JOURNAL OF AEROSPACE COMPUTING, INFORMATION, AND COMMUNICATION
Vol. 8, May 2011

Tractable Goal Selection for Embedded Systems with Oversubscribed Resources

Gregg Rabideau* Steve Chien† and David McLaren‡
Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA 91109

DOI: 10.2514/1.48919

We describe an efficient, online goal selection algorithm and its use for selecting goals at runtime. Our focus is on the replanning that must be performed in a timely manner on the embedded system where computational resources are limited (as in many aerospace systems). In particular, our algorithm generates near optimal solutions to problems with fully specified goal requests that can oversubscribe available resources but have no temporal flexibility. By using a fast, incremental algorithm, goal selection can be postponed in a "just-in-time" fashion allowing requests to be changed or added at the last minute. This enables shorter response cycles and greater autonomy for the system under control. We show that the average case complexity for updating the goals set is $O(N \lg N)$ and runtime execution is $O(N)$. We perform an empirical analysis on both synthetic data and space operations mission-like scenarios that confirm these performance characteristics. Finally, we show that scaling these performance figures to existing, very limited onboard spacecraft embedded environments (a Mars Reconnaissance Orbiter like environment) appears feasible.

Nomenclature

G	number of goal types
$\text{GPP}_{\max}$	maximum number of goals that can exist per priority level
M	number of selected goals
N	number of goal instances
NPG	number of goals per goal type
NPP	number of goals per priority level
NPT	number of goals per time unit
P_i	priority of goal i
$P_{\max}$	maximum priority of any goal
R	number of resource types
RPG	number of resources per goal

Received 13 January 2010; accepted for publication 1 November 2010.

* Software Engineer, Artificial Intelligence Group, M/S 301-260, 4800 Oak Grove Dr., Pasadena, CA 91109, gregg.rabideau@jpl.nasa.gov
† Senior Research Scientist, Artificial Intelligence Group, M/S 301-260, 4800 Oak Grove Dr., Pasadena, CA 91109, steve.chien@jpl.nasa.gov
‡ Software Engineer, Artificial Intelligence Group, M/S 301-260, 4800 Oak Grove Dr., Pasadena, CA 91109, david.mclaren@jpl.nasa.gov

S	total number of shared resources
S_i	number of resources shared by goal i
Score_i	score of goal i
X	total number of interacting goals
X_i	number of interacting goals for goal i

I. Introduction

SPACECRAFT flight systems typically have very limited computing resources. Autonomy software, like all flight code, must run within these limitations. In addition, many autonomous responses are time-critical. For these two reasons, autonomy algorithms must be designed to be efficient, preferably with provable guarantees on responsiveness. At first, this may sound like a difficult task, especially for complex problems such as planning and scheduling. But by utilizing general problem characteristics, we can narrow the scope of the problem, and increase our chances of finding a tractable solution. First, an autonomous flight system does not need to be indefinitely autonomous due to periodic contact with human operators. Spacecraft rarely operate for more than a few weeks without ground communication. Therefore, onboard plans typically do not need to cover a time frame greater than a week or two. Second, autonomy software only needs to address the dynamic, unpredictable behaviors of the overall system. Many spacecraft behaviors can be predicted in advance. For example, spacecraft orbit predictions can be pre-compiled and uploaded for use by the onboard planner. Limiting the scope of the problem gives us hope at finding efficient algorithms. Our work focuses on a restricted planning problem that can be solved with a tractable algorithm that has a guaranteed worst-case complexity.

Specifically, we address the problem of autonomous high-level goal management for computationally limited robotic systems. We enable onboard and remote goal triggering through the use of an embedded, dynamic goal set that can oversubscribe resources. From the set of conflicting goals, a subset must be selected that maximizes a given quality metric.

Our work solves this problem with the following properties:

- Goals have fixed start times and durations (making this a goal selection problem rather than an NP-hard scheduling problem). Although some space scheduling problems do have temporal flexibility (e.g., an activity may be executed within a continuous time range), we have found this simplification useful for a significant range of space autonomy problems.
- Goals can be satisfied in multiple ways, but the selection of alternative branches or timings cannot require search (e.g., we may wait for an event, retry a fixed number of times, or branch based on a condition, but no backtracking occurs). This can be viewed as backtrack-free hierarchical task network planning.
- Goals can conflict by exceeding the limits of shared resources (e.g., oversubscription) with selection based on a strict priority ordering (i.e., a goal can never be preempted by any number of lower-priority goals). Although we acknowledge that there are many models of priority and this model may not be appropriate for many autonomy problems, we have found this model to be useful for a significant spectrum of space autonomy problems. Goals are only allowed to interact through shared resources and not through more complex state models (e.g., spacecraft pointing, etc.,). Again, although we acknowledge that many space operations problems do not well match this assumption, our goal selection framework can still model these interactions with lower fidelity (exhibited as lower selection/execution efficiency).
- Goals can be added, removed, or updated at any time, and the "best" goals will be selected for execution.

This problem is motivated by our experience in space mission operations, such as autonomous operations of the Earth-Observing 1 (EO-1) satellite [1] and operations conducted by the Autonomous Sciencecraft Experiment (ASE) flight and ground software [2]. In ASE, events are detected onboard which trigger changes in goal requests. For example, images taken of the Earth can be processed onboard to detect interesting events such as volcanic eruptions. These detections can then trigger changes to upcoming goals such as increasing the priority of requests for images of the same volcano. On the ground, sensorweb processing may detect similar events and upload new goal requests in a short time frame.

Our new work aims to provide these capabilities, but in a minimal software system with runtime guarantees. We demonstrate our prototype implementation on ASE scenarios. In the goal manager (GM), goal selection is postponed until the latest possible time, allowing goals to be added, removed, or changed just before execution. This dynamic goal set enables additional autonomy capabilities such as onboard and ground-based event triggering, similar to ASE. The GM can provide these capabilities with theoretical guarantees on worst-case response time and without the need of a general purpose AI planner.

Although our algorithms are general, we have implemented them as prototype extensions to the virtual machine language (VML) execution system. VML is advanced, multimission flight, and ground software developed for NASA and flown on a number of past and current missions, including the Spitzer Space Telescope, Mars Odyssey, Stardust, Genesis, Mars Reconnaissance Orbiter (MRO), Phoenix, and Dawn [3]. Our prototype extends the largely *procedural* language to include *declarative* statements for goals and resources that can be used during both the planning and execution phases. Specifically, the user provides a set of resources and the goals that use them, and the GM maintains the set of requested goals, their priorities, and their interactions (i.e., shared resource constraints). When a goal is submitted, the GM quickly analyzes the goal to determine whether or not it should be selected for execution. When the current time approaches the scheduled start time of a selected goal, the goal is committed and satisfied by spawning the corresponding VML sequences. At this point, we are committed to the goal, and the sequences begin execution. During execution, any part of a sequence that requires resources can wait for the resources to become available or perform a failure response.

To provide tractable planning solutions, we must make some simplifying assumptions about the problem structure. First, we assume that start times of goals are fixed during goal selection. This is a reasonable assumption due to the nature of a spacecraft in orbit—opportunities for communications and science observations occur at specific (repeating) times. Once goals are selected, supporting activities can have flexible start times through the use of robust executives such as VML. Also, we have found that many spacecraft resource constraints can be abstracted to the goal level. For example, the EO-1 spacecraft can point science instruments to only one target at a time. Thus, for target locations in close proximity, we must choose one of possibly many observation goals. We take advantage of these assumptions to develop efficient algorithms that provide advanced onboard autonomy capabilities.

The remainder of this paper is organized as follows. First, we describe our representation for resources and goals. Second, we detail the goal management algorithm and analyze its computational complexity. Third, we describe an empirical analysis of the goal management algorithm, showing that it conforms to the predicted computational complexity. Specifically, we show that the goal management algorithm updates in time linear in the number of goals. Fourth, we show that a scaling to a flight processor for an existing embedded application, onboard observation selection for the MRO space mission, is feasible. Finally, we describe how this GM algorithm applies to spacecraft operations and describe related work.

