Mission Planning and Execution Within the Mission Data System

Anthony Barrett, Russell Knight, Richard Morris, Robert Rasmussen

Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive, M/S 126-347
Pasadena, CA 91109-8099
{Anthony.Barrett, Russell.Knight, John.R.Morris, Robert.Rasmussen}@jpl.nasa.gov

Abstract. Not only has the number of launched spacecraft per year exploded recently, but spacecraft are also getting progressively more complex as instruments become more capable and flyby missions give way to remote orbiters, then to rovers and other *in situ* explorers. To address the software issues in this expanding mission set, JPL started the Mission Data System (MDS) project – an effort to make engineering flight software more straightforward and less prone to error through the explicit modeling of spacecraft state. This paper presents how MDS performs mission planning and execution in the context of managing explicit spacecraft state.

1 Introduction

As instruments become more complex and *in situ* missions become more prevalent, the standard approach to control spacecraft with predictable time-tagged command sequences falls apart due to growing spacecraft sensitivity to its dynamic partially-understood environment. For instance, a Mars rover never knows exactly how much time and energy it takes to traverse to a target. There is no way to determine the intervening ground's looseness prior to actually driving over it. Currently such uncertainties condense operations cycles and force operators to work odd hours. For example, MER's operations cycle forces operators to digest the results of one command sequence and generate the next one while a rover sleeps during the Martian night – a 24.6-hour cycle.

One of the Mission Data System's (MDS) underlying objectives is to provide a less stressful approach toward handling uncertainty. Instead of commanding a spacecraft with predictable command sequences, MDS will control it with goals that capture an operator's intent and drive the spacecraft to perform fault-tolerant behaviors that locally adapt to environmental uncertainty. As hinted at in Figure 1, the MDS architecture addresses uncertainty by explicitly representing system state with its certainty and modeling how state knowledge evolves over time. Essentially state estimation is kept separate from control in order to force the explicit determination of state knowledge with its certainty. By separating estimation and control, MDS both facilitates logging state estimates for subsequent retrieval and facilitates a principled approach toward control where certainty assumptions are made explicit. These properties are required when managing remote spacecraft. Logs facilitate diagnosing unexpected problems, and certainty knowledge is required to facilitate fault-tolerant control in the face of partially understood environments.

Following this feedback-control-centered paradigm for reasoning about a system, MDS's Mission Planning & Execution (MPE) subsystem manipulates constraints on state variables instead of command sequences. Where other planning, scheduling, and execution architectures focus on representing actions one way to plan/schedule and another way for execution, the MPE only represents constraints on state, and controllers subsequently enforce these constraints. Not only does this approach remove consistency issues that derive from two models of each action, but it also enables a straightforward way to merge activity by simply merging constraints.

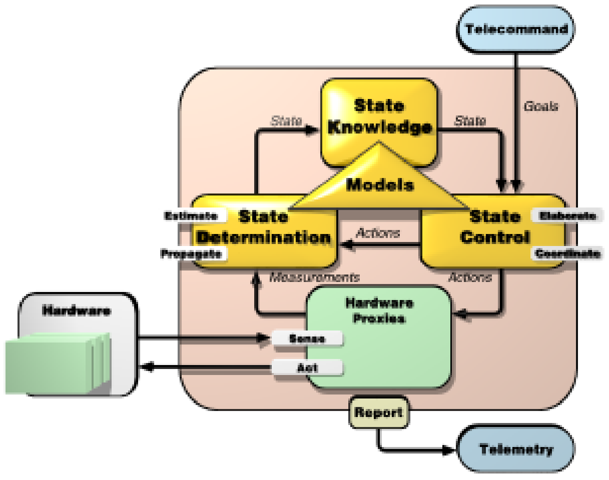

Figure 1. Diagram of the MDS architecture emphasizing the central role of explicitly represented state knowledge and models, goal directed operation, and the separation of state determination from control in all feedback loops.

This paper describes MDS's MPE by first explaining the state and constraint approach to representing plans and problems. With this representation, the subsequent four sections describe the MPE component architecture and how the components combine to implement a simple approach to planning, scheduling, execution, and failure response. Finally, the last three sections describe related work, future work, and conclude.

2 Plan & Problem Representation

At its heart, MDS represents a spacecraft as an interacting set of state variables (Dvorak, Rasmussen, and Starbird 2002). From a temperature to the data products in a file system, each variable denotes an aspect of the spacecraft, its environment, or its supporting ground system that has some measurable impact on mission objectives. The interactions within this set of variables is described in a *state effects model*. For instance, Figure 2 illustrates three such variables that impact whether or not data gets transmitted to ground, where arrows represent the fact that one variable affects another. In this case a transmitter is either off or sending data, the temperature is measured in degrees Celsius, and the heater can be either on, off, stuck on, or stuck off. Since the transmitter only works when warm, the temperature variable affects it. Similarly, an active heater is needed to control the temperature.

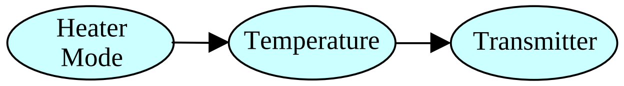

Figure 2. Three interacting state variables

Given a set of state variables representing a spacecraft, the MPE constrains variable values over time intervals to control the spacecraft, where a temporally bound constraint is called a *goal*. For instance, Figure 3 contains an ordered pair of goals for controlling the temperature state variable to warm to within a specified ten-degree range and then stay in that range for ten minutes. As illustrated, goals are connected in a goal network, where timepoints bound each goal's interval and temporal constraints control the separation between timepoints. In this case, the first goal can last from 100 up to 900 seconds, and the second must last for ten minutes. In general, timepoints are only ordered when explicitly constrained or when the ordering affects the evolution of some state variable's value.

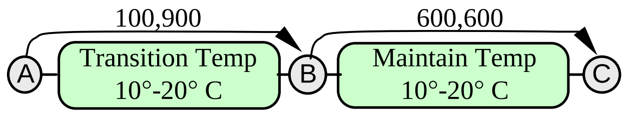

Figure 3. Example goal network with goals on the temperature state variable and min,max time separations between timepoints (in seconds).

