Runtime Goal Selection with Oversubscribed Resources

Gregg Rabideau and Steve Chien

Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Dr, Pasadena, CA 91109
firstname.lastname@jpl.nasa.gov

Abstract

We describe an online planning algorithm and its use for selecting goals at runtime. Our focus is on the re-planning that must be performed in a short time-frame by the embedded system where computational resources are quite constrained. In particular, our algorithm addresses problems with well-defined goal requests without temporal flexibility that oversubscribe available resources. 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.

Introduction

We address the problem of providing high-level, goal-based autonomy for computationally limited robotic systems. Specifically, we enable on-board 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 chosen that maximizes a given quality metric.

In our first prototype instantiation, we assume that the requested goals have fixed start times and durations, making this a goal selection problem rather than a scheduling problem. Goals can conflict by exceeding limitations of shared resources (e.g. oversubscription). The quality metric we use in our initial prototype is strict priority selection. In other words, a goal can never be pre-empted by a lower priority goal. High-level goals can be added, removed, or updated at any time, and the "best" goals will be selected for execution.

In addition, we provide robust and flexible execution of command sequences to satisfy goals with shared resources. Once we have committed to a given high-level goal, it must be broken down to the low-level commands understandable by the various subsystems (similar to hierarchical task network planning). However, rather than using rigid sequences with fixed command start times, we allow specification of flexible sequences that can be altered to accommodate run-time variations in the use of the resources.

We develop these capabilities as prototype extensions to the Virtual Machine Language (VML) [Grasso 2008] execution system. VML is advanced, multi-mission flight and ground software developed for NASA flown on a number of past and current missions including: The Spitzer Space Telescope, Mars Odyssey, Stardust, Genesis, Mars Reconnaissance Orbiter, Phoenix, and Dawn. The prototype concept of a resource constraint is added to the language for both planning and execution purposes. A new VML task, the Goal Manager (GM), maintains the set of requested goals, their priorities, and their interactions (i.e. shared resource constraints). When a goal is submitted, the GM task quickly analyses 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 satisfied by spawning the corresponding VML sequences. At this point, we are committed to the goal, and normal VML processing takes over.

The motivation for our work came from scenarios for operating the Earth Observing One satellite [GSFC], specifically from operations conducted by the Autonomous Sciencecraft (ASE) flight and ground software [Chien et al. 2005]. We demonstrate our prototype implementation on these scenarios. In our approach, goal selection is postponed until the latest possible time, allowing goals to be added, removed or changed just prior to execution. This dynamic goal set enables additional autonomy capabilities such as on-board and ground-based event triggering, similar to ASE. For example, images taken of the Earth can be processed on-board 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.

In these scenarios, start times of goals are assumed to be fixed. 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. 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.

Resource Constraints

A resource constraint is a value and a bound on that value over a period of time. Resource constraints exist as part of goals, activities, or sequences. 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). Resource constraints have the following attributes:

```
ResourceConstraint
  <IdType, TimeType, ValueType>
{
  IdType          id;
  ResourceType    type;
  TimeType        start;
  TimeType        end;
  ValueType       value;
  ValueType       initial;
  ValueType       min;
  ValueType       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 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.

There are four fundamental types of resource constraints:

```
enum ResourceType
{
  Producer,
  Consumer,
  Assigner,
  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. 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).

Resource Values and Operators

A resource constraint can be defined for any type of value as long as the following set of operators is available:

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

These operators allow us to compute resource values from a set of interacting resource constraints, as well as to test for 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).

We have demonstrated six common types of resource values:

```
  int
  double
  string
  set<int>
  set<double>
  set<string>
```

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 the sets are of equal size and 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.

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(bool* 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. A 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, resource consumption and production only differ 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. non-interruptible) 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 as follows:

```
bool ResourceProduce(IdType id,
                     ValueType val,
                     ValueType min,
                     ValueType max,
                     TimeType t = 0);

bool ResourceConsume(IdType id,
                     ValueType val,
                     ValueType min,
                     ValueType max,
                     TimeType t = 0);

bool ResourceAssign(IdType id,
                    ValueType val,
                    ValueType min,
                    ValueType max,
                    TimeType t = 0);

bool ResourceCheck(IdType id,
                   ValueType min,
                   ValueType max,
                   TimeType t = 0);
```

`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. `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 performed. They return false if the bounding constraint 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.