II. Resource Constraints

The primary driver for goal selection comes from the constraints on resources shared by the goals. A resource constraint represents a value and a bound on that value over a period of time. Resource constraints can exist as part of goals, activities, or sequences. Goal resource constraints are used for goal selection, whereas the other types are used during goal execution. A combination of effects of constraints on the same resource conceptually comprises a timeline (although we do not maintain an explicit representation of a timeline). We define a resource constraint as a tuple (*id*, *type*, *start*, *end*, *value*, *initial*, *min*, *max*). The *id* uniquely identifies the affected resource for the purpose of analyzing the interaction with other resource constraints. The *type* specifies the type of effect that the constraint has on the resource. The time range (*start*, *end*) specifies the temporal scope of the constraint. The last four values specify, respectively: the constraint value, the initial value of the resource in the absence of all constraints, the minimum valid resource value, and the maximum valid resource value.

We have identified four fundamental types of resource constraints: *Producer*, *Consumer*, *Assigner*, and *Requirement*. A *Producer* adds the constraint value to the resource at the start of the time range and subtracts it at the end (where the end may be infinity). A *Consumer* subtracts at the start and adds at the end. An *Assigner* simply assigns the constraint value at a specific time point and is necessary when specifying a new value that is independent of previous values. A *Requirement* specifies only a constraint on the value of the resource over a period of time (i.e., it has no effect on the resource value).

A resource constraint can be defined for any type that can be evaluated with the set of operators in Fig. 1. These arithmetic operators allow us to compute resource values from a set of interacting resource constraints, and the

```
+= (used for producers)
-= (used for consumers)
 = (used for assigners)
 < (used for validity check)
== (used for validity check)
```

Fig. 1 Operators.

boolean operators are for testing the validity of computed resource values. For example, for a single producer, we would add the produced value to the initial value and compare the result to the maximum value. If the constraint check fails, the resource value is considered invalid (i.e., has conflicting constraints).

The definitions of the operators are intuitive for simple types such as integers, doubles, and strings. For sets, we define them as follows: addition ($+$) is set union, subtraction ($-$) is set subtraction, assignment ($=$) replaces all values in one set with values from another, less ($<$) is a lexicographical ordering on two sets, and equals ($==$) returns true if each element in one set is equal to exactly one element in the other set. For set resources, we introduce another operator for set containment. This allows us to specify a constraint that requires the computed resource value (which is a set of values) to contain the constraint value.

A. Resources at Runtime

Even with fast dispatching algorithms, sequences must be issued in advance of their requested execution time, and the state of the system may change in the interim. Therefore, at execution time we want to prevent or postpone an activity or sequence from executing if that activity/sequence requires resources that are not yet available. To achieve this, we implement runtime resource constraints using a generalization of counting semaphores.

A counting semaphore works as follows. A global count is initialized to the number of units available for a resource. A task can acquire (take) a resource if the count is greater than zero, and in the same operation, the count is decremented. If the resource is not available (i.e., the count is zero), then the task will block until it becomes available. When a task is finished with the resource, it can release (give) the resource by incrementing the count. For example, a binary semaphore (i.e., mutex) can be implemented using the commonly found atomic instruction test-and-set:

bool TestAndSet(*lock*)

TestAndSet will take the resource by assigning *lock* to true if and only if it is currently false. Otherwise, it will simply return true. We can later give back the resource by assigning *lock* to false.

More generally, consuming a resource is similar to acquiring a semaphore, but with a specified amount to be consumed, and with a specified bound on the resulting resource value after consumption. Execution on the calling task will block until bounding condition is met. Any change in either the resource value or the restricting bounds will trigger a check on the condition. Producing a resource is similar to releasing a semaphore, but including the conditional check found in acquiring a semaphore. Resource production also has specified values for the amount to produce and the bounds on the resulting resource value. In this way, consumption and production differ only in the direction in which the resource is changed (decreased or increased, respectively).

For resources, we define atomic runtime operations for: producing, consuming, assigning, and checking a resource value. For the first three, in which the resource value is changed, the change will not occur until the given bounds cover the changed value. The resource check operation simply blocks the calling task until the resulting resource value will fall within the given bounds. In all cases, an optional timeout can be given to continue execution even when the resource constraint is not met. These operations must be atomic (i.e., noninterruptible) in order to guarantee that the resource value will not change by another task between the time that the bounding constraint is checked and the resource value is changed.

The four atomic resource operations are defined in Fig. 2: *id* is a reference to the shared resource; *val* is the amount to be produced, consumed, or assigned to the resource; *min* and *max* specify the bounding constraint on the resulting value; and *t* is the amount of time that the operation will be delayed for the bounding constraint. The functions return true if the bounding constraint was met and the operation was performed. They return false if the bounding constraint

```
bool ResourceProduce(id, val, min, max, t)
bool ResourceConsume(id, val, min, max, t)
bool ResourceAssign(id, val, min, max, t)
bool ResourceCheck(id, min, max, t)
```

Fig. 2 Atomic resource operators.

```
Seq1()
  if ResourceProduce(3, 2, 0, 10, 0)
    Cmd1()

Seq2()
  if ResourceConsume(3, 4, 0, 10, 1)
    Cmd2()
    Cmd3()
  else
    Cmd4()
    Cmd5()
```

Fig. 3 Example sequences using resource constraints.

was not met, the timeout was reached, and the resource operation was not performed. Using optional timeout values allows us to implement waits with interrupts instead of spin loops.

Figure 3 shows an example of two sequences that make use of runtime resource constraints. The first sequence (Seq1) includes a command that is expected to produce two units of the shared resource #3. Seq1 will fail to produce the resource if it is already near its maximum value. The second sequence (Seq2) contains commands that consume four units of the same resource. Cmd2 and Cmd3 use four units of resource #3, which must result in a value between 0 and 10 units for the commands to be valid. If the resource cannot be consumed, Seq2 executes Cmd4 and Cmd5 instead. Resource producers (such as Seq1) must execute before Seq2 to provide the resource, with Seq2 delaying the commands at most 1 s to wait for the resource to become available. If the resource is not available within 1 s, an alternate sequence of commands is executed.

III. Goals

We define a goal as a tuple (*id*, *priority*, *start*, *end*, *constraints*) to represent the request for execution of an activity or set of activities. The *id* is used to uniquely identify the goal. The *priority* is used to rank goals. The *start* and *end* values specify the expected temporal scope of the goal. Because of system uncertainties at the time the goal is requested, the start and end times contain only the requested or expected values. Goals also maintain a set of resource *constraints* that must hold for the goal to execute. Similar to the start and end times, resource constraints contain the requested or expected values for the resources.

The goal attributes are used for selecting and dispatching goals for execution. In addition, a goal must specify what is to be done when it is dispatched. Typically, this involves spawning a sequence to start execution at a given time. Essentially, we define goals as a summary of the intent and effects of one or more sequences.

A. Example: Onboard File System

When defining goals and resources, we worked to find a balance between a representation that is general and powerful, but also has the details required for efficient resource analysis and goal selection. We show the power of the representation with an example: managing an onboard file system. This example is of particular interest to us because many spacecraft (including EO-1) must deal with data products stored on an onboard file system. Typically, science activities write data to a file whereas engineering procedures read and downlink files, deleting them when appropriate to free up space.

First, the user has several goals that can be requested of the file system: create, delete, read, write, and format. Also, the file system has the following resource constraints: there is a limited number of file handles, there is limited disk space, performing an operation on a file requires that it exist on the file system, and finally, some operations cannot be done at the same time.

```
ResourceConstraint(id, type, start, end, value, initial, min, max)

FileHandleResource(type, time, value)
  : ResourceConstraint(1, type, time, infinity, value, 100, 0, 100)

FileMemoryResource(type, time, value)
  : ResourceConstraint(2, type, time, infinity, value, 1024, 0, 1024)

FileSetResource(type, time, value)
  : ResourceConstraint(3, type, time, infinity, value, {}, {}, {infinity})

FileReadWriteResource(start, end)
  : ResourceConstraint(4, Producer, start, end, 1, 0, 0, 1)
```

Fig. 4 Resource constraint constructors.

```
Goal(id, start, end, priority)

FileCreateGoal(id, start, end, priority, fileId)
   : Goal(id, start, end, priority)
  add FileHandleResource(Consumer, start, 1) to resources
  add FileSetResource(Producer, end, infinity, {fileId}) to resources

FileDeleteGoal(id, start, end, priority, fileId, size)
   : Goal(id, start, end, priority)
  add FileHandleResource(Producer, end, 1) to resources
  add FileMemoryResource(Producer, end, size) to resources
  add FileSetResource(Consumer, start, infinity, {fileId}) to resources

FileReadGoal(id, start, end, priority, fileId)
   : Goal(id, start, end, priority)
  add FileSetResource(Requirement, start, end, {fileId}) to resources
  add FileReadWriteResource(start, end) to resources

FileWriteGoal(id, start, end, priority, fileId, size)
   : Goal(id, start, end, priority)
  add FileMemoryResource(Consumer, start, size) to resources
  add FileSetResource(Requirement, start, end, {fileId}) to resources
  add FileReadWriteResource(start, end) to resources

FileSystemFormatGoal(id, start, end, priority)
   : Goal(id, start, end, priority)
  add FileHandleResource(Assigner, end, 100) to resources
  add FileMemoryResource(Assigner, end, 1024) to resources
  add FileSetResource(Assigner, start, {}) to resources
```

Fig. 5 Goal constructors.

