Using Abstraction to Coordinate Multiple Robotic Spacecraft

Bradley J. Clement, Anthony C. Barrett, Gregg R. Rabideau
Jet Propulsion Laboratory
4800 Oak Grove Drive, M/S 126-347
Pasadena, CA 91109-8099
{bclement, barrett, rabideau}@aig.jpl.nasa.gov

Edmund H. Durfee
University of Michigan
1101 Beal Avenue
Ann Arbor, MI 48109-2110
durfee@umich.edu

Abstract

The trend toward multiple-spacecraft missions requires autonomous teams of spacecraft to coordinate their activities when sharing limited resources. This paper describes how an iterative repair planner/scheduler can reason about the activities of multiple spacecraft at abstract levels in order to greatly improve the scheduling of their use of shared resources. By finding consistent schedules at abstract levels, refinement choices can be preserved for use in robust plan execution systems. We present an algorithm for summarizing the metric resource requirements of an abstract activity based on the resource usages of its potential refinements. We find that reasoning about this summary information and that of state constraints can offer exponential improvements in the time to find consistent schedules with an iterative repair planner. We analytically describe the conditions under which these improvements are made and show that sometimes the extra overhead involved does not warrant their use. We apply these techniques within the ASPEN planner/scheduler to a domain where a team of rovers must coordinate their schedules to avoid conflicts over shared resources. Experiments using the ASPEN planner/scheduler in a Mars multi-rover domain support our analyses and compare techniques for controlling decomposition.

1 Introduction

Autonomous spacecraft have recently used onboard planning and execution in order to improve the efficiency of exploration by reducing explicit remote control. However, a trend toward multiple-spacecraft missions requires autonomous teams of spacecraft to collectively plan and execute for goals that arise. In order to plan for coordinated execution, the spacecraft need to reason about the population's concurrent execution to detect and resolve conflicts among the individual spacecrafts' plans. For many applications, reasoning about a planning problem at multiple levels of abstraction enhances planning and scheduling efficiency.

In an effort to reason about actions at abstract levels, Hierarchical Task Network (HTN) planners [4] represent abstract plan operators that decompose into choices of action sequences that may also be abstract. These planners exploit domain knowledge to reduce the space of plans they generate. A domain expert can intuitively encode hierarchies of plan operators to guide the planner in building an ordering of actions that achieves abstract tasks, goals, and subgoals. Instead of building a plan from the beginning forward (or end backward), the planner incrementally refines abstract operators to eventually converge on specific actions that achieve the higher level goals. By structuring the refinement of 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 can still guarantee completeness by carefully structuring the abstract plan operators. The domain expert has this same responsibility when crafting operators for non-hierarchical planners.

Previous research [10; 13; 9] has shown that, under certain restrictions, hierarchical refinement search can reduce the search space by an exponential factor. Recent research has shown that the performance of hierarchical planners can be greatly improved without these restrictions by reasoning during refinement about the conditions embodied by abstract plan operators [2; 3]. These *summarized conditions* represent the internal and external requirements and effects of the abstract operator and those of the children in its decomposition. Using this information, a planner can resolve conflicts at abstract levels and sometimes can find abstract solutions or determine that particular decomposition choices are inconsistent. The domain expert can derive these summary conditions for each abstract operator in the domain offline before planning problems are encountered. Reasoning about this *summary information* can exponentially reduce the cost of finding a first solution and optimal solutions [3].

Iterative repair planners commonly use scheduling and constraint satisfaction techniques for handling large numbers of activities and metric resources. We extend summary information to additionally include a representation for summarizing the metric resource usages of an abstract activity's decompositions. We demonstrate the benefits of abstraction in ASPEN [1] (an iterative repair planner/scheduler) by using algorithms and techniques for effectively reasoning about this summary information.

While planning efficiency is a major focus of this ap-

This work was performed at the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration. It was also supported in part by DARPA (F30602-98-2-0142).

proach, another is the support of flexible plan execution systems such as PRS [6], UMPRS [11], RAPS [5], JAM [7], etc., that similarly exploit hierarchical plan spaces. Rather than refine abstract plan operators into a detailed end-to-end plan, however, 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 is possible for HTN planning (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. Using summary information, a planner can resolve conflicts at abstract levels while preserving refinement choices underneath that can be used in robust execution systems to handle unexpected or unknown events and to provide some ability to recover from failure.