Goals

A goal represents the request for execution of an activity or sequence. Goals have the following attributes:

```
Goal
  <IdType, TimeType, PriorityType>
{
  IdType                    id;
  PriorityType              priority;
  TimeType                  start;
  TimeType                  end;
  set<ResourceConstraint> constraints;
}
```

The `id` is used to uniquely identify the goal. The goal `priority` is used to rank goals. The `start` and `end` values specify the expected temporal scope of the goal. Due to 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 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.

Example: 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: controlling a file system. This example is of particular interest to us because

```
// goal for creating a file
FileHandleResource::FileHandleResource(Producer, end, 1));
  resources.insert(new FileMemoryResource::FileMemoryResource(Consumer, start, size));
  : Resource<int, int, int>(2, type, time, infinity, value, 1024, 0, 1024){}

// resource constraint for the current set of files on disk
FileSetResource::FileSetResource(ResourceType type, int time, set<int> value)
  : Resource<int, int, set<int> >(3, type, time, infinity, value, {}, {}, {infinity}){}

// resource constraint to prevent reads and writes from overlapping
FileReadWriteResource::FileReadWriteResource(int start, int end)
  : Resource<int, int, int>(4, Producer, start, end, 1, 0, 0, 1){}
```

Figure 1

```
// goal for creating a file
FileHandleResource::FileHandleResource(ResourceType type, int time, int value)
  : ResourceConstraint<int, int, int>(1, type, time, infinity, value, 100, 0, 100){}

// resource constraint for the 1024 available Kbytes of memory
FileMemoryResource::FileMemoryResource(ResourceType type, int time, int value)
  : ResourceConstraint<int, int, int>(2, type, time, infinity, value, 1024, 0, 1024){}

// resource constraint for the current set of files on disk
FileSetResource::FileSetResource(ResourceType type, int time, set<int> value)
  :  ResourceConstraint<int,  int,  set<int>>(3,  type,  time,  infinity,  value,  {},  {},
{infinity}){}

// resource constraint to prevent reads and writes from overlapping
FileReadWriteResource::FileReadWriteResource(int start, int end)
  : ResourceConstraint<int, int, int>(4, Producer, start, end, 1, 0, 0, 1){}
```

Figure 2

many spacecraft (including EO-1) must deal with managing data products on an onboard file system. Typically, science activities write data to the file system while engineering procedures read from files for downlink and delete files 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.

We can model this file system with five goals and four resource constraints, shown in pseudo C++ code in Figures 1 and 2. A general resource constraint constructor takes 8 arguments to specify values for the 8 attributes: `id`, `type`, `start`, `end`, `value`, `initial`, `min` and `max`. A subclass is defined for each of the 4 resource constraints in this example. A general goal constructor takes 4 arguments to specify values for the first 4 goal attributes: `id`, `priority`, `start`, and `end`. A subclass is defined for each of the 5 goal types in this example.

Each goal subclass constructor creates and adds the resource constraints for that goal type. Creating a file consumes one of the 100 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 file memory (1024K capacity) 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 subclass also defines the required method for executing the goal. For example, the `FileCreateGoal` might call `fopen()` on a Unix operating system.