We can model this file system with four resource constraints and five goals, shown in pseudocode in Figs. 4 and 5. A general resource constraint constructor takes eight arguments to specify values for the eight attributes: *id*, *type*, *start*, *end*, *value*, *initial*, *min*, and *max*. A subclass is defined for each of the four resource constraints in this example. The resources represented are: file handles, memory, the set of files on the disk, and read/write locks on files. A general goal constructor takes four arguments to specify values for the first four goal attributes: *id*, *start*, *end*, and *priority*. A subclass is defined for each of the five goal types in this example. The goals represent file system requests: create, delete, read, and write files, and format the entire file system. The resources shared by these goals are: a set of file handles (100), limited disk space (1024K), a current directory listing, and finally an exclusive use of the file system by certain operations (e.g., cannot read and write at the same time).

Each goal instance creates and adds the resource constraints for that goal type. Creating a file consumes one of the available file handles, and produces a file with the specified unique ID. Deleting a file produces a file handle while consuming the file with the specified unique ID. It also produces disk space equal to the size of the file. Writing to a file consumes available memory equal to the size of the data written. Both writing and reading require that the unique file ID is a member of the set of available file IDs, and that no other reads or writes occur at the same time.

Each goal type also defines the required method for executing the goal. For example, creating a file would call *fopen* ( ) on a Unix operating system.

IV. Goal Selection Algorithm

We present an algorithm for selecting goals with oversubscribed resources. The pseudocode is shown in Fig. 6. The algorithm can be categorized as a repair-based approach with no search—the constraints and priorities define exactly which goals to choose. We focus on the reselection that is required when the requested goal set changes, either by adding a new goal or removing an existing goal. Changing parameters of a goal (e.g., start time, priority) can be implemented by removing the goal and adding it back with new values. Although making selections, goal parameter values (e.g., start times) are assumed to be fixed. The result is a set of nonconflicting "best" goals that have been selected for execution. After selections are made, conflicting goals are retained for additional consideration in the event of future changes to the goal set. In this implementation, conflicts are defined by shared resource interactions, and choices are made among conflicting goals using a strict priority rule. Only the highest-priority goals are selected, with ties broken by earliest request time (i.e., first-come-first-served).

The GM maintains data structures that enable efficient goal selection and dispatch. One set of goals is sorted by priority so that the algorithm can efficiently perform goal selection. The goals are also sorted by start time so that it

```
dispatch(minStart, maxStart)
  for each Goal g in allGoalsSortedByTime
    with startTime(g) >= minStart and startTime(g) <= maxStart
    if g is in selectedGoals
      start(g)

addGoal(g)
  for each Goal ag in allGoals
    if g interacts with ag
      add g to interacting goals of ag
      add ag to interacting goals of g
  add g to allGoals, allGoalsSortedByTime, and allGoalsSortedByPriority
  updateSelectedGoals(g)

removeGoal(g)
  remove g from allGoals, allGoalsSortedByTime, and allGoalsSortedByPriority
  updateSelectedGoals(g)
  for each Goal ag in allGoals
    remove g from interacting goals of ag

updateSelectedGoals(g)
  for each Goal sg in selectedGoals
    with priority <= priority of g
    remove sg from selectedGoals
  for each Goal ag in allGoalsSortedByPriority
    with priority <= priority of g
    if wasStarted(ag) or isBestGoal(ag, selectedGoals)
      add ag to selectedGoals

isBestGoal(g, selectedGoals)
  initialize each Resource used by g
  add effects of g to each Resource used by g
  for each Goal ig in interacting goals of g
    if ig is in selectedGoals
      for each Resource r shared by ig and g
        add effects of ig to r
        if r is invalid
          return false
  return true
```

Fig. 6 Incrementally updating the set of selected goals.

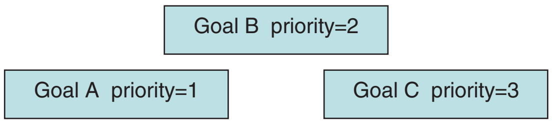


Fig. 7 Goals A and C are selected.

can quickly find the next selected goal to dispatch for execution. For each goal, the GM maintains a set of interacting goals (i.e., goals that share a resource) to assist resource reasoning. According to the priority rule, a goal will be selected and dispatched for execution if and only if it does not conflict with a higher-priority goal, which does not conflict with an even higher-priority goal, etc. For example, in Fig. 7 we assume that overlapping goals conflict. Goal C is highest priority and is selected. Goal B is not selected because it conflicts with C. Because B is not selected, goal A is selected even though it is lower-priority than B. Therefore, goals A and C are the "best" goals.

Adding and removing goals involves three steps: updating the sets of interacting goals, updating the sorted goal sets, and updating the selected goal set. This last step is the most interesting and is shown in pseudocode in Fig. 6. In this step, we first deconflict the schedule by unselecting all goals with lower or equal priority than the goal being added or removed (lines 22–24). Goals with higher-priority are unaffected and can remain selected or unselected. Then, all goals with lower or equal priority are reevaluated (lines 25–28). Evaluating a goal g involves adding the resource effects of g to the effects of all selected (i.e., higher-priority) goals that interact with g (lines 31–36). If the combined resource effects are invalid (i.e., a conflict exists), then g will not be selected (lines 37–38). We only need to consider conflicts with higher-priority goals because, according to the priority rule, a goal cannot be replaced by any number of lower-priority goals.

Finally, dispatching goals within a given time range simply involves finding selected goals that fall in that range. The dispatch function is intended to be called periodically with a small time range covering the near future. It is important to note that the low-priority conflicting goals are retained so that changes to the goal set can be made at any time (goals added, removed, or updated), and goals will be dispatched from the latest set of selected goals. Once a goal has started executing (determined by the "wasStarted" function on line 27) it will thereafter be selected regardless of priority. A goal expires if it is unselected and falls in the past, or if it is selected and all of its interacting goals completely fall in the past. Expired goals are periodically removed from the goal sets. As a final note, the definition of "interacting" and "conflicting" can be arbitrary boolean operators. In this work, resources define which goals interact and resource calculations are performed to determine conflicts.

A. Algorithm Analysis

We now describe the runtime computational complexity of our goal selection algorithms. Selecting the best goals and updating the cache (lines 21–39) is

$$O(\mathrm{Max}_{i=1,\ldots,N}(M \lg M + N(\lg M + X_i(\lg M + S_i))))$$

Goals are stored in tree data structures with a log-based lookup. The first term ($M \lg M$) comes from removing lower-priority goals from the selected goals (lines 22–24). The longer second term comes from reselecting the best goals (lines 25–39).

Assuming worst case, where each goal interacts with every other goal ($X_i == N$ for all i), each goal uses all resources ($S_i == R$ for all i), and all goals are selected ($M == N$):

$$O(N \lg N + N(\lg N + N(\lg N + R)))$$

Or

$$O(N \lg N + N \lg N + N^2 \lg N + N^2 R)$$

Since $N^2 \lg N$ dominates $N \lg N$:

$$O(N^2 \lg N + N^2 R)$$

And assuming that R is constant (defined by the domain), we have

$$O(N^2 \lg N)$$

This is a theoretical worst-case complexity. In practice, each goal will typically use a subset of the resources, and many of the goals will not be selected for execution. More important, goals interact with a small number of other goals (X_i is constant) due to the temporal scope of the resource constraints (i.e., effects on resources have limited extent). This gives us

$$\Theta(N \lg N)$$

Our empirical analysis (discussed in a later section) supports the average and worst-case bounds. Finally, this selection process is performed only when goals are added or removed from the dispatcher, which is assumed will be done at noncritical times. Once we have cached the best goals, checking a specific goal is a simple lookup in the set. Dispatching a selected goal for execution is

$$O(\lg N + \lg M)$$

The first term is from the lookup for goals due for execution which we assume to be small (typically one). The second term is from the lookup in the set of selected goals. Assuming worst case where all goals are selected ($M == N$), we get

$$O(\lg N)$$

Finally, we take a look at the expected quality of the output of the algorithm. Our primary claim is that the algorithm is optimal for the given priority rule. In other words, a goal i with priority P_i will always be selected in place of any number of goals with priority less than P_i. Intuitively, this follows from the decreasing priority order in which goals are selected. Now consider scoring the selected goal set with a weighted sum using weights sufficiently large at P_i to outweigh all goals with priority less than P_i. Our goal selection algorithm will maximize this score, but only when the requested goals are assigned unique priorities. If goal priorities are not unique, the overall weighted sum of priorities depends on the order in which we select goals with equal priority. In our implementation, the goal submitted first is selected first. This goal, however, may use more resources than the other goals with the same priority, accommodating fewer goals at lower priorities, and producing an overall lower score. Our initial empirical analysis, however, shows that our solutions do not fall far from optimal using this scoring method.