This research makes the following contributions:

- An algorithm summarizing metric resource usages for abstract activities;
- Complexity analyses showing that schedule operations are exponentially cheaper at higher levels of abstraction when summarizing activities results in fewer resource, state, and temporal constraints;
- Experiments in a multi-rover domain that show that summary information enables a planner to find solutions (consistent schedules) more quickly under the conditions reported in the complexity analyses and that reveal conditions under which summary information can introduce unnecessary overhead;
- An empirical comparison of search techniques for directing the refinement of abstract activities based on conflicts over summarized constraints, showing how summary information can further improve performance in finding solutions.

2 Representations

To illustrate our approach, we will focus on managing 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 1, where vertices denote distinguished locations, and edges denote allowed paths. While some paths are over hard ground, others are over loose sand where traversal is harder since a rover can slip.

2.1 Resources and Tasks

More formally, we represent each rover's status in terms of state and resource variables. The values in state variables record the status of key rover subsystems. For instance, a rover's *position* state variable can take on the label of any

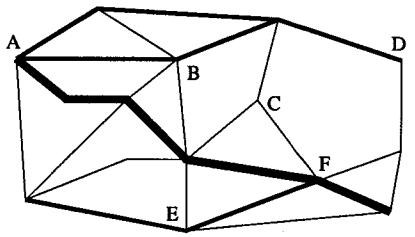

Figure 1: Example map of established paths between points in a rover domain, where thinner edges are harder to traverse, and labeled points have associated observation goals

vertex in the location network. Given this representation of state information, tasks have preconditions/effects that we represent as equality constraints/assignments. In our rover example traveling on the arc from point A to point B is done with a *go(A,B)* task. This task has the precondition $(position=A)$ and the effect $(position=B)$.

In addition to interacting with state variables, tasks use resources. While some resources are only used during a task, using others persists after a task finishes. The first type of resource is *nondepletable*, with examples like solar power which immediately becomes available after some task stops using it. On the other hand, battery energy is a *depletable* resource because its consumption persists until a later task recharges the battery. We model a task's resource consumption by subtracting the usage amount from the resource variable when the task starts and for nondepletable resources adding it back upon completion. While this approach is simplistic, it can conservatively approximate any resource consumption profile by breaking a task into smaller subtasks.

Primitive tasks affect state and resource variables, and an abstract task is a non-leaf node in an AND/OR tree of tasks.[1] An AND task is executed by executing all of its subtasks according to a some set of specified temporal constraints. An OR task is executed by executing one of its subtasks. Figure 2 gives an example of such an abstract task. Imagine a rover that wants to make an early morning trip from point A to point B on our 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 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. As the *take X path* branches suggest, there can be a number of temporal and other constraints between tasks in a branch.

2.2 Summary Information

An abstract task's state variable summary information includes elements for pre-, in-, and postconditions. Summary preconditions are conditions that must be met by the initial state or previous external tasks in order for a task to decompose and execute successfully, and a task's summary post-

[1] It is trivial to extend the algorithms in this paper to handle state and resource constraints specified for abstract tasks.

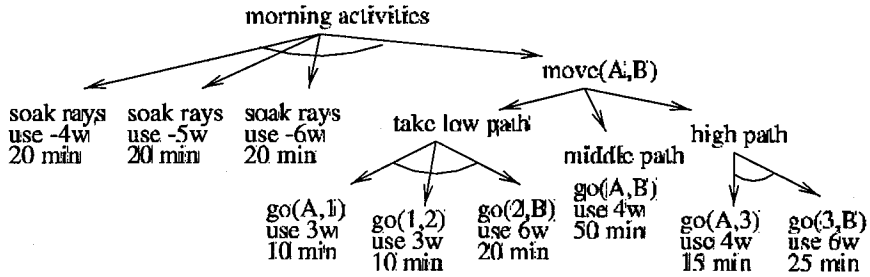

Figure 2: AND/OR tree defining abstract tasks and how they decompose for a morning drive from point A to point B along one of the three shortest paths in our example map, where 1, 2, and 3 denote unlabeled waypoints

conditions are the effects of its decomposition's execution that are not undone internally. We use summary inconditions for those conditions that are required or asserted in the task's decomposition during the interval of execution. All summary conditions are used to reason about how state variables are affected while performing an abstract task, and they have two orthogonal types of modalities:

- *must* or *may* indicates that a condition holds in all or some decompositions of the abstract task respectively and
- *first*, *last*, *sometimes*, or *always* indicates when a condition holds in the task's execution interval.

For instance, the $move(A, B)$ task in our example has a *must*, *first*(*position*=A) summary precondition and a *must*, *last*(*position*=B) postcondition because all decompositions move the rover from A to B. Since the $move(A, B)$ task decomposes into one of several paths, it has summary inconditions of the form *may*, *sometimes*(*position*=i), where i is 1, 2 or 3. State summary conditions are formalized in [2].