While full goal networks can be manually generated, in practice such manual methods are both laborious and error prone due to the interactions between state variables. For this reason a method for automatically elaborating goals was introduced to generate a goal network from a small set of user specified constraints. For instance, Figure 4 shows tactics for elaborating a constraint to transmit data with supporting constraints to have the temperature within a prerequisite range during transmission (a) and that the heater is on (b&c). Thus, a constraint to transmit data for ten minutes captures an operator's intent, and elaboration automatically fills in the supporting constraints.

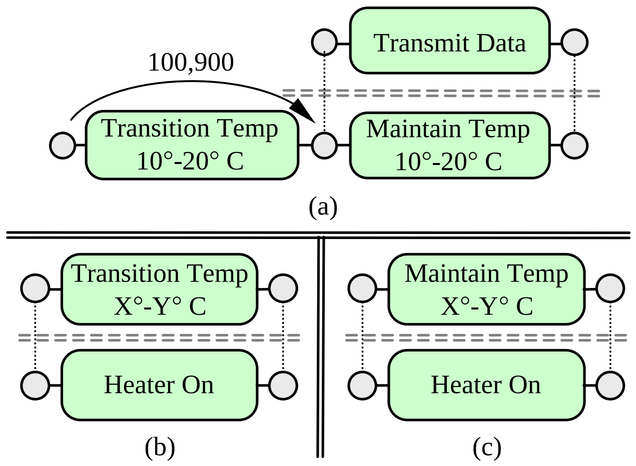

Figure 4. Elaboration tactics supporting a transmit data constraint on a transmitter state variable

As terminology used to describe elaboration implies, elaboration is like precondition achievement planning. Just as precondition achievement planning adds actions to set up conditions for executing other actions to achieve a target situation, elaboration adds goals to enable enforcing other goals to ultimately enable enforcing a target goal. The differences between the two approaches include the fact that the desired goal and its supporting goals are often enforced simultaneously. Also, this constraint-centered approach facilitates merging different, but compatible, goals that overlap on the same state variable – a feature that related action-centered systems in Table 1 currently have trouble supporting.

Finally, where other planning systems reason about propositions and metric resources, MDS allows mechanisms for creating arbitrary types of state variables. For instance, one state variable would capture the orientation of a rover using a quaternion, and another might capture the state of a memory storage unit with a set of file (name, size) tuples. This generality stems from a focus on application programmer interfaces to implement arbitrary classes of state variables (Knight, Chien, and Rabideau 2001). As such, MDS replaces a focus on textual domain representation languages with a focus on C++ functions to reason about state variables and merge their goals. MDS provides base classes for goals and state variables, and a domain is implemented by creating subclasses and instantiating objects for particular types of state variables and goals.

3 MPE Component Architecture

The most common framework for developing agent architectures within the Artificial Intelligence community is based on beliefs, desires, and intentions (Rao&Georgeff 1995). Within this framework an agent computes beliefs about the world's state by interpreting observations, combines these beliefs with desires that are furnished by an operator in the form of goals, and computes a set of intentions in the form of executable actions that result in satisfying the goals. The main difference among agent architectures revolves around how beliefs, desires, and intentions are computed, represented, and managed. Within the MDS architecture of Figure 1, beliefs are computed in state determination components, where each state variable has a single state determination component, but a single state determination component can estimate the state of multiple variables, and these estimates are managed by the state knowledge components. On the control side, passing constraints to a state variable's single associated controller – which can similarly service multiple state variables – performs intentions. Finally, desires are turned into intentions by the MPE components illustrated in Figure 5.

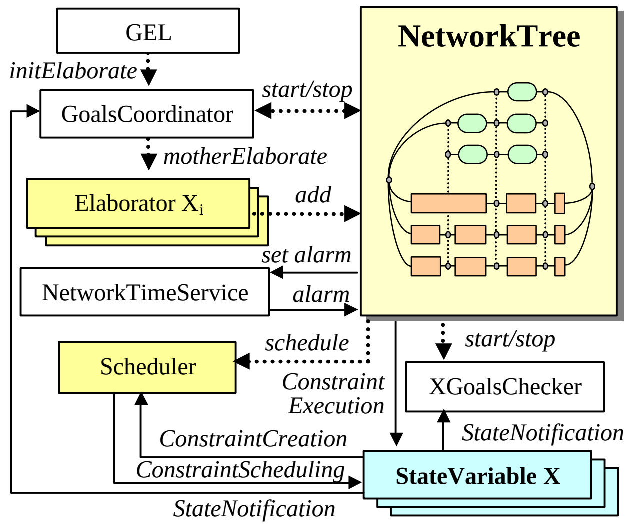

Figure 5. MPE system's component architecture

As illustrated, a goal elaboration language (GEL) interpreter builds a temporally constrained network of commanded goals. This network is put in an *initElaborate* message and passed to the GoalsCoordinator, which starts elaborators for commanded goals. These elaborators both insert goals into the network tree and start other elaborators for supporting goals. Once the goals are fully elaborated, the Scheduler component schedules them for execution, which in turn is mediated by the NetworkTimeService and XgoalsChecker. These two components collectively determine when to pass a constraint to a controller through its associated state variable. Finally, each goal's elaborator component persists through execution in order to facilitate changing a goal's elaboration, if execution problems arise.

4 Elaboration & Scheduling

Although a number of components are involved in elaboration and scheduling, they combine to implement the algorithm listed in Figure 6. At the top level this algorithm consists of four steps: copy the task network, try to add goals to the copy through elaboration, try to schedule the added goals in the context of the current goals, and finally replace the executing network with the newly changed task network. While the first and last steps are straightforward, the middle tasks elaborate and schedule using a backtracking search that branches at exhaustive choice steps. For instance, choosing how to elaborate a goal may have to be revisited if a supporting goal either could not be elaborated or could not be scheduled. Essentially, the three backtracking tests determine if a choice was infeasible and fixes bad choices by undoing a bad choice with subsequent work and making a more informed choice to continue the search.