Goal Selection Algorithm

We present an algorithm for selecting goals with oversubscribed resources. The pseudo-code is shown in Figure 3. The algorithm can be categorized as a repair-based approach with no search – the constraints and

```
 1  void dispatch(minStart, maxStart)
 2    for each Goal g in allGoalsSortedByTime
 3      with startTime(g) >= minStart and startTime(g) <= maxStart
 4      if g is in selectedGoals
 5        start(g)
 6
 7  void addGoal(Goal g)
 8    for each Goal ag in allGoals
 9      if g interacts with ag
10        add g to interacting goals of ag
11        add ag to interacting goals of g
12    add g to allGoals, allGoalsSortedByTime, and allGoalsSortedByPriority
13    updateSelectedGoals(g)
14
15  void removeGoal(Goal g)
16    remove g from allGoals, allGoalsSortedByTime, and allGoalsSortedByPriority
17    updateSelectedGoals(g)
18    for each Goal ag in allGoals
19      remove g from interacting goals of ag
20
21  void updateSelectedGoals(g)
22    for each Goal sg in selectedGoals
23      with priority <= priority of g
24      remove sg from selectedGoals
25    for each Goal ag in allGoalsSortedByPriority
26      with priority <= priority of g
27      if wasStarted(ag) or isBestGoal(ag, selectedGoals)
28        add ag to selectedGoals
29
30  bool isBestGoal(g, selectedGoals)
31    initialize each Resource used by g
32    add affects of g to each Resource used by g
33    for each Goal ig in interacting goals of g
34      if ig is in selectedGoals
35        for each Resource r shared by ig and g
36          add affects of ig to r
37          if r is invalid
38            return false
39    return true
```

Figure 3

priorities define exactly which goals to choose. We focus on the re-selection that is required when the requested goal set changes, either by adding a new goal or removing an existing goal[1]. While making selections, goal parameter values (e.g. start times) are assumed to be fixed. The result is a set of non-conflicting "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 dispatch algorithm maintains data structures that allow imminent, selected goals to be found efficiently. One set of goals is sorted by start time so that it can quickly find the next goal to dispatch. For each goal, the algorithm maintains a set of interacting goals (i.e. share a resource) so that it can quickly determine whether or not a goal should be selected. The goals are also sorted by priority so that the algorithm can focus on only the interacting goals that are higher priority. 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 Figure 4 we

[1]Changing parameters of a goal (e.g. start time, priority) can be implemented by removing the goal and adding it back with new values.

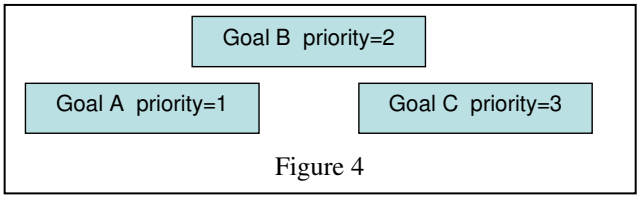

Figure 4

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.

Looking at the pseudo-code, adding and removing a goal each has 3 parts: updating the sets of interacting goals (lines 8-11 and 18-19), updating the sorted goal sets (lines 12 and 16), and updating the selected goal set (lines 13 and 17). When updating the selected goal set, we first de-conflict the schedule by removing all goals with lower priority than the goal being added or removed (lines 22-24). Higher priority goals are unaffected and can remain selected. Next, we re-evaluate each of the lower-priority goals (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, then `g` will not be selected (lines 37-38). Finally, dispatching goals within a given time range simply involves finding selected goals that fall in the range (lines 1-5). The dispatch function is intended to be called periodically with a small time range falling in the near future.

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 (details not shown in pseudo-code). As a final note, the definition of "interacting" could be any arbitrary function that takes a pair of goals as input, and returns true or false. In this work, resources define which goals interact.

Algorithm Analysis

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

$$O(MlgM + N(lgM + X_i(lgM + S_i)))$$

where N is the number of all goals, M is the number of selected goals, X_i is the number of interacting goals for goal i, and S_i is the number of resources shared by goal i. Goals are stored in tree data structures with a log-based lookup. The first term comes from removing lower-priority goals from the selected goals (lines 22-24). The longer second term comes from re-selecting 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 ($M == N$):

$$O(NlgN + N(lgN + N(lgN + R)))$$

Or:

$$O(NlgN + NlgN + N^2lgN + N^2R)$$

Since N^2lgN dominates $NlgN$:

$$O(N^2lgN + N^2R)$$

And assuming that $N >> R$, we have:

$$O(N^2lgN)$$

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 importantly, 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:

$$\Theta(NlgN)$$

Finally, this selection process is performed only when goals are added (line 13) or removed (line 17) from the dispatcher, which is assumed will be done at non-critical times.