Extending summary information to include metric resources involves defining a new representation and algorithm for summarization. A *summarized resource usage* consists of ranges of potential resource usage amounts during and after performing an abstract task, and we represent this summary information 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 an task with multiple decomposition choices and timing choices among loosely constrained subtasks. For example, the *high path* task has a $\langle [4,4], [6,6], [0,0] \rangle$ 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 $\langle [0,4], [4,6], [0,0] \rangle$ summary power use over a 50 minute interval. In both cases the $persist_range$ is $[0,0]$ because 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 we want 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.

3 Resource Summarization Algorithm

The state summarization algorithm [2] recursively propagates summary conditions upwards from an AND/OR tree's leaves, and the algorithm for resource summarization takes the same approach. Starting at the leaves, we find 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 we either come to an AND or an OR branch. For an OR branch the combined summary usage comes from the OR computation

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

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 0 resource usage during the extension. For instance, when we determine the resource usage for $move(A, B)$ we combine two 40 minute tasks with a 50 minute task. The resulting summary information is for 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. Our *take x path* examples illustrate the simplest subcase, where subtasks are tightly constrained to execute serially. Here profiles are appended together, and the resulting summary usage information comes form the SERIAL-AND computation

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

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

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

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, we consider all legal orderings of child task endpoints. For example, in our 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 3 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, we 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. Finally, the AND task's summary is computed by combining the summaries for all possible orderings using an OR computation.

Here we 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[2]:

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

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

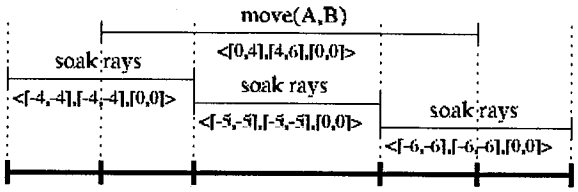

Figure 3: Possible task ordering for a rover's morning activities, with resulting subintervals.

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 our 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 3, 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 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 Using Summary Information

In this section, we describe techniques for using summary information in local search planners to reason at abstract levels effectively and discuss the complexity advantages. Reasoning about abstract plan operators using summary information can result in exponential planning performance gains for backtracking hierarchical planners [3]. In iterative repair planning, a technique called *aggregation* that involves scheduling hierarchies of tasks similarly outperforms

the movement of tasks individually [8]. But, can summary information be used in an iterative repair planner to improve performance when aggregation is already used? We demonstrate that summarized state and resource constraints makes exponential improvements by collapsing constraints at abstract levels. First, we describe how we use aggregation and summary information to schedule tasks within an iterative repair planner. Next, we analyze the complexity of moving abstract and detailed tasks using aggregation and summary information. Then we describe how a heuristic iterative repair planner can exploit summary information.

4.1 Aggregation and Summary Information

While HTN planners commonly take a generative least commitment approach to problem solving, research in the OR community illustrates that a simple local search is surprisingly effective [12]. Heuristic iterative repair planning uses a local search to generate a plan. It starts with an initial flawed plan and iteratively chooses a flaw, chooses a repair method, and changes the plan by applying the method. Unlike generative planning, the local search never backtracks. The repair methods can add, change, and remove features from the current plan. Since taking a random walk through a large space of plans is inefficient, heuristics guide the choices by determining the probability distributions for each choice. We build on this approach to planning by using the ASPEN planner [1].

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 siblings and subtasks is shown to be much less efficient [8]. Valid placements of the task hierarchy in the schedule are computed from the state and resource usage profile for the hierarchy. This profile represents one instantiation of the decomposition and temporal ordering of the abstract task's hierarchy.

A summarized state or resource usage represents all potential profiles of an abstract task before it is decomposed. Our approach involves reasoning about summarized constraints in order to schedule abstract tasks before they are decomposed. 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 used to move partially elaborated hierarchies based on their summarized constraints.

4.2 Complexity Analysis

To move a hierarchy of tasks using aggregation, valid intervals must be computed for each resource variable affected by the hierarchy.[3] These valid intervals are intersected for the valid placements for the abstract tasks and their children. The complexity of computing the set of valid intervals for a resource is $O(aC)$ where a 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 [8]. If there are 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 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. We 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 resource, 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 on detailed tasks.

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. Experiments in Section 5 exhibit great improvement for cases when tasks have constraints over com-

[3] The analysis also applies to state constraints, but we restrict our discussion to resource usage constraints for simplicity.

mon resource variables.