In general, the MPE framework facilitates building numerous different types of planners/schedulers due to its focus on components. By replacing components, the MPE system can include other constraint elaboration and scheduling approaches with arbitrary amounts of analysis over the NetworkTree to inform a search for a new task network. While MDS currently takes the simplest, least informed, approach toward elaboration and scheduling, the approach is easily changed.

```
Copy executing task net to proposed network for modification
While there are goals to elaborate do
  Choose a goal G@[S,E] to elaborate (heuristically ordered)
  Add goal to proposed network
  Exhaustively choose elaboration tactic M for G
  Apply M, which possibly generates new goals to elaborate
    – Backup (and remove goals) if application of M is illegal
For each state variable SV (from a heuristic ordering) do
  For each new goal G@[S,E] on SV (heuristically ordered) do
    Exhaustively choose how to constrain G's start timepoint S
    Exhaustively choose how to constrain G's end timepoint E
    Merge G@[S,E] into SV's timeline – Backup if merge illegal
  Propagate the expected behavior of SV given the new merges
    – Backup if the propagation is illegal
Replace executing task net with proposed one if it scheduled
```

Figure 6. High-level elaboration and scheduling algorithm description, where a "backup" regresses computation to the last unexhausted choice

4.1 Elaboration

To get a better understanding of elaboration, consider Figure 7 with its illustration of the initial inter-component messages that occur when elaborating an operator's request to transmit data for ten minutes. The first message passes the commanded transmit network to the mother elaborator, which starts elaboration by sending a ProposerRequest message to the network tree and waiting for a response. At this point the MotherElaborator task manages the operator request's elaboration by creating an elaborator for each requested goal and incrementally sending messages that tell it to start. In the case of Elaborator1, messages 5, 6, and 12 first add the commanded goal to the proposed network and

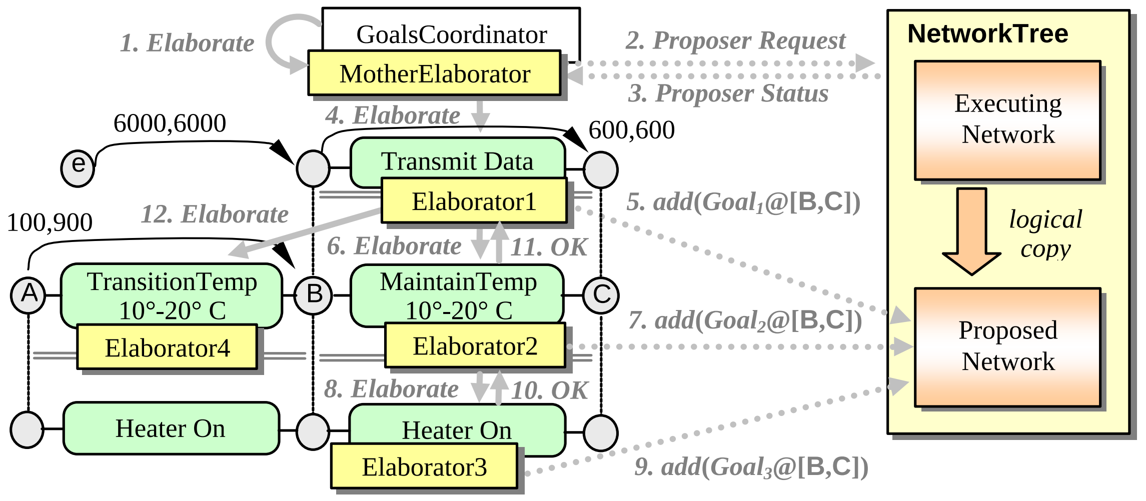

Figure 7. The first twelve out of twenty inter-component messages as components coordinate to elaborate a 600-second transmit data request 6000 seconds after a reference time e*, where dashed arrows denote messages relayed through the GoalsCoordinator.*

subsequently command the elaboration of supporting goals. While not illustrated due to space reasons, each elaborator collects "OK" messages from its children and subsequently signals "OK" to its parent, just like the illustrated "OK" messages of elaborator2 and elaborator3.

Ultimately the mother elaborator informs the goals coordinator of success or failure depending on whether or not all of the mother elaborator's child elaborators signal "OK". The goals coordinator either requests to promote or cancel the local copy, respectively. Even after sending a promote request, elaborators persist to handle subsequent problems that occur when trying to schedule/validate the proposed network and execute a validated network.

While the elaborator for each type of goal can be an arbitrary C++ process, the underlying thread model makes elaborators collectively execute the `WHILE` loop in Figure 6, where each elaborator implements the last three lines in the loop. The goal choice line is defined by how elaborator's send messages to children and await responses. Incrementally sending messages gives elaborators full control, and batch sending gives a thread scheduler full control. Finally, requiring that elaborators relay messages through the GoalsCoordinator limits what they can see and do.

4.2 Scheduling and Promotion

While elaboration was spread over multiple components, all scheduling is performed by the scheduler component shown in Figure 5, which schedules newly added goals once elaboration completes. As the nested `FOR` loops in Figure 6 suggest, scheduling focuses on one state variable at a time and one goal at a time for a given state variable. For instance, Figure 8 illustrates the scheduling of the running example's goals on the temperature state variable timeline upon entering the inner `FOR` loop and after each iteration, where a state variable *timeline* is an executable string of goals (XGoals). In this case the initial timeline contains an "unconstrained" XGoal that gets subsequently cut up by merging in the new goals, and the resulting timeline is checked for consistency using a propagation test. This test validates a variable's XGoal in the context of its previous XGoal.

