

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
firstname.lastname@jpl.nasa.gov

Abstract

Not only has the number of launched spacecraft per year exploded over the past few years, but also spacecraft are getting progressively more complex as flyby missions give way to remote orbiters, which in turn give way 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 flight software engineering 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 explicitly managing spacecraft state.

Introduction

As *in situ* missions become more prevalent, the standard approach to commanding spacecraft with predictable time-tagged command sequences falls apart due to the dynamic partially understood environments that *in situ* explorers must respond to. For instance, a rover on Mars never knows exactly how long it will take to traverse to a target or how much energy will be required. 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 instance, 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 an *in situ* explorer with predictable command sequences, MDS will control it with goals that capture an operator's intent and drive the explorer 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 systems like 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, the Mission Planning & Execution (MPE) subsystem of MDS manipulates constraints on state variables instead of sequences of commands. Where other planning, scheduling, and execution architectures focus on representing actions one way for planning/scheduling and another way for low-level 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.

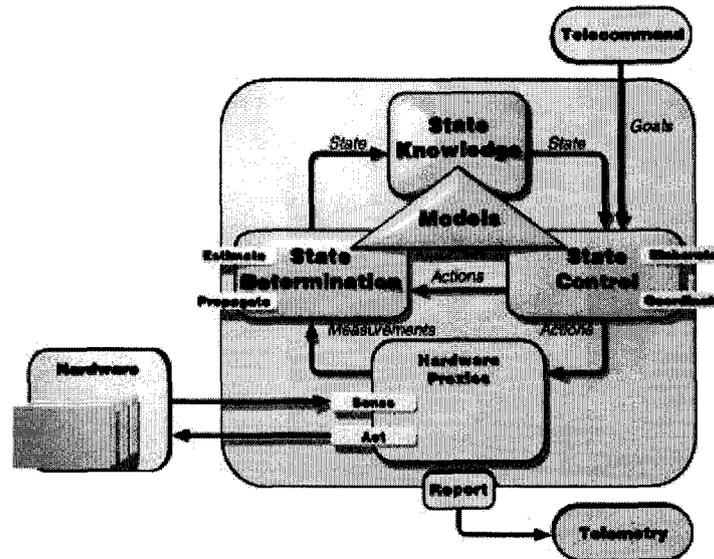


Figure 1. Diagram of the MDS architecture emphasizing the central role of explicitly represented state knowledge and models, goal directed operation, and the separation of state determination from control in all feedback loops.

To describe the MDS's MPE, this paper starts by describing the state and constraint approach to representing plans and problems. With this representation, the subsequent two sections describe the MPE component architecture and how these components combine to implement a fairly simple approach to planning, scheduling, and execution. Finally, the last two sections describe related work and conclude with a discussion of future work.

Plan & Problem Representation

At its heart, MDS represents a spacecraft as an interacting set of state variables (Dvorak, Rasmussen, and Starbird 2002), where each variable denotes an aspect of the spacecraft, its environment, or its supporting ground system that has some measurable impact on mission objectives. 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 healthy or stuck off. Since the transmitter only works when warm, the temperature variable affects it. Similarly, a healthy 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 a sequential pair of goals for controlling the temperature state variable to either warm or cool 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 are used to control the separation between timepoints. In this case, the first goal can last from 100 to 900 seconds, and the second must last for ten minutes.

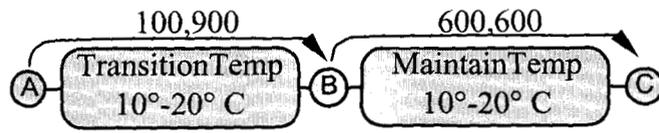


Figure 3. Example goal network with goals on the temperature state variable and min,max time separations between timepoints (in seconds).

While controlling 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 ok (b). Thus a constraint to transmit data for ten minutes captures an operator's intent, and elaboration automatically fills in the supporting constraints.