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, $O(b^{d-i})$ scheduling operations may be avoided. In Section 5, 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 [10; 9]. 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. We do not make this assumption for our problems. However, the speedup described above does assume that the hierarchies need not be fully expanded to find solutions.

4.3 Decomposition Heuristics for Iterative Repair

Despite this optimistic complexity, 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, we describe techniques for reasoning about summary information as abstract tasks are detailed.

We explored two approaches that reason about tasks from the top-level of abstraction down in the manner described in [3]. 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 we 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.

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) [3] heuristic that expands (decomposes) tasks involved in more conflicts before others. (Thus, we later will refer to this heuristic as EMTF.) This heuristic avoids unnecessary reasoning about the details of non-conflicting tasks. This is similar to a most-constrained variable heuristic often employed in constraint satisfaction problems. Experiments in Section 5 suggest that EMTF performs better than level-decomposition, but only when EMTF uses decomposition rates suited for the problem domain.

Another heuristic for improving planning performance prefers decomposition choices that lead to fewer conflicts. In effect, this is a least-constraining value heuristic used in constraint satisfaction approaches. 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.[4] This is the *fewest-threats-first* (FTF) heuristic that is shown to be effective in pruning the search space in a backtracking planner [3]. Likewise, the experiments in Section 5 report that using FTF can find solutions much more quickly when decomposition choices cause significantly varying numbers of conflicts.

5 Empirical Comparisons

The experiments we describe here show that, for our chosen domain, 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, we find cases where abstract reasoning incurs significant overhead when solutions are only found at deeper levels. However, in domains where decomposition choices are critical, we show that this overhead is insignificant because the FTF heuristic finds solutions at deeper levels with better performance. These experiments also show that the EMTF heuristic outperforms level-decomposition for certain decomposition rates. In addition, we 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 4.2 claiming that more constraints at deeper levels exponentially complicate the scheduling problem.

The domain for our problems expands the single rover problem described in earlier sections to a team of rovers that must resolve conflicts over shared resources. 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. In addition,

[4] Or, in stochastic planners like ASPEN, the children are chosen with probability decreasing with their respective number of conflicts.

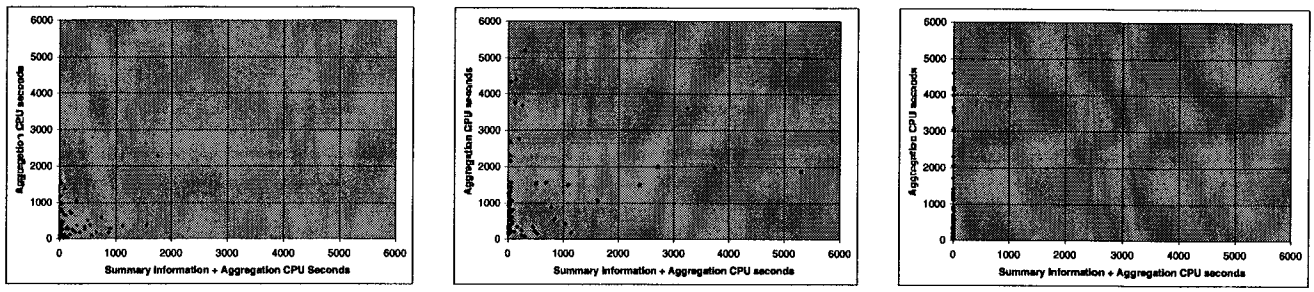

Figure 4: Plots for the *no channel*, *mixed*, and *channel only* domains

rovers must communicate with the lander for telemetry using a shared channel of fixed bandwidth. Depending on the terrain, the required bandwidth varies. 80 problems were generated for two to five rovers, three to six observation locations per rover, and 9 to 105 waypoints. In general problems that contain fewer waypoints and more observation goals are more difficult because there are more interactions among the rovers. Schedules ranged from 180 to 1300 tasks. Note that we use a prototype interface for summary information, and some of ASPEN's optimized scheduling techniques could not be used.

We compare ASPEN using aggregation with and without summarization for three variations of the domain. The use of summary information includes 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, we expect greater improvement in performance when using summary information according to the complexity analyses in Section 4.2. Tasks within a rover's hierarchy rarely place constraints on other variables more than once, so the *no channel* domain corresponds to the case where summarization collapses no constraints.

Figure 4 (left) 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). We find that for these problems finding a solution requires the planner to dig deep into the rovers' hierarchies, and once it decomposes the hierarchies to these levels, 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. Previous work shows that this overhead is rarely significant in backtracking planners because summary information can prune inconsistent search spaces at abstract levels [3]. However, in non-backtracking planners like ASPEN, the only opportunity we found to prune the search space at abstract levels was using the FTF heuristic to avoid greater numbers of conflicts in particular branches. Later, we will explain why FTF is not helpful for this domain but very effective in a modified domain.

