning with sharable resource constraints. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1643–1649, 1995.

[Lansky, 1990] A. Lansky. Localized search for controlling automated reasoning. In *Proceedings of the DARPA Workshop on Innovative Approaches to Planning, Scheduling and Control*, pages 115–125, November 1990.

[Lee *et al.*, 1994] J. Lee, M. J. Huber, E. H. Durfee, and P. G. Kenny. UMPRS: An implementation of the procedural reasoning system for multirobot applications. In *Proceedings of the AIAA/NASA Conference on Intelligent Robotics in Field, Factory, Service, and Space*, pages 842–849, March 1994.

[McAllester and Rosenblitt, 1991] D. McAllester and D. Rosenblitt. Systematic nonlinear planning. In *Proceedings of the National Conference on Artificial Intelligence*, pages 634–639, 1991.

[Meiri, 1992] I. Meiri. *Temporal Reasoning: A Constraint-Based Approach*. PhD thesis, University of California, Los Angeles, 1992.

[Muscettola, 1994] N. Muscettola. HSTS: Integrating planning scheduling. *Intelligent Scheduling*, pages 169–212, 1994.

[Papadimitriou and Steiglitz, 1998] C. Papadimitriou and K. Steiglitz. *Combinatorial Optimization - Algorithms and Complexity*. Dover Publications, New York, 1998.

[Pappachan, 2001] P. Pappachan. *Coordinating Plan Execution in Dynamic Multiagent Environments*. PhD thesis, University of Michigan, Ann Arbor, 2001.

[Pollack and Horty, 1999] M. Pollack and J. Horty. There's more to life than making plans: Plan management in dynamic, multi-agent environments. *AI Magazine*, 20(4):71–84, 1999.

[Pynadath *et al.*, 1999] D. Pynadath, M. Tambe, N. Cauvat, and L. Cavedon. Toward team-oriented programming. In *Proceedings of the Workshop on Architectures, Theories, and Languages*, 1999.

[Rabideu *et al.*, 1999] G. Rabideu, R. Knight, S. Chien, A. Fukunaga, and A. Govindjee. Iterative repair planning for spacecraft operations in the ASPEN system. In *Proceedings of the International Symposium on Artificial Intelligence, Robotics, and Automation in Space*, 1999.

[Rao and Georgeff, 1995] A. S. Rao and M. P. Georgeff. BDI-agents: From theory to practice. In *Proceedings of the First International Conference on Multiagent Systems*, San Francisco, 1995.

[Rathmell, 2001] R. Rathmell. A coalition force scenario: Binni - gateway to the golden bowl of Africa. http://www.aiai.ed.ac.uk/project/coalition/binni/, 2001.

[Russell and Norvig, 1995] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, New Jersey, 1995.

[Sacerdoti, 1974] E. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5(2):115–135, 1974.

[Sacerdoti, 1977] E. D. Sacerdoti. *A Structure for Plans and Behavior*. Elsevier-North Holland, 1977.

[Shoham and Tennenholtz, 1992] Y. Shoham and M. Tennenholtz. On the synthesis of useful social laws for artificial societies. In *Proceedings of the National Conference on Artificial Intelligence*, pages 276–281, 1992.

[Tambe, 1997] M. Tambe. Towards flexible teamwork. *Journal of Artificial Intelligence Research*, 7:83–124, 1997.

[Tate, 1977] A. Tate. Generating project networks. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 888–893, 1977.

[Tsamardinos *et al.*, 2000] I. Tsamardinos, M. Pollack, and J. Horty. Merging plans with quantitative temporal constraints, temporally extended actions, and conditional branches. In *Proceedings of the International Conference on AI Planning and Scheduling*, pages 264–272, 2000.

[Tsuneto *et al.*, 1997] R. Tsuneto, J. Hendler, and D. Nau. Space-size minimization in refinement planning. In *Proceedings of the Fourth European Conference on Planning*, 1997.

[Tsuneto *et al.*, 1998] R. Tsuneto, J. Hendler, and D. Nau. Analyzing external conditions to improve the efficiency of HTN planning. In *Proceedings of the National Conference on Artificial Intelligence*, pages 913–920, 1998.

[Vilain and Kautz, 1986] Vilain and H. Kautz. Constraint propagation algorithms for temporal reasoning. In *Proceedings of the National Conference on Artificial Intelligence*, pages 377–382, 1986.

[Weld, 1994] D. Weld. An introduction to least commitment planning. *AI Magazine*, 15(4):27–61, 1994.

[Weld, 1999] D. Weld. Recent advances in AI planning. *AI Magazine*, 20(2):93–123, 1999.

[Yang and Chan, 1994] Q. Yang and A. Chan. Delaying variable binding commitments in planning. In *Proceedings of the International Conference on AI Planning and Scheduling*, pages 182–187, 1994.

[Yang, 1990] Q. Yang. Formalizing planning knowledge for hierarchical planning. *Computational Intelligence*, 6(1):12–24, February 1990.

[Yang, 1997] Q. Yang, editor. *Intelligent Planning: A Decomposition and Abstraction Based Approach*. Springer, 1997.

[Yokoo and Hirayama, 1998] M. Yokoo and K. Hirayama. The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 10(5):673–685, 1998.

[Young *et al.*, 1994] M. Young, M. Pollack, and J. Moore. Decomposition and causality in partial-order planning. In *Proceedings of the International Conference on AI Planning and Scheduling*, pages 188–193, 1994.

[Zilberstein and Russell, 1992] S. Zilberstein and S. Russell. Efficient resource-bounded reasoning in AT-RALPH. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 260–266, 1992.