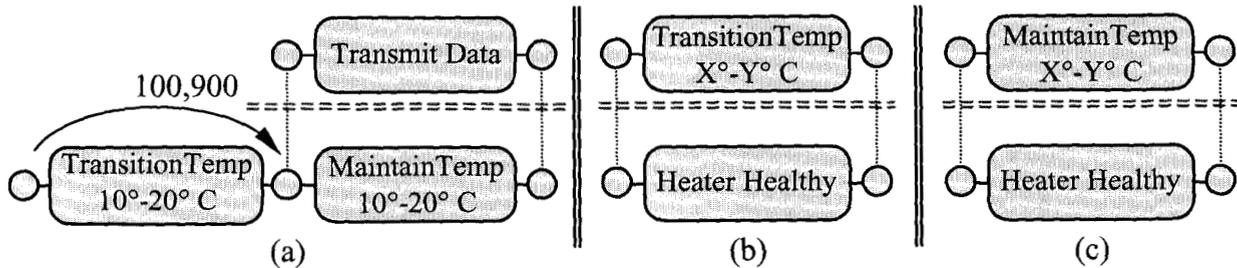


Figure 4. Elaboration tactics supporting a transmit data constraint on a transmitter state variable

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 a enforcing a target goal. The differences between the two approaches include the fact that supporting goals are often enforced simultaneously with the desired goal. Also, this constraint-centered approach facilitates merging different, but compatible, goals that overlap on the same state variable – a feature that other systems currently do not support. 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.

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 between 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.

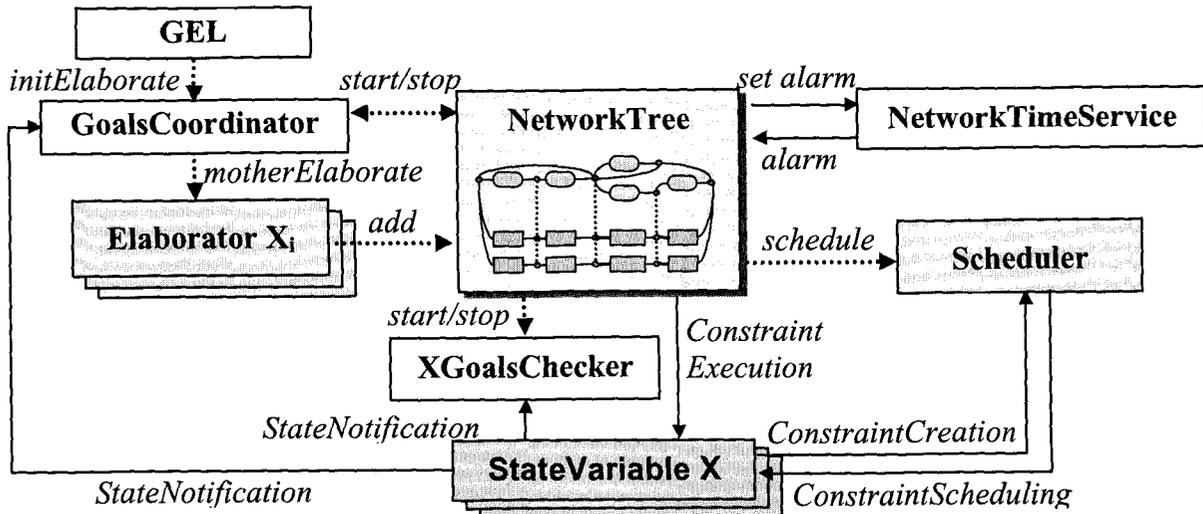


Figure 5. Mission planning and execution system's component architecture

As illustrated, a goal elaboration language (GEL) interpreter inserts a temporally constrained network of commanded goals. These goals are passed to the GoalsCoordinator using an *initElaborate* message, and this component starts elaborators for commanded goals. These elaborators both insert goals into the network tree and start other elaborators for supporting goals. Once a set of goals is 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 the goal's execution in order to facilitate changing a goal's elaboration if execution problems pop up.