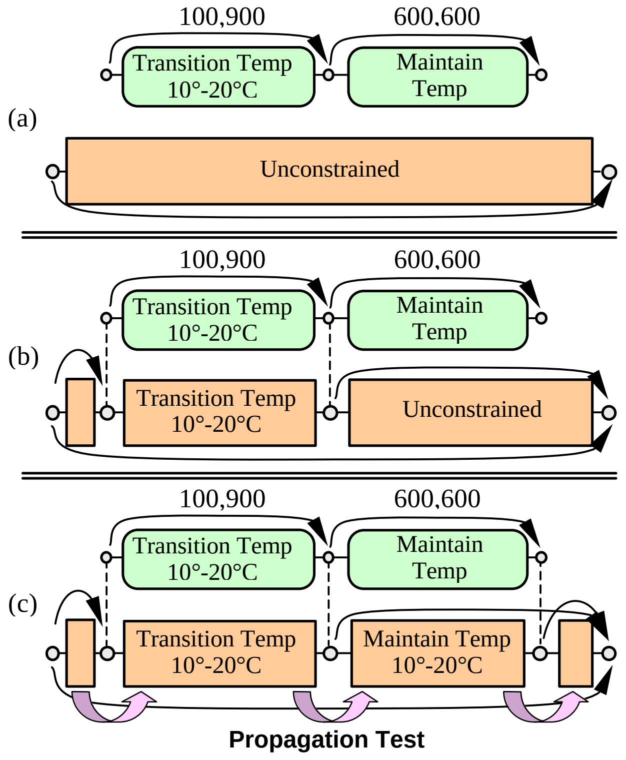

Figure 8. Scheduling temperature state variable goals, where (a) denotes the status prior to scheduling, (b) denotes the status after merging in the first goal, and (c) denotes the status during the propagation test.

In the example, the propagation test passes, because the transitioning from unconstrained to transition-temp to maintain-temp is feasible. If elaboration did not add a transition-temp goal, the unconstrained to maintain-temp transition would fail and cause backtracking within the scheduler and possibly back into elaboration. Finally, trying to combine incompatible goals causes an illegal merge backtrack. For instance, replacing the temperature's unconstrained XGoal with an XGoal to stay between 0° and 50° Celsius still merges with the example goals, but a 0° to 5° range causes conflicts.

Since this elaboration and scheduling are computation intensive, they performed on a copy of the executing task network, which will replace the executing network once scheduling successfully completes, or get discarded if it could not complete. To keep the altered copy consistent with the executing network, the scheduler cannot order any new timepoint before a fired one even if it fired after making the copy. This ensures that the current constraints do not change when replacing the executing task network.

5 Plan Execution

As previously mentioned, plan execution in MDS involves incrementally changing the constraints imposed on state variables. The NetworkTimeService and XgoalsChecker components from Figure 5 combine to facilitate evolving the commanded constraints over time. More precisely, these two components combine to provide a real-time implementation of the algorithm shown in Figure 9 where *firing* a timepoint grounds its time to *now*, passes its subsequent XGoals to state variable controllers, and alters goal failure monitoring. The two components respectively handle temporal constraints and state constraints. In the case of the NetworkTimeService, the NetworkTree sets alarms to signal when unfired timepoints are not constrained into the future and when they are about to time out. The XgoalsChecker manages a set of timepoints that are temporally allowed to fire by progressively testing their subsequent XGoals to see when they are ready to start.

```
For each unfired timepoint TP not constrained into the future do
  If TP's XGoals can start or TP is about to time out then (fire)
    Stop monitoring each goal G@[*,TP]
    For each XGoal XG@[TP,*] do
      Change imposed constraint on XG's state variable to XG
    Start monitoring each goal G@[TP,*]
```

Figure 9. Timepoint firing algorithm for executing task networks

As the algorithm shows, execution mediates the evolution of enforced state variable constraints by firing timepoints. At any given moment a set of zero or more timepoints can fire. Each of these timepoints fires when either its XGoals can be enforced or its temporal constraints force it to fire – possibly leading to a failure condition. When a timepoint fires, its subsequent XGoals are enforced and the monitored goals change. These goal monitors detect failures for elaborators to respond to. Since goal monitor functions are hand crafted for each goal, the actual failure response is locally determined.

5.1 Constrained Timepoint Firing

To improve understanding of the execution algorithm and its component implementation, consider Figure 10 with the result of planning the transmit data example. In this case there is a timeline for each of the three state variables, and they collectively refer to five timepoints: the first and last pre-existed to refer to a temporal reference and the horizon respectively; B and C were added as part of the original request; and A arose during elaboration. Given the temporal constraints, the windows relative to the reference time appear at the bottom.

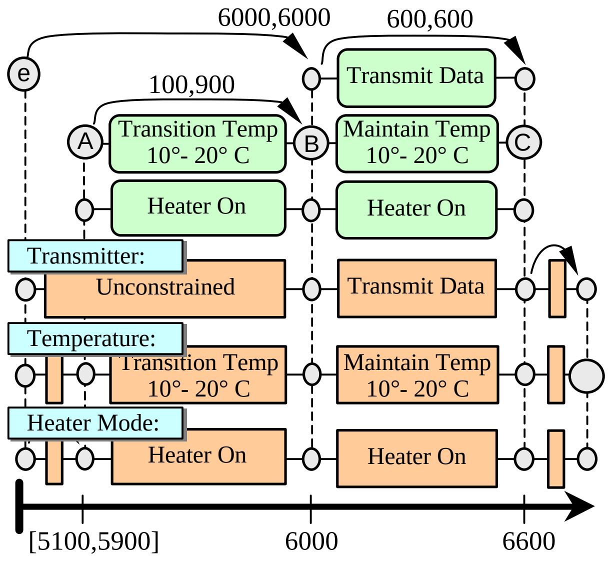

Figure 10. Executable goal network in support of transmitting data for 10 minutes, starting 100 minutes after a reference "epoch" time e

To simplify the algorithm trace, we first ignore the ramifications of goal monitoring, which we return to in the next subsection. The main focus here is to show how to efficiently perform timepoint firing by using interrupts to let the algorithm sleep when no timepoints need to be monitored. The NetworkTimeService provides alarms to facilitate sleeping when all unfired timepoints are constrained into the future. Also, the XgoalsChecker component is event-driven by state notifications. In this way the timepoint firing algorithm sleeps while waiting for a new timepoint or notification.