B. Algorithm Assumptions, Limitations, and Requirements

We make several assumptions to keep the goal selection algorithm simple and efficient.

- We do not solve the general planning problem. We only decide which high-level goals should be selected. We do not search for alternate methods of achieving the high-level goals. Although less powerful, this tends to be more accepted by spacecraft engineers who prefer consistency and predictability. The tasks of goal decomposition and command execution are left to an executive or sequencing engine (e.g., VML). These systems can be very expressive and allow goals to be expanded in a complex, context-dependent manner.
- We do not solve the general scheduling problem. We only decide on *which* subset of requested goals and activities to add to the plan, not on *when* they should be scheduled. Goals and activities must be submitted with predetermined start times. As an example, for an orbiting spacecraft with repeating science opportunities, this restricted form of planning can select which observation to perform on a specific orbit, but cannot select alternate overflights for a particular observation.
- We are only reasoning at the goal level. Resource reasoning is performed on goal resources which are assumed to be abstractions of the expected use of resources by the lower-level commands. We found this abstraction useful for many of the EO-1 resource constraints.
- We also assume that goals are ranked using the strict priority rule. In other words, any number of low-priority goals can be preempted by a high-priority goal. When goals have equal priority, the goal that was requested first will take precedence (i.e., first-come-first-served). EO-1 scientists were most comfortable with this simple priority scheme.

It is worth pointing out that even with these restrictive assumptions, the goal replacement capabilities we are offering far exceed what is available on typical spacecraft today, whether implemented in general commanding capabilities or custom flight software. Specifically, they are capable of representing the goal replacement capabilities currently operational in ASE.

To benefit from these capabilities, however, users must encode some additional knowledge when defining goals compared with defining activities or sequences strictly for the purpose of execution. First, users must provide some form of selection criteria. In our case, this is a priority for the goal. The user must also specify a summary of the expected resource usage for each goal. Where resource use at runtime may be intricate or even implicit, goal resources force the user to define resource use in an explicit and predictable way. Finally, users must provide an expected start and end time for the activity or sequence requested by the goal. This is necessary for predicting the temporal scope of the resource use.

V. Autonomous Spacecraft Operations

Our work was motivated by scenarios taken from the ASE used in operating the EO-1 satellite. In these scenarios, the science team starts by providing a set of data collect requests that oversubscribe spacecraft resources. A baseline set of collects and alternates are selected and uplinked. During execution, onboard science processing may generate new goal requests [2]. Ground-based sensorweb processing may do the same using uplinked commands [4]. A prototype GM was implemented and tested on these scenarios. The EO-1 model consists of VML sequences that implement activities for operating EO-1, including collecting and downlinking science data. The system was run on a typical EO-1 collect-downlink cycle where onboard resources (e.g., science data storage) are oversubscribed. At runtime, a simple spacecraft simulator was used to mimic command behavior, including effects on resources. Goal request changes were simulated using a time-tagged file containing the change specifications.

We also studied planning and sequencing problems from the MRO and the Mars Exploration Rover (MER) missions. From these, we identified several scenarios that might benefit from this technology. For example, in data relay scenarios an anomaly can either trigger a request for an emergency relay communication goal, or create a relay opportunity from a failed goal that releases resources. In addition, goal selection could be used to maximize the use of onboard data storage. Rejected science goals, which at first seem to oversubscribe this highly contended resource, could be selected at runtime if more data storage is available than originally expected.

First, we examine a data relay scenario, where low-priority MER data can be sent to Earth via MRO (Fig. 8). Initially, MRO cannot service the downlink relay because it is collecting high-priority science data. MER would check for signal, fail, and continue doing something else (e.g., take observations). However, if before the MER fly-over, an MRO anomaly makes the MRO science collect goal obsolete, the downlink relay would become possible. MER would check for signal, succeed, and both would initiate sequences necessary to relay the low-priority MER data. This scenario requires a dynamic goal set to allow the high-priority (but failed) science collect goal to be replaced by the low-priority downlink goal.

Next, we consider another data relay scenario, where critical MER engineering data must be sent to Earth as soon as possible (Fig. 9). Initially, MRO has science goals planned, and goals for MER relay opportunities are stored onboard MRO at low-priority. At some point, an anomaly on MER creates the need to downlink high-priority data. On the next MER fly-over, MER would indicate the change in priority for the next relay opportunity. Assuming this is higher priority than the MRO science, MRO would replace the science goal with the MER relay goal. Again, a dynamic goal set is necessary for this type of scenario.

Finally, we consider a change in an MRO science request initiated by the science team (Fig. 10). MRO maintains requests in an integrated target list (ITL) where each line/item in an ITL has a reference to a file describing the science request. Data volume is a major driver for science requests. With more available storage, a scientist might choose longer durations or higher resolutions (i.e., larger image size). Onboard software could automatically analyze the data volume, choose better parameters for the target, and replace the science request file referenced by the ITL. Changes to parameters may require constraints to be rechecked, and/or other sequences to be adjusted to make the change fit. This demonstrates a need for both a dynamic goal set and flexible execution. The ephemeris is a driver for the CRISM science instrument on MRO because of its scanning mechanism. Changes in orbit/ephemeris require changes to parameters used for pointing the instrument. Parameter calculations on the ground can be triggered from ephemeris changes, automatically generating a new file uplink request, and changing the onboard goal set.

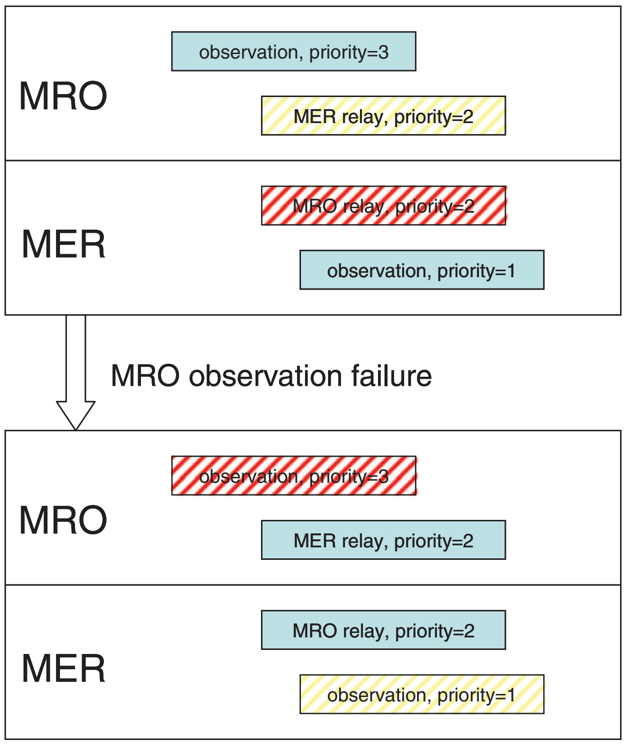


Fig. 8 Blue indicates selected goals, yellow is rejected, and red is failed.

VI. Empirical Analysis

The motivation for our experiments is to show that:

- the runtime of the algorithm matches the theoretical analysis (most importantly that updates to the goal set are performed in time linear in the size of the goal set),
- the quality of our solutions is near optimal and much better than a greedy algorithm.

Experiments were run on two problem sets:

- randomly generated problems for randomly generated domains,
- randomly generated problems for the EO-1 domain.