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 replacing the executing network with the newly changed task network. While the first and last steps are straightforward copies, the middle tasks to elaborate and schedule define 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 implementing a number of different types of planners/schedulers due to its focus on components. By replacing components, 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 task network to proposed network for modification
while there are goals to elaborate do
  Choose a goal G@[S,E] to elaborate (from a heuristic ordering)
  Add goal to proposed network
  Exhaustively choose elaboration tactic M for G
  Apply M, which possibly generates new goals to elaborate
  - Backtrack (and remove goals) if application of M is illegal
For each state variable SVAR (from a heuristic ordering) do
  For each new goal G@[S,E] on SVAR (from a heuristic ordering) 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 SVAR's timeline - Backtrack if merge illegal
  Propagate the expected behavior of SVAR given the merged goals
  - Backtrack if the propagation is illegal
Replace executing task network with proposed one if it scheduled
  
```

Figure 6. High-level elaboration and scheduling algorithm description

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 proposal request to the network tree and waiting for a response. At this point the motherElaborator task manages the commanded goal subnet's elaboration by creating an elaborator for each commanded 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 subsequently command the elaboration of supporting goals. While not illustrated due to space reasons, each elaborator collects "OK" messages from its children and subsequently signal "OK" to its parent just like the illustrated "OK" messages of elaborator2 and elaborator3.

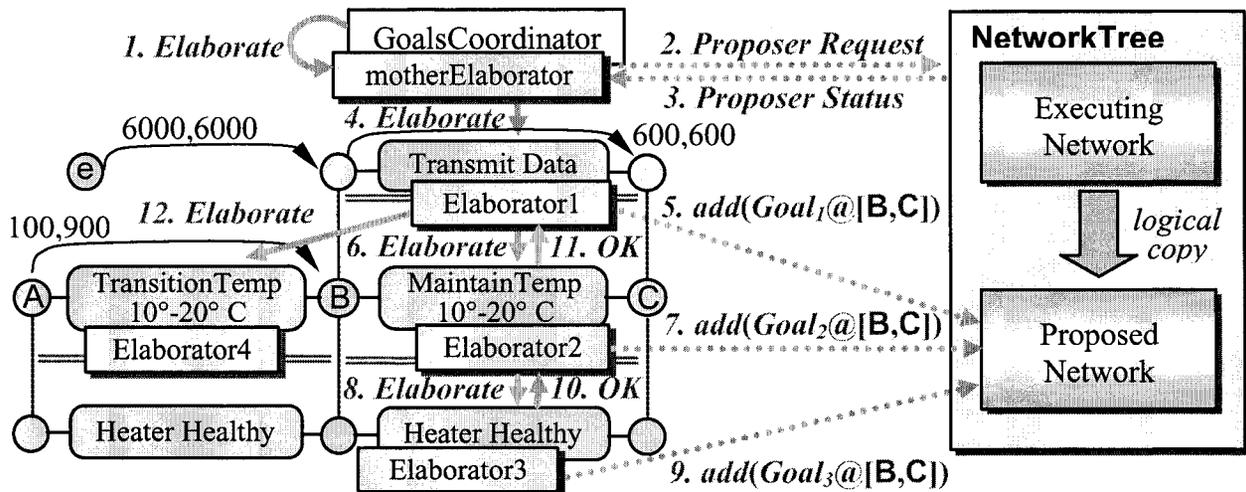


Figure 7. The first twelve out of twenty inter-component messages as components coordinate to elaborate a 600-second transmit data request 6000 seconds after a reference time e , where dashed arrows denote messages relayed through the GoalsCoordinator.

Ultimately the mother elaborator informs the goals coordinator of success or failure depending on whether or not the mother elaborator can get all of its child elaborators to signal "OK". The goals coordinator either requests to promote or cancel the local copy respectively. Even after sending a promote request, all 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.

Scheduling and Promotion

While elaboration was spread over a number of 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 suggests, the scheduler 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's timeline upon entering the inner FOR loop and after each iterate, where a state variable's *timeline* is a string of executable goals (XGoals). In this case the initial timeline contains a single "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.

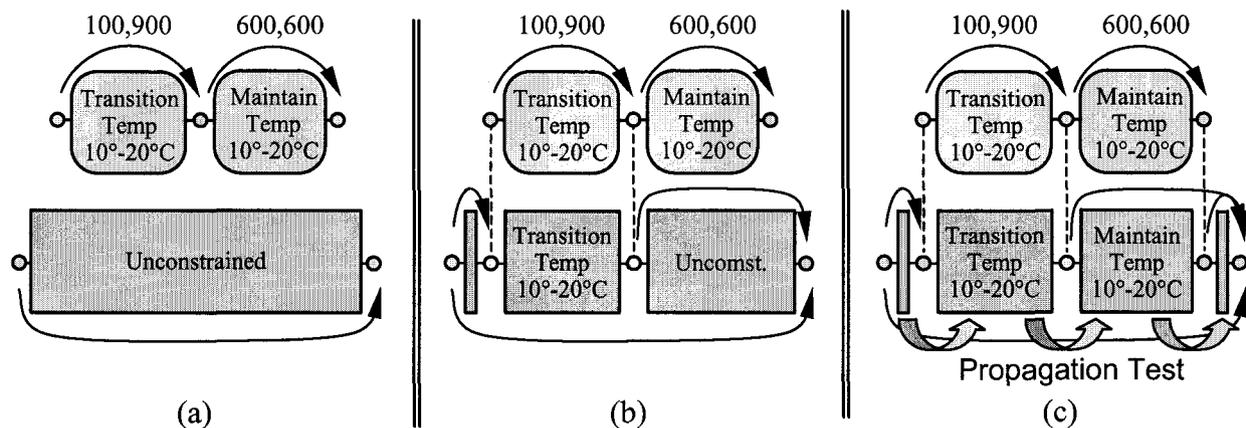


Figure 8. Scheduling example goals on the temperature state variable, where (a) denotes the status prior to scheduling, (b) denotes the status after merging in the first goal, and (c) denotes the status during the propagation test.

In the example, the propagation test passes because the unconstrained to transition-temp to maintain-temp transitions are 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 unconstrained XGoal with an XGoal to maintain the temperature between 0° and 50° Celsius still merges with the example goals, but a 0° to 5° range causes conflicts. Finally, elaboration and scheduling is computation intensive and is performed on a copy of the executing task network. That copy will ultimately replace the executing task network once scheduling successfully completes or get discarded if scheduling could not complete.

Plan Execution

As previously mentioned, plan execution in MDS involves incrementally changing the constraints imposed on state variables and the NetworkTimeService and XgoalsChecker components from Figure 5 combine to facilitate incrementally evolving the commanded constraints. 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 and passes its subsequent XGoals to state variable controllers. 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)
    For each XGoal G@[TP,*] do
      send "start[G]" to G's state variable's controller & estimator
      Start monitoring G's constituent goals
    For each XGoal G@[*,TP] do
      Stop monitoring G's constituent goals
  
