

# The MDS Autonomous Control Architecture

Erann Gat

*Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA*

## ABSTRACT

We describe the autonomous control architecture for the JPL Mission Data System (MDS). MDS is a comprehensive new software infrastructure for supporting unmanned space exploration. The autonomous control architecture is one component of MDS designed to enable autonomous operations.

**KEYWORDS:** autonomous control, unmanned space exploration

## INTRODUCTION

Mission operations represent a large fraction of the total cost of unmanned space missions. One way to reduce these costs is to automate parts of the spacecraft control process. Earlier this year the feasibility of this approach was demonstrated by the Remote Agent Experiment (RAX) [1] running aboard the New Millennium Deep Space 1 (DS1) spacecraft. JPL is currently developing a second-generation autonomous control system for spacecraft as part of the Mission Data System (MDS) project [2].

The MDS autonomous control architecture is based on the concept of a *goal*, which is defined as a constraint on a state over a temporal interval. Goals are an improvement over traditional methods of commanding for several reasons. First, they explicitly represent intentions, which makes it possible for the spacecraft to respond more robustly to faults. Second, they allow decisions to be made locally aboard the spacecraft using information that might not be available on the ground. Third, they allow temporal flexibility and event-based actions.

To understand goals and their resulting advantages it is useful to begin with a review of current spacecraft commanding methods and their shortcomings.

## SEQUENCING

Spacecraft are currently controlled by command sequences, which are time-tagged lists of commands. A command is essentially a black-box subroutine call. They range from very simple (e.g. turn a device on or change its mode) to very complex (e.g. change the spacecraft attitude). By stringing commands together in sequences the spacecraft can be made to perform its mission functions.

The initiation of a command is almost always triggered by time. (In the few cases where this rule has been violated it has required very expensive customized single-use flight

software.) The use of time-tagged command sequences is motivated by the desire to be able to accurately predict exactly what the spacecraft is going to do. Historically this approach has worked well, but it has costs and limitations that are now becoming unacceptable.

The cost of command sequences arises from the fact that they do not just *enable* prediction of spacecraft behavior, they *require* it. Executing a command will have different effects depending on the circumstances under which the command is executed. A given command could, under different circumstances have a desirable effect, a disastrous effect, or no effect at all. As a result it is absolutely crucial that the state of the spacecraft be accurately predicted. This in turn requires very precise modeling of spacecraft behavior, which is expensive.

Further complicating the matter is the fact that spacecraft behavior is in some aspects inherently non-deterministic. The onset of faults, for example, cannot be predicted. Even in nominal operations there are non-deterministic aspects of spacecraft behavior. For example, the amount of time it takes to complete a turn to a new attitude cannot be precisely predicted.

The historical approach to these problems has been to 1) make worst-case assumptions about things like resource usage and settling times and 2) to take a very conservative approach to fault response. The general approach is that any deviation from predicted behavior causes the spacecraft to enter a safe mode in which the all systems are shut down except the bare minimum needed to communicate with Earth. The spacecraft then waits for instructions from ground controllers. Unfortunately, there are a few situations where this approach would cause loss of mission, e.g. during an orbit insertion. In such situations the usual fault responses are disabled, and fault protection is done by a special sequence customized for the particular circumstance called a critical sequence. But the sequencing ontology is not designed to handle faults. (This limitation is deliberate, since the whole point of sequences is to force spacecraft behavior to be deterministic, but fault handling is inherently non-deterministic). As a result, generating critical sequences is extraordinarily expensive. The cost of generating critical sequences can dominate an operations budget despite the fact that only a tiny fraction of the mission actually requires them.

Even during non-critical operations traditional sequencing extracts a significant cost. Ground intervention is required every time the spacecraft enters safe mode, which is to say, any time anything unexpected happens. This costs time and money, and can result in the loss of significant science data. Finally, there are new classes of missions for which the sequencing approach breaks down completely because the spacecraft must interact with an unpredictable environment, e.g. comet landers, rovers, and the Mars Airplane.

## **GOALS**

MDS is developing a fundamentally different approach to spacecraft commanding designed to address the shortcomings of the sequencing approach. Instead of commands, spacecraft operations in MDS are based on the concept of a *goal*, which is defined as a

constraint on a state over a temporal interval. A state in turn is defined as a property of an object or a relationship between objects. Examples of states are: “The attitude of the spacecraft,” “The power state of the camera,” and “The distance between the spacecraft and Earth.” An example of a constraint is, “Constrain the power state of the camera to be ON.” Constraints become goals when associated with a temporal interval, e.g. “Constrain the power state of the camera to be ON between time T1 and time T2.” Notice that goals describe the intended results of actions, not the actions themselves.