For instance, the NetworkTree starts executing the task network by setting up an alarm for 5100, when the first timepoint becomes applicable. Until then, reasoning about timepoint firing ceases and the state variables remain unconstrained. At 5100, an alarm triggers the NetworkTree to pass the two XGoals related to timpoint A to the XgoalsChecker. Since one XGoal is a transition and the other is a heater mode requirement, they are both immediately ready to start. Thus, A fires, and the NetworkTree (1) passes the transition and mode XGoals to the temperature and heater mode state variable respectively for relay to their controllers and estimators and (2) alerts the GoalsCoordinator to start monitoring the two goals starting at A. In addition to enforcing/monitoring new constraints, firing A uncovered B, which results in setting a second alarm for 6000. Now reasoning about timepoints ceases since none are pending while the transmitter warms.

When the alarm at 6000 goes off, the NetworkTree triggers and detects a timeout. A timeout immediately fires B, and the NetworkTree then passes the XGoals to the state variables, changes the goal monitors, and sets an alarm for timepoint C at 6600. At this point the state variables are constrained to continually transmit data while reasoning about timepoints ceases again. Finally, at 6600 the alarm triggers the NetworkTree to fire timepoint C, which ends the transmission by unconstraining the three state variables and stopping the monitors. While "unconstrained" intuitively lets a state variable be anything, the controllers and estimators are always active, even when a state variable is unconstrained. Thus, a controller will perform a prudent behavior. In the case of the transmitter variable, the prudent behavior is to keep it off.

5.2 Forced Timepoint Firing and Failure Response

The previous section explicitly avoided discussion about responses when goal monitoring detects a failure. In general, an XGoal is a merge of zero or more constituent goals that appear in the network. Whenever an XGoal's constraint is passed down, the GoalsCoordinator either starts or continues monitoring that XGoal's constituents against state variable value notifications to both detect and respond to failure. Also, a constituent goal G@[S,E]'s monitoring stops when ending timepoint E fires, which also stops the last XGoal affected by G.

For instance, at 5100 in the previous trace timepoint A fired to both pass two XGoals to the heater-mode and temperature variables and start the GoalsCoordinator's passing state notifications to monitor functions in the two starting goals. The monitor in the transition goal detects situations where the temperature will not transition to the target range by some deadline. In general, monitors can perform any user-supplied response, but the default iterates down from the third of the following six responses:

1. Reschedule to delay the goal and its supporting goals;
2. Reschedule to delay the goal that the failed goal supports, progressively moving farther up;
3. Re-elaborate the failed goal and reschedule;
4. Re-elaborate the goal that the failed goal supports and reschedule, progressively moving farther up;
5. Remove the set of operator-requested constraints that resulted in creating the failed goal; and
6. Remove all operator requests and enter a safe state.

While the last response corresponds to the typical approach to put a spacecraft into a safe mode, iterating through the first five facilitate less drastic responses that correspond to how the original elaboration/scheduling algorithm in Figure 6 regressed its computation upon detecting a problem with selections in the exhaustive choice steps. The first two responses corresponded to regressing in the scheduling loop, the second two corresponded to regressing through the elaboration loop, and the fifth choice corresponded to the algorithm's response when it was unable to elaborate and schedule a commanded set of constraints.

Returning to the transmit example, suppose that the heater spontaneously switched off at 5400. Thus, the "heater on" goal's monitor function would detect a failure. In this case, the "heater on" cannot reschedule, but the start of its parent "transition temp" goal can be delayed to after 5400 – subsequently delaying the start of the "heater on" goal. Thus, a reschedule to move timepoint A after 5400 results in a legal task network that tries to turn the heater on again.

While a failure at 5400 can be resolved with a reschedule, such is not the case for a failure at 5910 due to the temporal constraint between timepoints A and B. More precisely, repair efforts first fail to reschedule the "heater on" goal due to its simultaneity constraint with the "transition temperature" parent, second fail to reschedule the "transition temperature" goal due to figure 4(a)'s temporal constraint, and third fail to reschedule the "transmit data" goal due to the commanded constraint that it occur 6000 seconds after epoch. Also, attempts to progressively re-elaborate the "heater on" goal, the "transition temperature" goal, and the "transmit data" goal fail due to each goal having only one elaboration tactic. Thus, the only resolutions are either to completely remove the operator's "transmit data" request or to safe the rover.

To provide an example of re-elaboration, suppose that Figure 11's elaboration was also available for an emergency transmission, lacking goals on temperature, but only transmitting at a low bandwidth. With this extra elaboration, the transmit data request can be re-elaborated, and the resulting task network appears in Figure 12, where the data is transmitted at a lower data rate in an emergency.

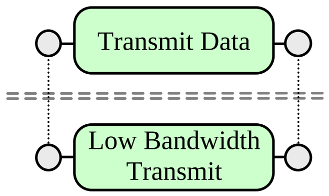

Figure 11. Elaboration tactic to support transmitting data during emergencies when temperature controls fail

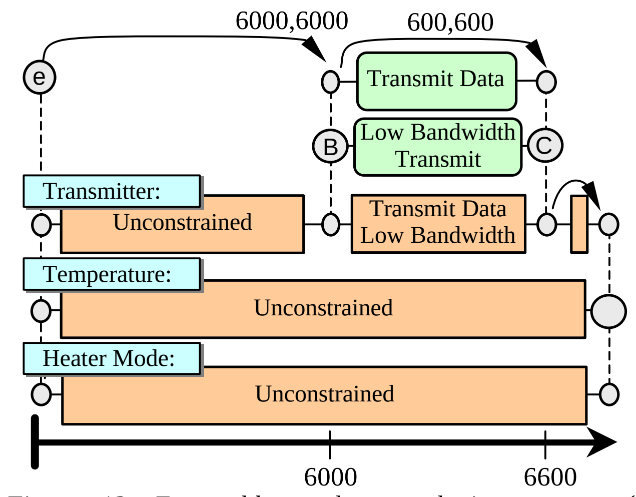

Figure 12. Executable goal network in support of transmitting data for 10 minutes, starting 100 minutes after a reference "epoch" time e

Of course these mechanisms depend on both the ability to elaborate/schedule during execution and to maintain elaboration information on each distinct operator requested set of constraints. Thus, the elaborator tasks introduced in subsection 4.1 persist even after elaborating and scheduling goals in order to facilitate re-elaboration upon detecting a failure.