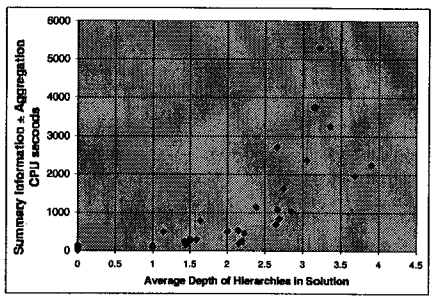

Figure 5: CPU time for solutions found at varying depths.

Figure 4 (middle) shows significant improvement for summary information in the *mixed* domain compared to the *no channel* domain. Adding the channel resource rarely affected the use of summary information because the collapse in summary constraints incurred insignificant additional complexity. However, the channel resource made the scheduling task noticeably more difficult for ASPEN when not using summary information. In the *channel only* domain (Figure 4 right), 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 Section 4.2 that argues that summary information exponentially improves performance when tasks within the same hierarchy make constraints over the same resource and solutions are found at some level of abstraction.

Figure 5 shows the CPU time required for ASPEN using summary information 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, supporting our claim that finding a solution at higher levels is exponentially easier because of an exponential increase in the number of constraints at lower levels.

For the described domain, choosing different paths to an observation location usually does not make a significant difference in the number of conflicts encountered because if the rovers cross paths, all path choices will still lead to conflict. We created a new set of problems where obstacles force the rovers to take paths through corridors that have no connection to others paths. For these 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

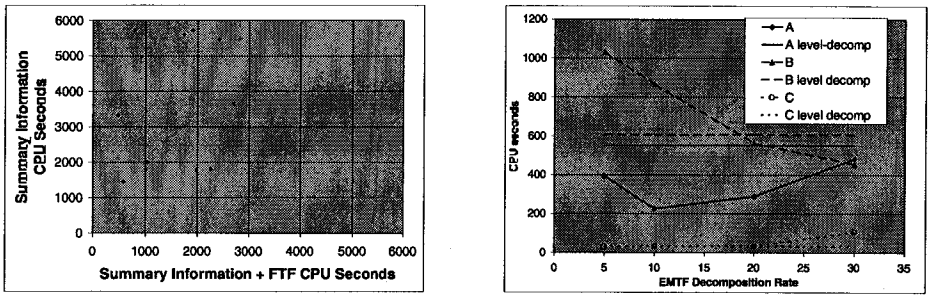

Figure 6: Performance using FTF and EMTF vs. level-decomposition heuristics.

that causes one. The planner using the FTF heuristic dominates the planner choosing decompositions randomly for all but two problems (Figure 6 left).

Figure 6 (right) 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. Our future work will include investigating the relation of decomposition rates to performance based on problem structure.[5]

6 Conclusion

Reasoning about abstract constraints exponentially accelerates finding schedules when constraints collapse during summarization, and solutions at some level of abstraction can be found. Similar speedups occur when decomposition branches result in varied numbers of conflicts. The offline algorithm for summarizing metric resource usage makes these performance gains available for a larger set of expressive planners and schedulers. We have shown how these performance advantages can improve ASPEN's effectiveness when scheduling the tasks of multiple spacecraft. The use of summary information also enables a planner to preserve decomposition choices that robust execution systems can use to handle some degree of uncertainty and failure. Our future work includes evaluating the tradeoffs of optimizing plan quality using this approach as well as developing protocols to allow multiple spacecraft planners to coordinate their tasks asynchronously during execution.

References

[1] 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.

[2] B. Clement and E. Durfee. Theory for coordinating concurrent hierarchical planning agents. In *Proc. AAAI*, pages 495–502, 1999.

[3] B. Clement and E. Durfee. Performance of coordinating concurrent hierarchical planning agents using summary information. In *Proc. ATAL*, pages 202–216, 2000.

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

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

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

[7] M. Huber. Jam: a bdi-theoretic mobile agent architecture. In *Proc. Intl. Conf. Autonomous Agents*, pages 236–243, 1999.

[8] R. Knight, G. Rabideau, and S. Chien. Computing valid intervals for collections of activities with shared states and resources. In *Proc. AIPS*, pages 600–610, 2000.

[9] C. Knoblock. Search reduction in hierarchical problem solving. In *Proc. AAAI*, pages 686–691, 1991.

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

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

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

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

[5] For other experiments, we used a decomposition rate of 20%.