```

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 there is a set of zero or more timepoints that 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 constituent goals are monitored. These monitored goals are used to both detect and respond to failure.

Constrained Timepoint Firing

To get a better understanding of the execution algorithm and its component implementation, consider Figure 10 with XGoals resulting from the running 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 temporal 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.

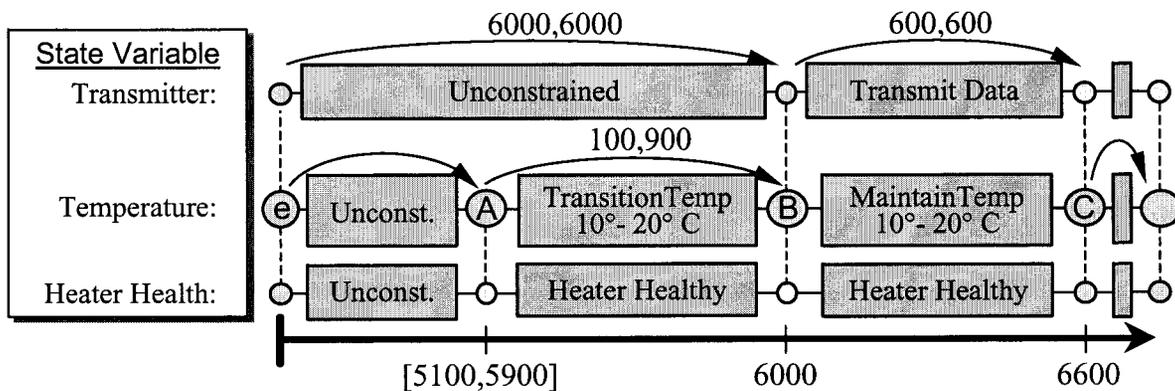


Figure 10. XGoals in support of transmitting data for 10 minutes, starting 100 minutes after a reference "epoch" time e.