For the randomly generated domains, we identified several domain parameters that affect performance, including: number of goal types, number of resource types, and number of resources per goal. These parameters impact the level of interaction between goals. We expect higher levels of interaction to result in slower runtimes, but a higher potential for quality improvements when compared with greedy solutions.

When generating random problem sets for either a random domain or the EO-1 domain, we looked at the following parameters: number of goal instances, number of goals per goal type, number of goals per time unit, and number of goals per priority level. Again, the first three parameters impact the level of goal interaction. For example, more goals in a smaller time range will have more shared resources that interfere. The last parameter, number of goals per priority level, was used to show that our solutions are optimal when guaranteed ($\mathrm{NPP} == 1$) and do not stray far from optimal otherwise ($\mathrm{NPP} > 1$).

Experiments were generated using a random problem generator. Specifically, random domains include 20 resource types and 20 goal types with each goal type using 10 resources. The EO-1 domain contains seven resource types and five goal types with each goal using about three resources on average. Goals are defined for collecting, processing, and downlinking data. Shared resources are defined for the onboard data recorder and file system. For both random and EO-1 domains, we generated 300 problems sets containing 300 goals each.

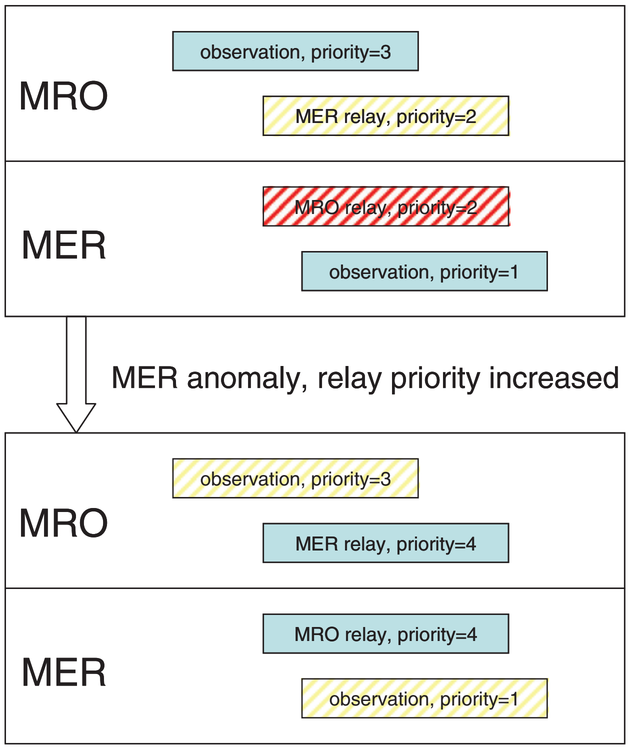


Fig. 9 Blue indicates selected goals, yellow is rejected, and red is failed.

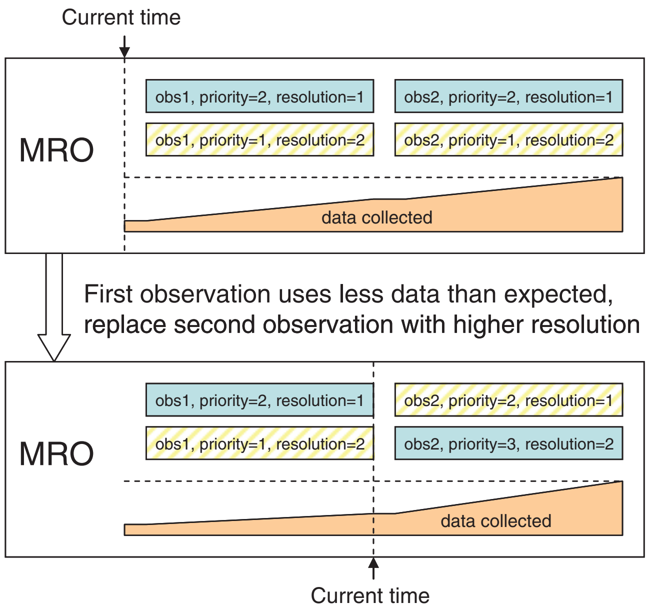

Current time
MRO
obs1, priority=2, resolution=1
obs2, priority=2, resolution=1
obs1, priority=1, resolution=2
obs2, priority=1, resolution=2
data collected
First observation uses less data than expected,
replace second observation with higher resolution
MRO
obs1, priority=2, resolution=1
obs2, priority=2, resolution=1
obs1, priority=1, resolution=2
obs2, priority=3, resolution=2
data collected
Current time

Fig. 10 Blue indicates selected goals, yellow is rejected.

Three algorithms were used to generate solutions from the problems sets:

- Goal manager (GM),
- Optimal (OPT),
- Greedy forward dispatch (GFD).

GM runs the algorithm described in this paper. OPT is a variant of this algorithm that tries all possible orderings for adding goals with equal priority. This is meant to find better solutions that may have been missed by GM which breaks ties using a first-come-first-served rule. Because OPT considers all goal priority ties at each search option OPT is in effect performing branch and bound search of the goal priority search space with a simple upper bound on the remaining goals priority. Therefore as branch and bound with an admissible heuristic estimator for the remaining goal priority OPT is indeed an optimal solver for the goal selection problem as formulated. In GFD, the same GM algorithm is used but with a short time window of consideration. Only goals within that time window are considered for selection, and are selected based on priority. This limited time horizon reduces the amount of resource propagation needed (by propagating only over the shorter interval) and by reducing the number of goals considered (by only considering those within the time horizon). Once goals are selected they are not allowed to be preempted by later higher-priority goals and rejected goals are not maintained for future consideration, reducing the number of interactions that must be analyzed. GFD is therefore suboptimal in that earlier goals may consume resources that could have been used for later priority goals (not in the window of consideration when the earlier goal was selected). GFD is used as a representative algorithm that is simple and fast, but more naïve at selecting goals, giving a lower bound on both runtime and solution quality. All algorithms were implemented in C++ and run on a Sun workstation configured with two 2.6 GHz AMD Opteron™ 252 processors, 16 GB of RAM, and the 64-bit Red Hat Enterprise Linux operating system.

The graphs in Fig. 11 show the average runtimes (top) and scores (bottom) for adding a single goal to an increasing baseline goal set for random domains (left) and the EO-1 domain (right). The CPU times reported are for adding one new goal to a set of N goals plotted on the x-axis. The scores reported are for the resulting set of selected goals. The score of a single goal i is calculated using the function:

$$\text{Score}_i = \text{GPP}_{\max}(P_i - P_{\max})$$

The score for a set of goals is the sum of scores for all goals in the set. This function ensures that the score for any number of goals at a lower priority will not sum up to more than the score of a single goal at a higher priority.

In these runs, start times and priorities were chosen at random from an increasing range of values, representing typical scenarios where NPT and NPP are relatively constant. This corresponds to the average case complexity analysis, and the data for random domains support the $\Theta(N \lg N)$ result. The $N \lg N$ fit to the GM data is shown in the graph. The EO-1 domain demonstrates the worst case where nearly all goals share the same resource (the onboard file system), and we see a best fit to an N^2 curve. The exponential runtime of OPT is due to our naïve implementation that considers all possible ordering of goals with equal priority. The graphs also show GM producing solutions with scores at or slightly lower than OPT (plotted on top of each other), but with GM and OPT both scoring much higher than GFD. The 95% confidence intervals (using standard error of the mean) are shown, but are very small for OPT and GM. The scores produced by GFD are noisy due to its preference for selecting goals with earlier start times instead of higher priority.

In other experiments we looked at how the algorithms scaled when adding a large number of goal requests. Here we examined algorithm behavior along individual dimensions that were expected to have an impact on performance. In one experiment, we focused on the number of resources per goal. The results can be seen in Fig. 12. Looking back at the algorithm analysis, the two terms affected by RPG for a given goal are X and S. For these terms, the algorithm is worst-case $O(SX)$. Because both X and S will increase as *RPG* increases, this roughly makes it $O(\text{RPG}^2)$, matching the CPU curve in Fig. 12. The second graph in Fig. 12 shows the score drop quickly as RPG increases, until very few goals are selected and all algorithms perform equally poor.

In another experiment, we focused on the number of goals per priority level, and the results are shown in Fig. 13. Here, as expected, the runtime of GM and GFD were not affected. OPT, however, demonstrates exponential behavior and problems quickly become too difficult for it to solve in a reasonable amount of time. All algorithms find better solutions as NPP increases, but with varying rates. GM scores increase slightly faster than GFD, and OPT slightly faster than GM.