Once we have cached the best goals, checking a specific goal is a simple lookup in the set. So dispatching a selected goal for execution (lines 1-5) is:

$$O(lgN + lgM)$$

The first term is from the lookup for goals due for execution (lines 2-3) which we assume to be small (typically one). The second term is from the lookup in the set of selected goals (line 4). Assuming worst case, where all goals are selected ($M == N$), we get:

$$O(lgN)$$

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. While 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 the provided set of 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 an orbit, but can not select an alternate overflight 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 trumped 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 capabilities we are offering far exceed what is available on typical spacecraft today, either implemented in general commanding capabilities or custom flight software. Specifically, they are capable of representing the goal replacement capabilities currently operational in ASE.

Providing goal selection capabilities places some requirements on users when defining goals compared to 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 scope of the resource use.

Planning and Execution with VML

Designed as a multi-mission application, VML is one of the most advanced onboard execution systems in widespread use for NASA missions [Grasso 2008]. Missions currently using VML include Odyssey, Spitzer, Dawn, MRO, and Phoenix. On these missions, 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 VML task, the Goal Manager (GM), implements the goal selection algorithm, and calls 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 [Grasso 2004] capabilities, we employ 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. VML sequences for a goal or activity call other VML sequences that implement sub-activities or executable commands. When appropriate, command 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.

Autonomous Spacecraft Operations

Our work was motivated by scenarios taken from the Autonomous Sciencecraft Experiment (ASE) used in operating the Earth-Observing 1 (EO-1) satellite. A prototype was implemented and tested on these scenarios. Figure 5 shows the system diagram for the prototype. In ASE scenarios, on-board science processing may generate new goal requests [Chien et al., April 2005]. Ground-based sensorweb processing may do the same using uplinked commands [Chien et al., June 2005]. We simulate goal request changes using a time-tagged file containing the change specifications. The EO-1 model consists of VML functions 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 on-board resources (e.g. science data storage) are oversubscribed. At runtime, a simple spacecraft simulator was used to mimic command behavior, including effects on resources. We have not yet run formal benchmarking experiments, but anecdotally the additional processing required for goal selection does not seem to have a significant impact on performance.