Finally, in addition to responding to problems after a goal starts, constituent goal monitoring also deals with cases where a goal was not ready to start when its initial timepoint fires. This happens when the timepoint is temporally constrained. For instance, timepoint B fires at 6000, even if the temperature is only 5° C, and the constituent MaintainTemp 10°-20° C goal's monitor function both detects and resolves problem.

6 Observed MPE Activity

While the components back in Figure 5 do combine to implement the elaboration and execution algorithms in Figures 6 and 9, they do so in a manner that facilitates letting the algorithms run in parallel – letting execution continue during elaboration. To facilitate this feature, each component collects messages from other components in a queue and services received messages when activated, where MPE component activation occurs in a round robin fashion on a single thread of execution that coexists with other threads associated with lower level hard real-time functions for state estimation and control.

A number of experiments were developed to test the MPE components on both a Rocky7 rover simulation and the rover itself, and the simplest just (1) waits five seconds, (2) turns the rover 90°, (3) makes it drive forward 3 meters, and finally (4) parks it. Thus, the simplest test has three main goals and five timepoints that elaborate to constrain 54 interacting state variables that model Rocky7. While the ultimate behavior contains four visible activities, the test exercises the full system – including the MPE.

To observe MPE's behavior in this simple test, we measured the amount of wall clock time spent in each component to service its message queue during a round robin iteration, and graphed the results in Figure 13, where actual time is a multiple of the number of cycles with a processor dependent ratio. For instance, this test was preformed in simulation on a Pentium based Linux workstation that performs 3.06E+09 cycles per second. Thus, the longest amount of time spent in any component was less than a third of a second during the test.

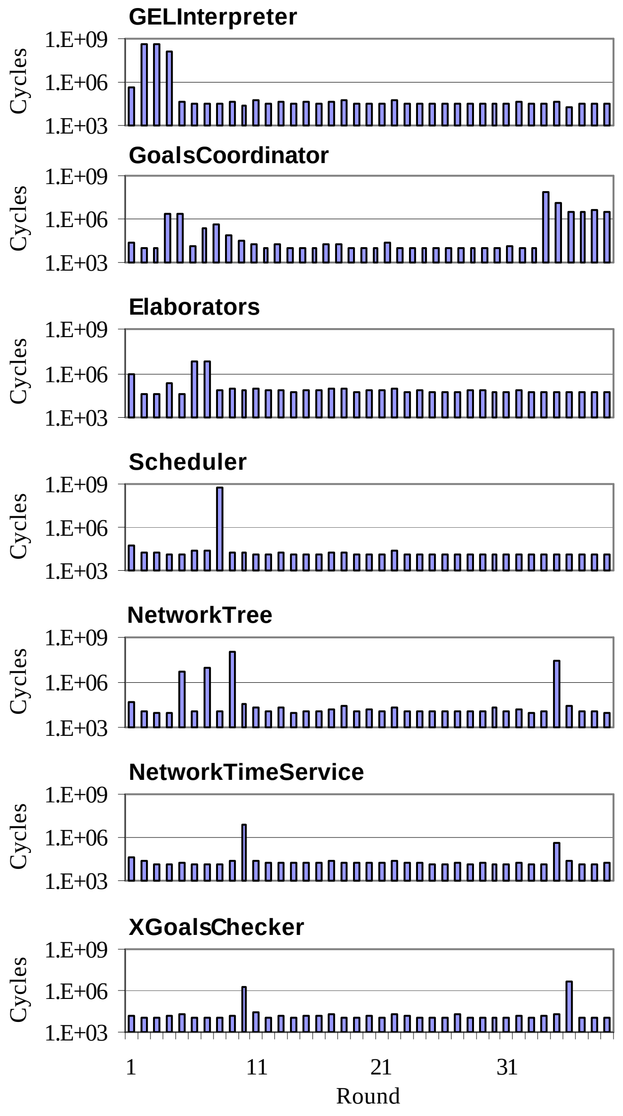

Figure 13. Component activity pattern of the MPE during elaboration, scheduling, and execution

Figure 13 illustrates the first 40 rounds through the MPE components. These rounds trace MPE activity through elaboration, scheduling, and execution up through the start of the turn activity.

- In the first four rounds the GEL interpreter processes command files to pass to the GoalsCoordinator.
- The GoalsCoordinator becomes active at round 4 as it starts elaborators and mediates messages between elaborators and the NetworkTree until round eight.
- The scheduler activates in round eight in order to promote the copied network for execution.
- At round nine the NetworkTree's activity spikes as it promotes the newly scheduled network.
- The NetworkTimeService and XGoalsChecker spike at round ten in order to fire a timepoint that starts a five second wait period, where subsequent rounds have all components servicing empty message queues.
- The GoalsCoordinator, NetworkTree, XGoalsChecker, and NetworkTimeService activate and interact during rounds 35, 36, and 37 in order to orchestrate the second timepoint's firing, where subsequent rounds have the rover turning 90°.
- Finally, subsequent rounds have the GoalsCoordinator servicing state update messages to monitor the turn goal's progress.

In order to simplify explanation, this example only showed the MPE performing one task at a time, but parallelism is possible. The message queues facilitate parallelism by collecting messages from multiple sources and servicing them when a component becomes active. For instance, the GoalsCoordinator can mediate messages during elaboration while monitoring executing goals against state update messages.

7 Related Work

As the BDI framework suggests, most agent architectures consist of three levels: a planner level to turn *desired* goals into *intended* activities; an executive level to perform *intended* activities and collect sensory information into a *believed* system state; and a hardware driver layer to interface to a robot's actual sensors and effectors. While the hardware drivers always function in hard real time, the point where agent architectures differ revolves around layer and interface implementations and resultant real-time guarantees (see Table 1). Systems like CIRCA (Musliner, Durfee, and Shin 1993) and EVAR (Schoppers 1995) provided real-time guarantees across the board, but only applied to relatively inflexible applications for simple tasks. These systems compiled the planner into a control system that always provided the next action to achieve restricted objectives from the currently perceived state. While EVAR used a handcrafted controller for a robot with a single objective to rescue an astronaut, CIRCA had an offline planner to generate a controller from a given objective. On the other hand, the Remote Agent (Muscettola et al. 1998) had an onboard planner to control a flexible spacecraft during a short experiment in autonomy, and the ASE (Chien et al. 2003) similarly used an onboard planner to control a satellite to autonomously collect science observations. CLARAty (Volpe et al. 2001) provides a rover specific control environment for integrating of autonomy research algorithms. MDS provides a unified flight/ground control architecture for complex spacecraft with an application to the Mars Science Lab (MSL) mission, which involves a rover.