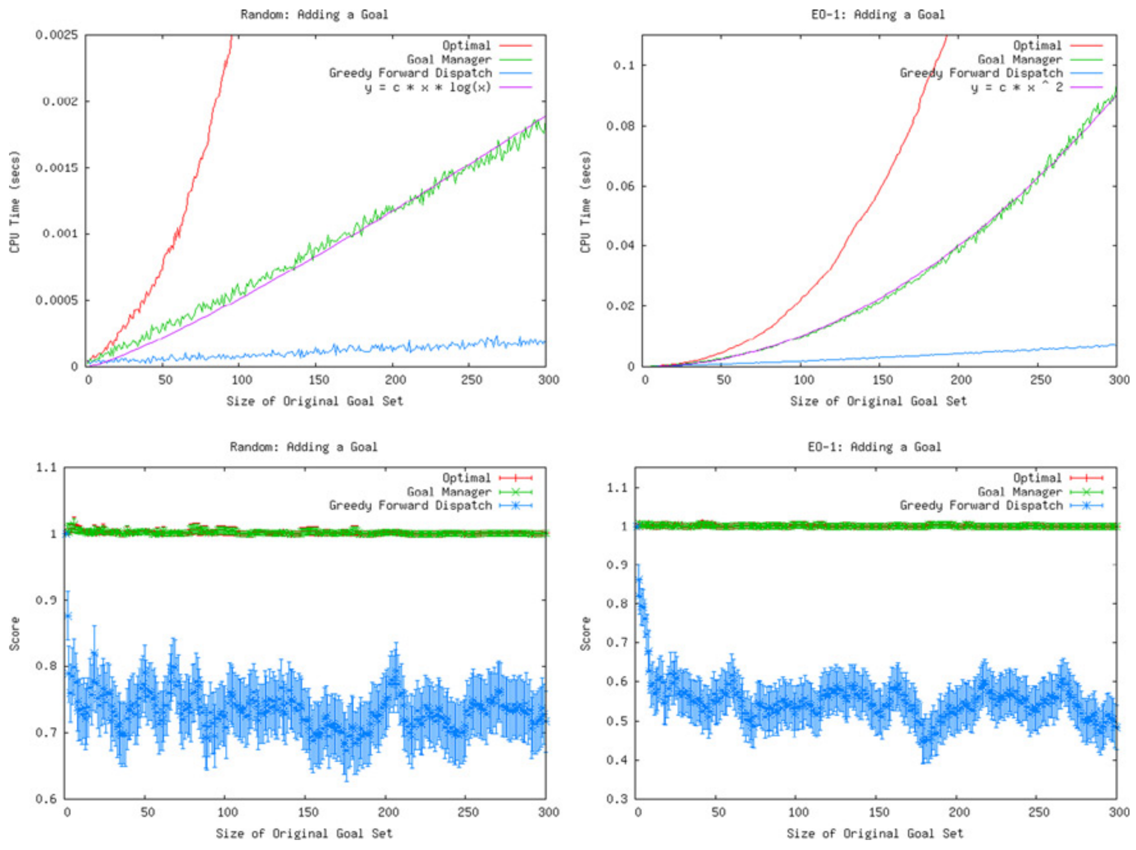


Fig. 11 Response time and solution quality from adding a goal in the random and EO-1 domains.

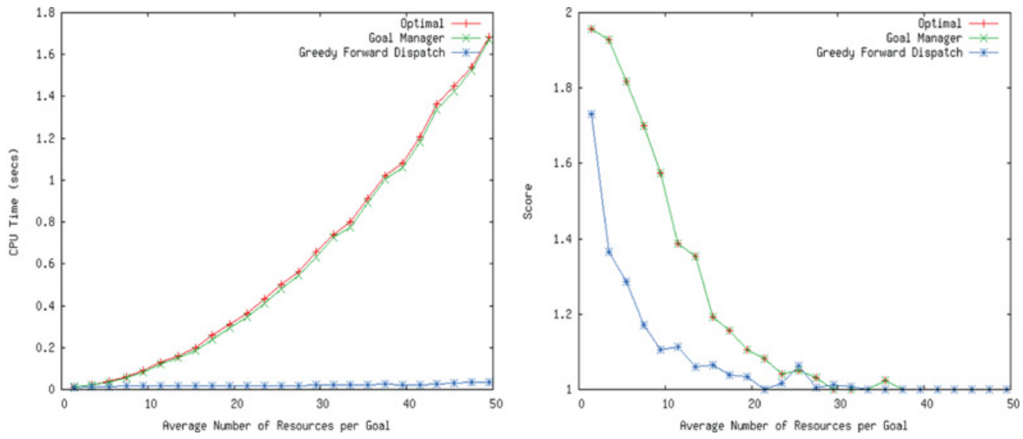
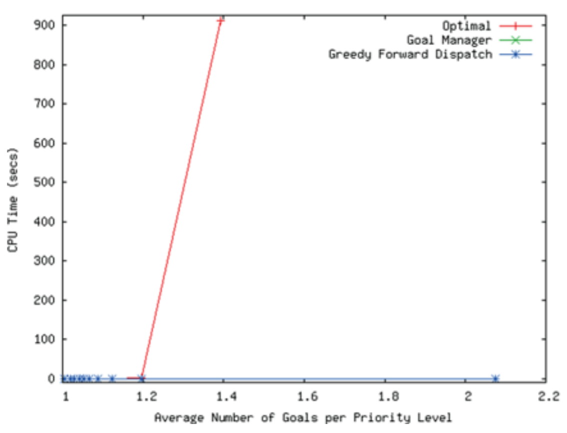


Fig. 12 Increasing the number of resources per goal.

Finally, we looked at how the percent of selected goals might affect performance. The results are shown in Fig. 14. Here we were expecting the runtime to strictly increase as the percent of selected goal increases. But as you can see in the first graph of Fig. 14, OPT and GM runtimes actually decrease at low percentages but eventually begin increasing. This can be explained by noting that solutions with a low number of selected goals are found only when the problem has a high number of interacting goals. As shown earlier, increasing interactions will cause a polynomial



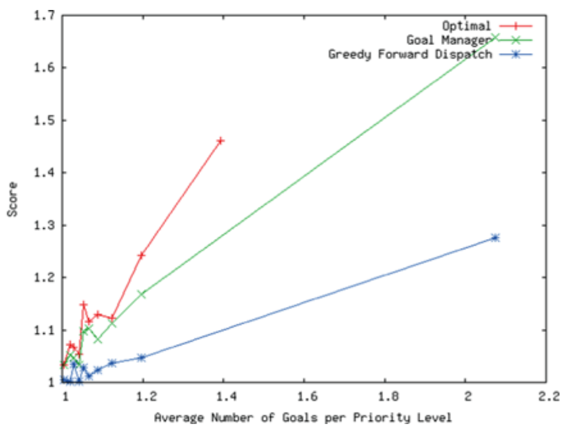
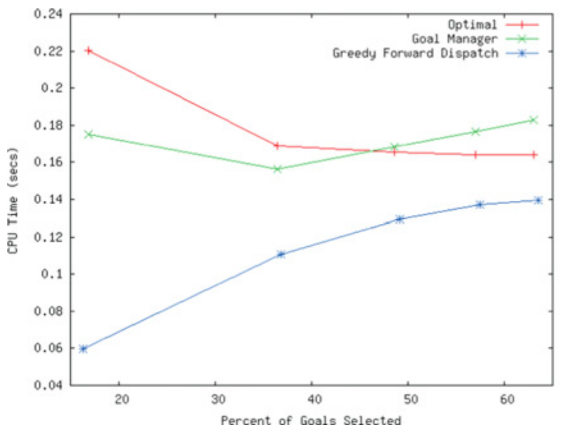


Fig. 13 Increasing the number of goals per priority level.



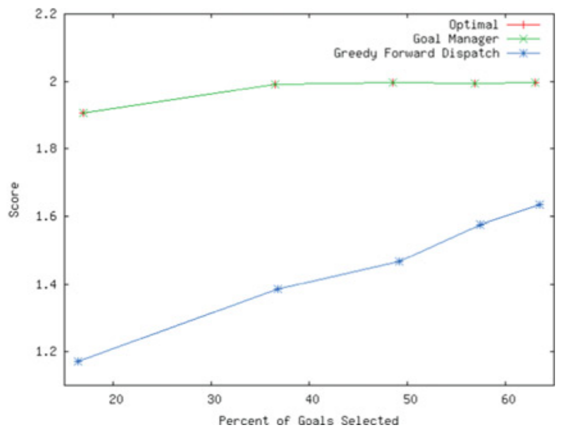


Fig. 14 Increasing the percentage of goals selected.

increase in runtime. At least for small numbers of selected goals, this seems to be dominating the linear increase in runtime caused by the increase in selected goals. Looking at the second graph of Fig. 14, both OPT and GM produce better solutions than GFD. But the gap narrows as the problem gets easier and more goals can be selected.