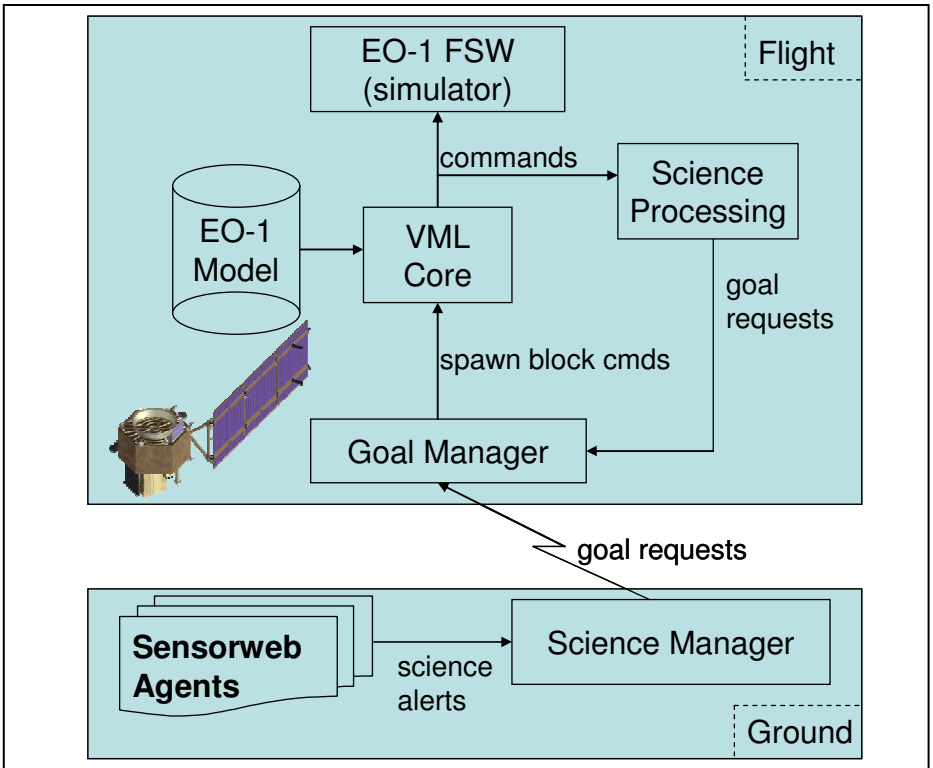

Figure 5

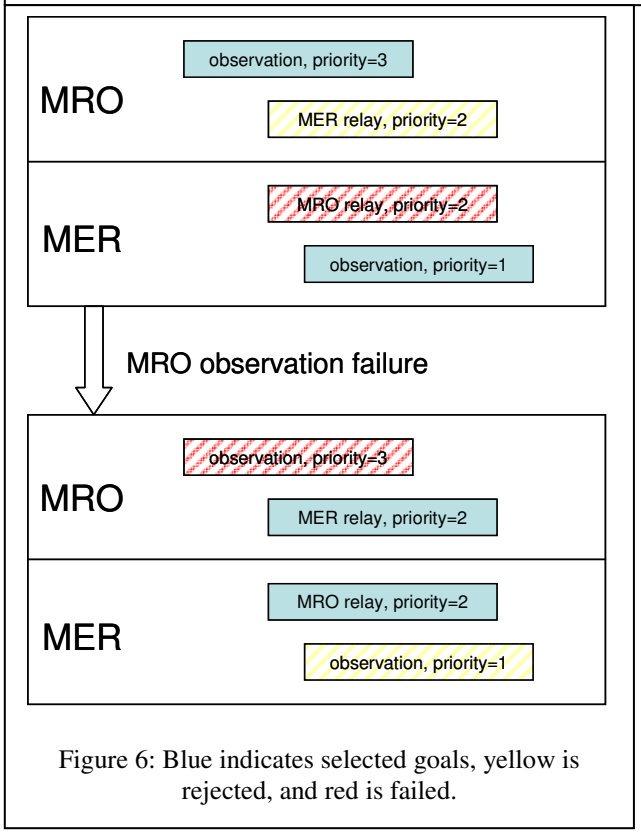

Figure 6: Blue indicates selected goals, yellow is rejected, and red is failed.

We also studied planning and sequencing problems from the Mars Reconnaissance Orbiter (MRO) and the Mars Exploration Rover (MER) missions. From these, we identified several scenarios that might benefit from this technology. First, we examine a data relay scenario, where low-priority MER data can be sent to Earth via MRO (Figure 6). Initially, MRO can not 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 (Figure 7). Initially, MRO has science goals planned, and goals for MER relay opportunities are stored on-board 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 (Figure 8). 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, scientist might choose longer durations or higher resolutions (i.e. larger image size). On-board 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 re-checked, 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

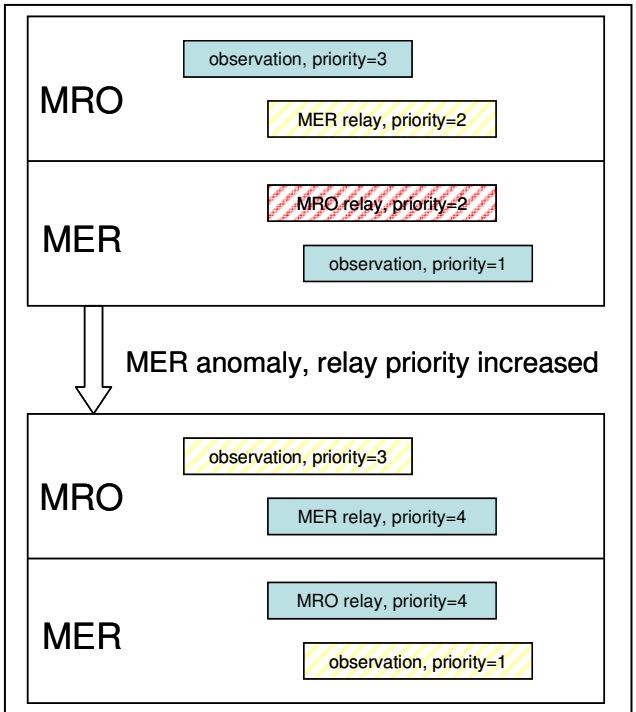

Figure 7: Blue indicates selected goals, yellow is rejected, and red is failed.