First Year	System	Layer R-T Guarantee		Application
		Plan	Exec	
1993	CIRCA	Hard	Hard	PUMA robot arm
1995	EVAR	Hard	Hard	EVA astronaut retrieval Sim.
1998	RAX	None	Soft	DS-1 probe (experiment)
2001	CLARAty	Soft	Soft	Research rovers Rocky7 & 8
2003	ASE	Soft	Hard	EO-1 satellite (experiment)
2004	MDS	Soft	Hard	MSL's unified flight/ground sys

Table 1. Comparing the per layer real-time performance guarantees of planner and executive layers within different autonomous agent architectures (when each was initially defined).

While MDS has similar performance guarantees to a number of other systems. There are a number of points where the MDS architecture is unique. First, the focus on explicitly constraining explicitly known state information derives from a need to keep flight software measurable and forces the dissection of the executive layer into goal monitors, state estimators, and state controllers. Second, the focus on reasoning about state constraints instead of actions facilitates a clean way to merge simultaneous activity by merging multiple goals into a single XGoal. One criticism of MDS's constraint focus might involve representing complex behaviors that would span multiple state variables, like a behavior that would involve camera and wheel variables while driving across Mars with hazard avoidance. The MDS planning layer can orchestrate such complex behaviors involving multiple state variables by linking multiple state variable controllers in what is called a *delegation pattern* where a controller accepts constraints directly from others, but that is beyond the scope of this paper.

8 Future Work

While the MDS architecture currently drives a rover around the JPL Marsyard with the infrastructure mentioned above, a number of improvements are planned to facilitate scaling the MPE up to control the MSL rover when it arrives on Mars in 2010. These improvements include adding capabilities, improving performance, and improving the human-computer interface. On the performance side, memory and CPU measurements on the Rocky rovers are beginning in parallel with analysis of how the actual domain will grow to manage the much more capable MSL rover. On the capability side the most pressing needs involve side-effect reasoning and the responding to situations when the rover is oversubscribed.

8.1 Side-Effect Reasoning

Side-effect reasoning involves modeling how a state variable's constraints affect other variables, which is similar to checking side effects on action centered AI planners. For instance, Figure 14 adds an extra state variable to the running example to capture the fact that the status of available power has an affect on satisfying mission objectives. While the heater's mode affects the current temperature, it also affects available power since an active heater consumes power to raise the temperature. Similarly, a transmitter consumes power in order to send data. Since elaboration starts with a request on the transmitter and only involves state variables that affect the transmitter, effects on the power margin are side effects to the transmission request.

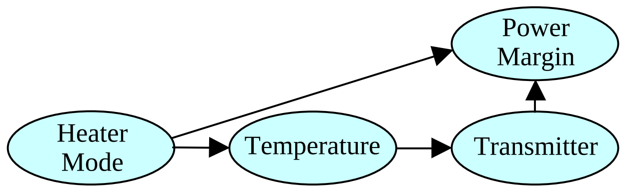

Figure 14. Four interacting state variables

Dealing with side effects starts with extending the propagation function over XGoals on a state variable's timeline from just using the previous XGoal to also using concurrent XGoals on state variables affecting the current state variable. For instance, Figure 15 illustrates that the propagation test on the power margin state variable includes references to the transmitter and heater mode state variables. In this case, the heater requires 10 Watts and the transmitting data requires 25 Watts.

In addition to extending the propagation test, scheduling has to change in order to reflect that the combined side-effects of XGoals depend on the order of their timepoints. Just as before, timepoints that appear on a given state variable's timeline are totally ordered, but now both elaboration *and* side-effects reasoning can put a timepoint on a state variable's timeline. For instance, timepoints A, B, and C appear on Figure 15's power margin timeline because of side effects reasoning. Due to this requirement the scheduling loop in Figure 6 has to be extended into the loop that appears in Figure 16, and the heuristic ordering in which state variables are visited must conform to the partial order defined by the arrows in the state effects model. In the case of Figure 14 there is only one such ordering that starts with the heater mode and ends with the power margin. This ordering is required to avoid propagating over a particular state variable before determining the XGoals of state variables that affect it.

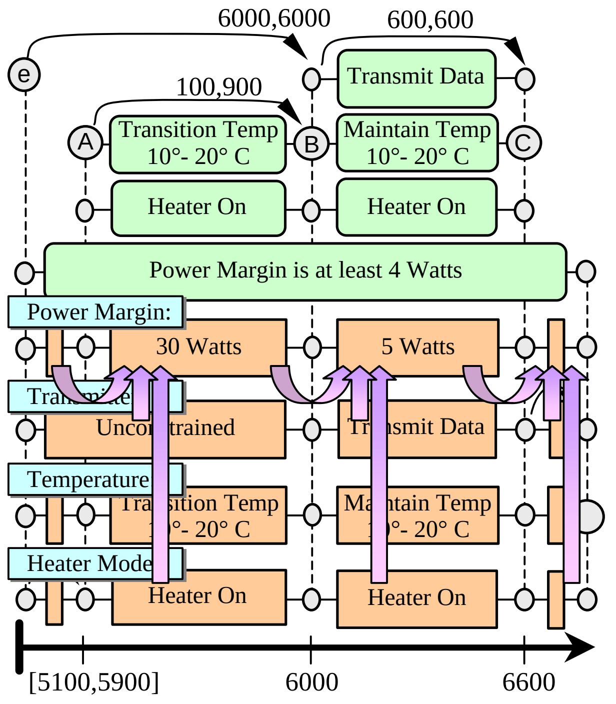

Figure 15. Extending the example to reason about power margin (a resource) during propagation in the scheduler.

```
For each state variable SV (from a heuristic ordering) do
  For each new XGoal XG@[S,E] affecting SV do
    Exhaustively choose how to constrain XG's start timepoint S
    Exhaustively choose how to constrain XG's end timepoint E
    Split XGoals on SV if S and/or E are new to SV's timeline
  For each new goal G@[S,E] on SV (heuristically ordered) do
    Exhaustively choose how to constrain G's start timepoint S
    Exhaustively choose how to constrain G's end timepoint E
    Merge G@[S,E] into SV's timeline – Backup if merge illegal
  Propagate the expected behavior of SV given the new merges
    – Backup if the propagation is illegal
Replace executing task net with proposed one if it scheduled
```