In summary, we have empirically shown our goal selection algorithm to:

- Exhibit low-order polynomial runtime behavior with respect to the size of the problem,
- Generate solutions near optimal and much better than a greedy approach.

In other words, with a few restrictions, we show that it is possible to preserve alternative goal sets and (when the need arises) reconsider previously rejected goals in a timely fashion in order to maintain high quality solutions.

VII. Estimating Onboard Running Time

In addition to empirically demonstrating the runtime of the GM algorithm and the quality of its solutions, we have conducted tests to estimate its runtime onboard a spacecraft. This scenario projects expected onboard runtimes if the GM algorithm were running onboard the MRO mission. Although actually running the GM algorithm on an embedded processor would have been ideal, the version of the VxWorks compiler (c++ppc) available to the developers was incapable of building the GM. To estimate runtimes for the algorithm on a flight processor and compare to an operational scenario we performed the following steps:

1. A Linux workstation was scaled against a commercial PowerPC 750 processor by running automated scheduling and planning environment (ASPEN) [5] on both platforms.
2. The PowerPC 750 was compared against the RAD 750 operating on MRO.
3. An MRO operations scenario mission requirement was derived based on the historical MRO observation request rate (e.g., requested observations per week).
4. Runtimes on the Linux workstation were scaled to runtimes on the PowerPC 750, and then to the MRO flight computer.

Both ASPEN and the GM were executed on a workstation running 64-bit Red Hat Linux, with a pair of 2.6 GHz AMD Opteron 252 processors and 16 GB of RAM. ASPEN was also run on a Gespac 150 MHz PowerPC 750 embedded board, with 128 MB RAM, running VxWorks 5.3. The MRO flight computer is a RAD 750 running at 133 MHz, with 128 MB of DRAM, also running VxWorks [6]. The Gespac system and the MRO flight computer are both based on the same architecture, have similar clock speeds, and run the same OS, and hence we expect similar performance.

ASPEN is a modular, reconfigurable application framework based on Artificial Intelligence techniques, which is capable of supporting a variety of planning and scheduling applications including spacecraft operations planning, planning for mission design, surface rover planning, ground antenna utilization planning, and coordinated multiple rover planning. It is operating onboard the EO1 spacecraft. It was run as a point of comparison between the workstation and the embedded system, using the same inputs on both systems. Over twenty test runs, ASPEN took an average of 44.3 s to complete on the embedded processor, while taking 1.58 s on the workstation—a 28-fold speedup.

The time scaling factor comes from three elements. Firstly, the 150 MHz Gespac is about 1.13 times as fast as MRO's 133 MHz processor. Secondly, it is assumed that up to 0.5% of total processor time could be allocated to the GM algorithm onboard MRO. The embedded processor can devote nearly 100% of its cycles to running ASPEN, and hence this results in a 200-fold scaling factor between the embedded system and MRO. Thirdly, the workstation used to perform the tests completes ASPEN test runs about 28 times faster than the embedded system. Multiplying the above three factors together results in expected runtimes on MRO that are 6322 times as long as those on the workstation.

MRO can perform up to 1200 nadir and off-nadir observations every 14 days, and MRO science teams consider more observations than are actually scheduled. A single observation can take up to 20 min. Two 8-hour downlink sessions for primary science acquisition were conducted each day for the science phase of the mission [7]. Thus, goals would only have to be stored for 24 hrs. These facts provide several inputs to the test: a 1-day window for scheduling observations, the number of observations that should be selected, and the duration range for a single observation (1–20 min). Rather than creating a brand-new model to simulate MRO, the model created for EO-1, used when analyzing the runtime of the GM, was reused. EO1 proved to be a comparably difficult problem to scheduling observations for MRO. It was executed with 2030 goals for these tests, divided over 14 days, yielding about 145 goals per day, of which about 90 were selected in each trial.

Two types of data were collected:

1) Generating random sets of 145 goals each, and measuring total time to schedule these.
2) After running the algorithm with an initial set of 145 goals, adding one more goal with a random priority, rerunning the algorithm, and measuring the time taken to perform this addition.

Running the GM with the input goals took an average of 0.193 s on the workstation. Multiplying by the scale factor, this turns out to be 20.3 min on MRO. The maximum time for adding all goals was 0.236 s on the workstation, translating to 24.9 min on MRO. The GM would thus use a small fraction of time to schedule all goals for a day of operations, less than 30 min out of 24 hrs. The time to add one goal varies with its priority, but is even shorter. Adding maximum-priority goals requires that all goals be rescheduled, but even this does not take an inordinate amount of time. Adding a maximum-priority goal took up to 22.3 ms on the workstation, scaling to 145.4 s on MRO. On average, adding a single goal to the set took 11.5 ms, or 72.5 s on MRO.

These runtimes indicate that it would be possible to run the GM onboard MRO and future space missions. It would be most useful when scheduling a new observation on short notice. If science planners noticed an interesting phenomenon taking place in current observations, they could quickly upload a new opportunity for observing it, and the spacecraft could reschedule observations to accommodate the request. By keeping all requested observations in memory, including those that were not scheduled, the spacecraft would be able to schedule new observations to take the place of others discarded in favor of new requests.

VIII. Planning and Execution with VML

Designed as a multimission application, VML is one of the most advanced onboard execution systems in widespread use for NASA missions [3]. VML has been used for a wide range of sequencing functions including: launch routines, orbit insertion, entry-descent-and-landing, science acquisition, and fault response.

We have implemented goals, resources, and the goal selection algorithm as prototype extensions to VML. Runtime resources (generalized semaphores) are integrated into existing VML sequence execution capabilities. A new thread/task, the GM, implements the goal selection algorithm and invokes the dispatch function periodically. Finally, new user interface functions are added to allow goals to be added, removed, or changed.

At runtime, we ultimately need a set of executable commands that achieve the selected goals. Using existing VML 2.0 sequencing capabilities [8], we define a general pattern to the language to enable goal achievement with flexible and robust execution. Specifically, the language pattern consists of defining hierarchies, preconditions and effects familiar to the AI planning community. To implement a hierarchy, a VML sequence for a goal or activity can spawn other VML sequences for subactivities, eventually breaking down to executable commands. When appropriate, sequence execution can be delayed to wait for the preconditions to be met, allowing more flexible execution. Finally, effects of the commands are monitored and appropriate responses can be defined to recover from failures and provide more robust execution.

IX. Related Work

Much of the research in planning and goal selection has focused on more general, *intractable* problems. For example, Smith looks at the more general problem that includes selecting goals and choosing their order when resource usage depends on the order of the goals (e.g., for a traveling rover) [9]. Both the Squeaky Wheel Optimization [10] and the Task Swapping [11] algorithms have been shown to improve oversubscribed schedules by rescheduling tasks to allow more goals to fit. Instead, we look at a more constrained problem where we can *guarantee* a solution in polynomial time, while still providing advanced autonomy capabilities useful for many embedded applications.

Tractable planning solutions typically take one of the following three approaches:

1) focus on average-case performance,
2) use domain-specific knowledge to simplify the general problem,
3) apply general restrictions to the problem to make planning tractable.

In the first approach, average-case performance is considered for difficult applications when occasional failures are acceptable (e.g., with the use of heuristics [12]). When relying on average-case performance, one must accept the fact that the algorithm may not efficiently solve some problems. In the second approach, domain-specific knowledge is used to encode problem-specific solutions when possible including the use of hierarchies [13,14] or context-dependent effects [15]. A knowledge-based solution, however, is one that is tailored for a particular problem and can be difficult to formally verify. Our work is most closely related to the third approach, which is supported by theoretical work showing how the efficiency of planning is related to the expressivity of the planning domain language. For example, Bylander [16] and Erol et al. [17] examine limited forms of STRIPS-style operators and the effect on planning complexity. Erol et al. [18] investigate restricted HTN planning. Jonsson and Bäckström examine how structural restrictions on state transition graphs impact planning complexity for the SAS+ formalism (a state variable representation) [19]. In the goal selection problem we describe, goals have fixed start times and durations. In all cases, different levels of restrictions result in different guarantees on the worst-case performance of planning. For applications that can meet sufficient restrictions, planning becomes verifiably tractable.