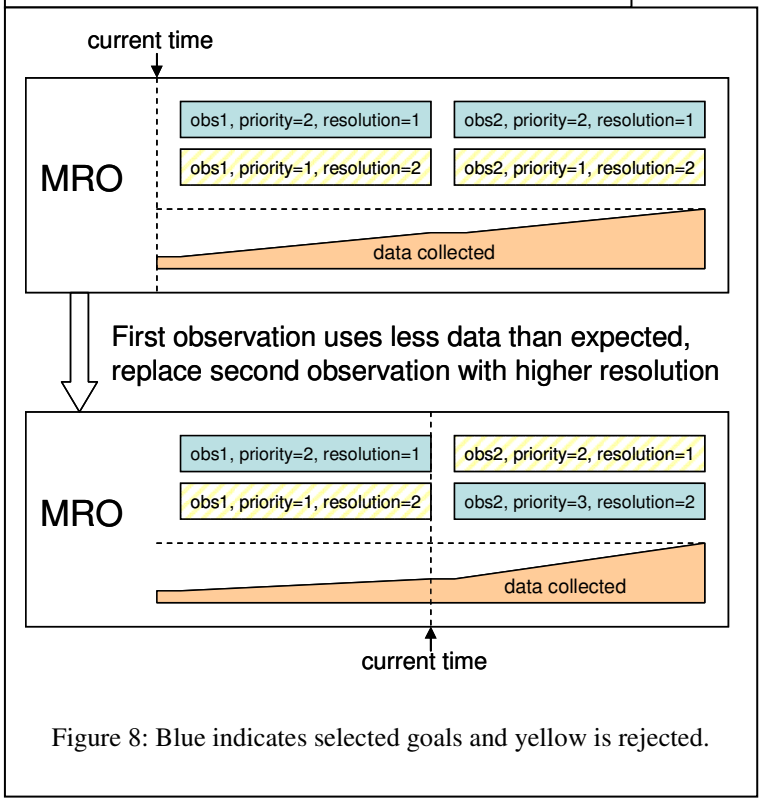

Figure 8: Blue indicates selected goals and yellow is rejected.

pointing the instrument. Parameter calculations on the ground can be triggered from ephemeris changes, automatically generating a new file uplink request, and changing the on-board goal set.

Related Work

Much of the research in planning and goal selection has focused on more general, intractable problems. For example, [Smith 2004] 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). Both the Squeaky Wheel Optimization [Joslin and Clements 1999] and the Task Swapping [Kramer and Smith 2004] algorithms have been shown to improve oversubscribed schedules by re-scheduling tasks to allow more goals to fit. Instead, we look at a more constrained problem that can be solved in polynomial time, while still providing advanced autonomy capabilities useful for many embedded applications.

A considerable amount of work has been done in the area of online planning and execution. We list some of the implemented systems here. For example, SCL [ICS] provides a procedural language for spacecraft commanding similar to VML. VML blocks and the constructs they use are very similar to SCL scripts and constructs. Both provide functions, conditionals, variables, and loops. SCL, however, has a large component that allows the user to build and maintain a telemetry database. This is outside the scope of our work, which instead focuses on intelligent commanding.

ESL [Gat 1996] is an execution language for autonomous agents, implemented as an extension to the Common Lisp programming language. Therefore, programs in ESL can include any of the constructs provided by Lisp. Instead, we extend the VML feature set to include constructs similar to some found in ESL. By maintaining a limited set of features, we hope to contain the effort required to verify programs or sequences.

TDL [Simmons et al.] extends the C++ programming language to include the concept of a task. Like ESL, programs in TDL can take advantage of the generic language on which they are based. Providing such a rich language, however, has a cost when it comes to verifying programs written in that language. The capabilities we propose, while not as powerful, are similar to some provided by the TDL task types and constraints.