Figure 16. Extending the scheduling algorithm description to include side effect reasoning

Finally, side-effects reasoning complicates failure recovery due to problems determining the actual source of a failure. For instance, suppose that the power margin dipped to 2 Watts between timepoints B and C. The minimum margin goal monitor detects this problem, but there is no direct relationship between this goal and goals that consume power. While other solutions are possible, the easiest and safest resolution is to remove all operator requests and enter a safe state.

8.2 Prioritized Requests

As hinted back in section 2, an operator's request within the MDS framework is a set of state constraints bound by temporally constrained timepoints that gets elaborated and then scheduled into the executing goal network. Unfortunately, a flight project often runs into a situation where its *in situ* explorer's time and resource limitations preclude satisfying all requests of it – a rover can only visit so many rocks and transmit so much data a week. Thus, requests are prioritized and elaborated/scheduled in priority order. Currently, those requests that fail to schedule are dropped by the MPE system, but that is soon to change.

Instead of dropping requests, the MPE will maintain a list of "pending" unscheduled requests that it will try to schedule upon discovering unexpected resources and time. This change involves keeping requests that fail to schedule, marking them as "pending", and then activating Figure 17's algorithm upon discovering extra time and resources. Such a discovery happens when either another request fails or a goal's monitor function detects under-utilization. For instance, the example power margin goal's monitor function might fire up the algorithm

```
Delete "pending" operator requests with timed out timepoints
For each "pending" operator request REQ in priority order do
  Try to elaborate/schedule REQ (see Figures 6 and 16)
  Mark REQ as not "pending" if it scheduled
```

Figure 17. Using the elaboration/scheduling algorithm to maintain an overflow list of pending goal subnets

In general, there are two ways for a request to become pending: either fail to be initially elaborated and scheduled or fail and be removed during execution. As Figure 17 suggests, there are also two ways for an operator request to leave the pending list: either to timeout and be dropped or to be successfully elaborated and scheduled. The timeout option occurs whenever a request contains a timepoint that will never schedule due to being temporally constrained to fire in the past.

9 Conclusions

This paper discussed the autonomy architecture of the Mission Data System with an emphasis on how the MPE implements the planning layer. In MDS a spacecraft is modeled as an interacting set of state variables, where each variable is associated with a single estimator and controller within an execution layer. An MPE plan is represented as a consistent network of timepoints connected by temporal and state constraints. While temporal consistency requires the existence of a timepoint grounding that satisfies all temporal constraints, state consistency holds when no such temporal grounding results in simultaneous state variable constraints that violate the model of how state variables interact. Given such a plan, the MPE progressively grounds timepoints to the current time in order to evolve the active set of variable constraints. These evolving constraints control the spacecraft to satisfy both science observation and health maintenance objectives.

Acknowledgements

This work was performed at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. The authors would also like to thank Steve Chien, Dan Dvorak, Erann Gat, Kim Gostelow, Bob Keller, and Tom Starbird for significant contributions to the MPE during its formative stages as well as discussions with James Kurien, Mark Drummond, and Michael Freed. Finally, discussions with Matthew Bennett were key to documenting the approach to side effect reasoning, and Lin Song provided the timing infrastructure that enables measuring MPE component activity.

References

S. Chien, R. Sherwood, D. Tran, R. Castano, B. Cichy, A. Davies, G. Rabideau, N. Tang, M. Burl, D. Mandl, S. Frye, J. Hengemihle, J. Agostino, R. Bote, B. Trout, S. Shulman, S. Ungar, J. Van Gaasbeck, D. Boyer, M. Griffin, H. Burke, R. Greeley, T. Doggett, K. Williams, V. Baker, J. Dohm. 2003. "Autonomous Science on the EO-1 Mission," In Proceedings of the International Symposium on Artificial Intelligence, Robotics, and Automation in Space (i-SAIRAS 2003). Nara, Japan.

D. Dvorak, R. Rasmussen, and T. Starbird. 2002. "State Knowledge Representation in the Mission Data System." In Proceedings of the 2002 IEEE Aerospace Conference, Big Sky, MT.

R. Knight, S. Chien, G. Rabideau. 2001. "Extending the Representational Power of Model-based Systems using Generalized Timelines," In Proceedings of the 6th International Symposium of Artificial Intelligence, Robotics, and Automation in Space (i-SAIRAS 2001), Montreal, Canada.

N. Muscettola, P. Nayak, B. Pell, B. Williams. 1998. "Remote Agent: To Boldly Go Where No AI System Has Gone Before," *Artificial Intelligence*. 103(1-2):5-47.

D. Musliner, E. Durfee, K. Shin. 1993. "CIRCA: A Cooperative Intelligent Real-time Control Architecture," *IEEE Transactions on Systems, Man and Cybernetics*, 23(6):1561-1574.

A. Rao, M. Georgeff. 1995. "BDI Agents: From Theory to Practice," In Proceedings of ICMAS-95.

M. Schoppers. 1995. "The Use of Dynamics in an Intelligent Controller for a Space Faring Rescue Robot," Artificial Intelligence 73:175-230.

R. Volpe, I. Nesnas, T. Estlin, D. Mutz, R. Petras, H. Das. 2001. "The CLARAty Architecture for Robotic Autonomy," In Proceedings of the 2001 IEEE Aerospace Conference, Big Sky, MT.