A considerable amount of work has been done in the area of online planning and execution. For example, Spacecraft Command Language (SCL) [20] provides a procedural language for spacecraft commanding similar to VML. Our prototype extension for goals and resources adds a declarative component not found in either language. Execution Support Language (ESL) [21] is an execution language for autonomous agents, implemented as an extension to the Common Lisp programming language. Task Description Language (TDL) [22] extends the C++ programming language to include the concept of a task. Both ESL and TDL take advantage of the generic language on which they are based. Providing such a rich language, however, has a cost—it can be much more difficult to verify programs written in such an expressive language. The capabilities we propose, whereas not as powerful, are similar to some provided by the TDL task types and constraints.

Finally, our goals and resources are similar to the concepts of goals and state variables that are central to JPL's Mission Data System (MDS). MDS [23,24] is a comprehensive approach to systems engineering and a methodology for the design and development of control system applications. Our goals and resources are similar to the concepts of goals and state variables that are central to MDS. MDS goals express intent of the operator over a time interval, whereas MDS state variables are used to model both the system under control and the intent of the control system.

X. Conclusions

This work targets development and maturation of more advanced embedded resource reasoning capabilities for execution systems. One such target is the onboard spacecraft execution language VML. Although the more typical orbital mission operations could benefit from onboard resource reasoning, future missions to dynamic environments would benefit even more. For example, future missions to a comet would need to reason about state and resources to exploit observation of transient events such as outbursts and jets. Round trip light times for such bodies means that ground control would not be timely enough to protect the spacecraft from these potentially hazardous events nor would it enable the spacecraft to image these exciting scientific events.

We have described a carefully constrained set of resource and priority reasoning capabilities designed to enable runtime goal selection within a limited computational environment. These capabilities enable fast reoptimization of goal sets which oversubscribe available resources and have a strict priority ranking. We have presented both a theoretical and an empirical analysis of our algorithm showing that for a limited goal selection problem linear reasoning time can be achieved. We have described its application to a number of typical spacecraft operations scenarios and also shown that the algorithm is scalable to onboard spacecraft computing constraints. Future work involves deploying this capability to future space missions.

Acknowledgment

This work was performed by the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration.

References

[1] Goddard Space Flight Center, "The Earth Observing One Mission Page," http://eo1.gsfc.nasa.gov [retrieved 21 February 2011].

[2] Chien, S., Sherwood, R., Tran, D., Cichy, B., Rabideau, G., Castano, R., Davies, A., Mandl, D., Frye, S., Trout, B., Shulman, S., and Boyer, D., "Using Autonomy Flight Software to Improve Science Return on Earth Observing One," *Journal of Aerospace Computing, Information, and Communication*, Vol. 2, April 2005, pp. 196–216.

[3] Grasso, C., and Lock, P., "VML Sequencing: Growing Capabilities over Multiple Missions," *Proceedings of SpaceOps*, Heidelberg, Germany, May 2008, AIAA-2008-3295.

[4] Chien, S., Cichy, B., Davies, A., Tran, D., Rabideau, G., Castano, R., Sherwood, R., Greeley, R., Doggett, T., Baker, V., Dohm, J., Ip, F., Mandl, D., Frye, S., Shulman, S., Ungar, S., Brakke, T., Descloitres, J., Jones, J., Grosvenor, S., Wright, R., Flynn, L., Harris, A., Brakenridge, R., Cacquard, S., and Nghiem, S., "An Autonomous Earth-Observing Sensorweb," *IEEE Intelligent Systems*, Vol. 20, May/June 2005, pp. 16–24.

[5] Chien, S., Rabideau, G., Knight, R., Sherwood, R., Engelhardt, B., Mutz, D., Estlin, T., Smith, B., Fisher, F., Barrett, T., Stebbins, G., and Tran, D., "ASPEN—Automating Space Mission Operations using Automated Planning and Scheduling," *Proceedings of the International Conference on Space Operations* (*SpaceOps*), Toulouse, France, June 2000.

[6] NASA, "Mars Reconnaissance Orbiter Arrival," March 2006, p. 39, http://www.nasa.gov/pdf/143619main_mro-arrival.pdf [retrieved September 2009].

[7] Zurek, R. W., and Smrekar, S. E., "An Overview of the Mars Reconnaissance Orbiter (MRO) Science Mission," *Journal of Geophysical Research*, Vol. 112, E05S01, May 2007.
doi: 10.1029/2006JE002701

[8] Grasso, C., "Virtual Machine Language (VML) v2.0 Users Guide," *JPL Document*, D-28342, June 2004.

[9] Smith, D. E., "Choosing Objectives in Over-Subscription Planning," *Proceedings of the International Conference on Automated Planning and Scheduling*, Whistler, Canada, June 2004.

[10] Joslin, D. E., and Clements, D. P., "Squeaky Wheel Optimization," *Journal of Artificial Intelligence Research*, Vol. 10, 1999, pp. 353–373.
doi: 10.1613/jair.561

[11] Kramer, L. A., and Smith, S. F., "Task Swapping for Schedule Improvement, A Broader Analysis," *Proceedings of the International Conference on Automated Planning and Scheduling*, Whistler, Canada, June 2004.

[12] Bonet, B., and Geffner, H., "Planning as Heuristic Search," *Artificial Intelligence*, Vol. 129, 2001, pp. 5–33.

[13] Nau, D., Au, T., Ilghami, O., Kuter, U., Murdock, J. W., Wu, D., and Yaman, F., "SHOP2: An HTN Planning System," *Journal of Artificial Intelligence Research*, Vol. 20, 2003, pp. 379–404.

[14] Tate, A., Drabble, B., and Kirby, R. B., "O-Plan2: An Open Architecture for Command, Planning, and Control," *Intelligent Scheduling*, edited by M. Fox and M. Zweben, Morgan Kaufmann, San Francisco, CA, 1994, pp. 213–239.

[15] Wilkins, D., and DesJardins, M., "A Call for Knowledge-Based Planning," *AI Magazine*, Vol. 22, No. 1, 2001, pp. 99–115.

[16] Bylander, T., "The Computational Complexity of Propositional STRIPS Planning," *Artificial Intelligence*, Vol. 69, 1994, pp. 165–204.

[17] Erol, K., Nau, D., and Subrahmanian, V., "Complexity, Decidability and Undecidability Results for Domain-Independent Planning," *Artificial Intelligence*, Vol. 76, No. 1–2, 1995, pp. 75–88.
doi: 10.1016/0004-3702(94)00080-K

[18] Erol, K., Hendler, J., and Nau, D. S., "UMCP: A Sound and Complete Procedure for Hierarchical Task Network Planning," *Proceedings of the International Conference on Artificial Intelligence Planning Systems (AIPS)*, June 1994, pp. 249–254.

[19] Jonsson, P. and Bäckström, C., "State-Variable Planning Under Structural Restrictions: Algorithms and Complexity," *Artificial Intelligence*, Vol. 100, No. 1–2, 1998, pp. 125–176.
doi: 10.1016/S0004-3702(98)00003-4

[20] SRA International Inc., "SCL for Mission Critical Command and Control," http://www.sra.com/space/ [retrieved 27 October 2010].

[21] Gat., E., "ESL: A Language for Supporting Robust Plan Execution in Embedded Autonomous Agents," *AAAI Fall Symposium: Issues in Plan Execution*, AAAI, Cambridge, MA, October 1996, pp. 59–64.

[22] Simmons, R., and Apfelbaum, D., "A Quick Reference for the Task Description Language (TDL) Version 1.3.2," http://www-2.cs.cmu.edu/∼tdl/tdl.html [retrieved 27 October 2010].

[23] Dvorak, D., Rasmussen, R., Reeves, G., and Sacks, A., "Software Architecture Themes in JPL's Mission Data System," *Proceedings of the AIAA Guidance, Navigation, and Control Conference*, AIAA, Portland, OR, 1999; also AIAA-99-4553.

[24] Barrett, A., Knight, R., Morris, R., and Rasmussen, R., "Mission Planning and Execution within the Mission Data System," *Proceedings of the International Workshop on Planning and Scheduling for Space*, Darmstadt, Germany, June 2004, pp. 13–22.

Christopher Rouff
Associate Editor