Titan [Williams et al. 2003] and Kirk [Kim et al. 2001] are model-based executives. That is, they use a declarative specification of system behavior (plant model) to track system state and compute desired sequences of control actions. Titan combines a deductive controller (evolved from Livingstone, a mode-identification and reconfiguration capability flight validated on DS-1 [Williams and Nayak, 1996]) with a procedural state-based control sequencing module that allows for the specification of desired system state

trajectories. By coupling deductive reasoning capabilities with the task control capabilities of an advanced procedural sequencer, the resulting executive demonstrates greater flexibility, fault-awareness and robustness than purely procedural executives. The Kirk executive provides additional capabilities for reasoning about activity-level contingencies, scheduling, and path planning.

The Mission Data System (MDS) [Dvorak et al. 1999, Barrett et al. 2004] 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, while MDS state variables are used to model both the system under control and the intent of the control system.

Future Directions and Conclusions

This work targets development and maturation of more advanced onboard resource reasoning capabilities to be available to future space missions via VML. While the more typical orbital mission operations could benefit from onboard resource reasoning, future mission 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 run-time planning within a limited computational environment. This capability enables fast incremental selection of goals oversubscribing available resources in a strict priority order. We have presented a computational complexity analysis of these algorithms as well as described its application to a number of typical spacecraft operations scenarios.

Acknowledgements

This work was performed by the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration. A special thanks to Chris Grasso for VML software and support.

References

A. Barrett, R. Knight, R. Morris, R. Rasmussen, Mission Planning and Execution Within the Mission Data System, *International Workshop on Planning and Scheduling for Space (IWPSS)*, Darmstadt, Germany, June 2004.

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

S. Chien, B. Cichy, A. Davies, D. Tran, G. Rabideau, R. Castano, R. Sherwood, D. Mandl, S. Frye, S. Shulman, J. Jones, S. Grosvenor, An Autonomous Earth-Observing Sensorweb, *IEEE Intelligent Systems*, May/June 2005.

D. Dvorak, R. Rasmussen, G. Reeves, and A. Sacks. Software architecture themes in JPL's Mission Data System. *In Proceedings of the AIAA Guidance, Navigation, and Control Conference*, number AIAA-99-4553, 1999.

E. Gat. ESL: A language for supporting robust plan execution in embedded autonomous agents. *AAAI Fall Symposium: Issues in Plan Execution*, Cambridge, MA, 1996.

Goddard Space Flight Center, The Earth Observing One Mission Page, eo1.gsfc.nasa.gov.

C. Grasso. Virtual Machine Language (VML) v2.0 Users Guide. *JPL Document*, D-28342, June 2, 2004.

C. Grasso, P. Lock. VML Sequencing: Growing Capabilities over Multiple Missions. *In Proceedings of SpaceOps 08*. Heidelberg, Germany. May 2008.

Interface and Control Systems (ICS), Inc., http://www.interfacecontrol.com

D. E. Joslin and D. P. Clements, "Squeaky Wheel" Optimization, *Journal of Artificial Intelligence Research* (1999), 10:353-373.

P. Kim, B. Williams, and M. Abramson. Executing reactive, model-based programs through graph-based temporal planning. *In Proceedings of the International Joint Conference on Artificial Intelligence*, 2001.

Kramer, L. A., and Smith, S. F., Task swapping for schedule improvement, a broader analysis. *In Proc. 14th Int'l Conf. on Automated Planning and Scheduling*, 2004.

R. Simmons and D. Apfelbaum. TDL Quick-Reference Manual (v1.3.2). http://www-2.cs.cmu.edu/~tdl/tdl.html, 2002.

B.C. Williams, M.D. Ingham, S.H. Chung, and P.H. Elliott. Model-based Programming of Intelligent Embedded Systems and Robotic Space Explorers. *In Proceedings of the IEEE, Special Issue on Modeling and Design of Embedded Software*, Vol. 91, No. 1, Jan. 2003, pp. 212-237.

B.C. Williams and P.P. Nayak. A Model-based Approach to Reactive Self-Configuring Systems. *In Proceedings of the AAAI-96*, pp. 971-978, 1996.