To simplify the algorithm trace, we first ignore constituent 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 temporally constrained into the future. Also, the XgoalsChecker component is interrupt event driven by state notifications. In this way the timepoint firing algorithm sleeps while waiting.

For instance, the NetworkTree starts executing the task network by setting up an alarm for 5100, when the first timepoint becomes applicable. Until then all 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 timepoint A to the XgoalsChecker. Since one XGoal is a transition and the other is a health requirement, they are both immediately ready to start. Thus A fires, and the NetworkTree passes the transition and health XGoals to the temperature and heater health state variable respectively for relay to their controllers and estimators. In addition to enforcing new constraints, firing A uncovered B, which results in setting a second alarm for 6000. Now reasoning about timepoints ceases while since none are pending while the transmitter warms.

When the alarm at 6000 goes off, the NetworkTree similarly triggers and detects a timeout. A timeout immediately fires B, and the NetworkTree both passes the executable constraints to the state variables 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. 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

Forced Timepoint Firing and Failure Response

While the previous section discussed the mechanics of timepoint firing and passing constraints down to a state variable's estimator and controller, it explicitly avoided discussion about constituent goal monitoring. In general, an XGoal is a merge of zero or more constituent goals. Whenever an XGoal's constraint is passed down, that XGoal's constituents are monitored against state variable notifications to both detect and respond to failure. Also, constituent goal monitoring stops when the XGoal's ending timepoint fires, resulting in enforcing a subsequent XGoal's constraints.

For instance, when the temperature transition XGoal started at 5100 in the previous trace, a function in the temperature transition goal started receiving state notifications from the GoalsCoordinator. This function's purpose is to respond to situations where the temperature is not transitioning to the target range. There are many forms of responses depending on the function associated with the constituent goal. These can vary from either re-scheduling or re-elaborating the goal to simply failing the goal and extracting it with all goals that either it supports or support it. In the case of our temperature transition, the simplest response to discovering a problem is to fail and eject the entire transmit data goal with supporting goals. Another, more desirable, response is to re-elaborate the transmit goal to use a lower bandwidth transmitter for reporting the problem.

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 transmitter's temperature is only 5° C, and the problem is resolved by the constituent MaintainTemp 10°-20° C goal's monitor function.

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. 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 single objective rescue robot, 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. Finally, CLARAty (Volpe et al. 2001) and MDS both provide general robotic control environments with a current focus on rovers. While the CLARAty environment facilitates the integration of research algorithms, MDS focuses on flight software.

First Year	System	Layer R-T Guarantee			Applications
		Plan	Exec.	Driver	
1993	CIRCA	Hard	Hard	Hard	PUMA robot arm
1995	EVAR	Hard	Hard	Hard	EVA astronaut retrieval robot Simulation
1998	Remote Agent	None	Soft	Hard	DS-1 probe (experiment)
2001	CLARAty	Soft	Soft	Hard	Research on Rocky7&8 rovers
2003	ASE	Soft	Hard	Hard	EO-1 Satellite (experiment)
2004	MDS	Soft	Hard	Hard	Rocky7 & 8 rovers (targeting MSL)

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

While MDS will have 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 state estimators and state controllers. Two, the focus on reasoning about state constraints instead of actions facilitates a clean mechanism for merging 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 state variable controllers in what is called a delegation pattern, but that is beyond the scope of this paper.

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 state variable is associated with a single estimator and controller

within an execution layer. A plan in the MPE 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 both satisfy science observation and health maintenance objectives.

While the MDS architecture currently drives a rover around the JPL Marsyard, a number of improvements are planned to facilitate scaling the MPE up to control the Mars Science Lab when it arrives on Mars in 2010. The first involves modeling how a state variable's constraints affect other variables, which is similar to checking side effects on action centered AI planners. This starts by extending Figure 6's propagation step to check state variables that affect the variable currently being scheduled. Other extensions involve performance improvements in Figure 6's algorithm.

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.

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. May 2003
- 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.