The times that constitute the beginning and end of a goal’s temporal interval are not necessarily fixed times. Instead they can be flexible time points constrained by relative intervals to other time points. For example, it is possible to say, “Constrain state X to have value Y starting at least N and at most M seconds after time point T1 (associated with some other goal) and ending at least P and at most Q after time point T2.” Such a “floating” time point is called an *ungrounded* time point. A time point fixed in time is said to be *grounded*. A set of goals that are associated with each other by virtue of interrelated time points is called a *goal network* or goal net. Goal nets replace command sequences in MDS.

## CONTROL ARCHITECTURE

The software architecture that implements goal nets is conceptually similar to a traditional closed-loop control system. The architecture is divided into two major components, *state determination* or estimation, and *state control* [3]. The state determination component is responsible for estimating the past, present and anticipated future values of states. State control takes the output of state determination, compares the results to the constraints in the goal net, and performs actions to influence system state to conform to those constraints.

State determination and state control are both further subdivided into modules that are responsible for estimating or controlling subsets of the total system state. On the estimation side these modules are called *estimators*, and on the control side they are called *goal-achieving modules* or GAMs. The MDS control architecture is thus sometimes referred to as the GAM architecture.

Both estimators and controllers employ *models* of the system to perform their function. A model is an object that describes a relationship among states. For example, a model might represent the fact that the power state of a device is ON if the switching state of its power switch is CLOSED. This information can be used by a GAM to figure out that it can achieve goals on a device’s power state by issuing subgoals on the switch state. It can also be used by an estimator to conclude that if the device is operating that its switch is closed.

The interfaces to these components can be specified with some rigor, though we give only an informal sketch here. An estimator takes as input a series of *measurements*, which are the raw data from low-level transactions with the spacecraft hardware, and produces as output a set of estimates of the values of certain states. Because these estimates are not just estimates of the present value but of past and future history as well

we refer to these estimates as *value histories*. (To be precise, a value history is a data object that represents a state estimate over time.)

A GAM takes as input a set of value histories and a set of goals, and produces as output either primitive commands or *subgoals* which are inputs to other GAMs. Thus, the goal structure is recursive, with high-level goals giving rise to lower-level subgoals. The recursion bottoms out in primitive commands sent directly to the hardware or to real-time control loops.

The process of expanding a goal into subgoals and primitive commands is called goal *elaboration*. Goal elaboration can be done manually or automatically. Automatic goal elaboration is the principal mechanism for autonomy in the MDS architecture. Notice that MDS enables autonomy but does not require it. Automatic goal elaboration in MDS is done using AI planning techniques developed elsewhere (e.g. [4]).

The MDS architecture is similar to a traditional hierarchical control architecture (e.g. [5]). The distinctive features of the MDS architecture are 1) the explicit extension of estimates into the past and future, 2) the representation of control “setpoints” as goals extending over flexible temporal intervals (an idea borrowed from the Remote Agent [1]), 3) the subdivision of estimation and control into separate regimes (estimators and GAMs) that interact with each other, and 4) the extension of the paradigm to states with discrete values like power switching states.

## **FAULT PROTECTION**

MDS provides a model of fault protection that is smoothly integrated into the architecture rather than being a separate module as in the traditional sequencing approach. Fault protection is based on the idea of goal failure. A goal is said to *succeed* if the value of the goal’s state variable conforms to the goal’s constraint over the goal’s interval, otherwise the goal *fails*. Goals can fail for many reasons, including hardware faults, unexpected environmental interactions, and conflicts with other goals. MDS makes no architectural distinction among these various causes; all goal failures are created equal.

The effect of a goal failure is limited to the failed goal’s parent goal. If a goal G1 has a subgoal G2 and G2 fails then the GAM responsible for G1 can locally contain the failure by selecting an alternate method of achieving G1. For example, suppose G1 is a goal to make the camera power state be ON, which results in a subgoal G2 to make the state of a particular power switch S1 be CLOSED. If a hardware fault prevents S1 from closing then G2 will fail. But the GAM responsible for G1 might issue a second subgoal, G3, to close a backup switch S2 in stead of S1.

In a traditional sequencing architecture, the failure of S1 would result in the spacecraft going in to safe mode. This would require expensive manual intervention, and could result in the loss of science data or even the entire mission. In MDS the situation is handled locally with no disruption. The need for manual intervention is eliminated in many cases, as is the need to distinguish between critical and non-critical sequences. Only in the case where all local recovery options are exhausted does the failure propagate

up the goal hierarchy. At every stage the spacecraft responds as robustly as its knowledge of the circumstances permit.

Fault states can be modeled and estimated just like any other state; no architectural distinction is made. For example, a switch could be modeled as having two orthogonal states, its switching state (OPEN or CLOSED) and its operational state (OK or STUCK). The model would represent the fact that the switching state can change only if the operational state is OK. This gives the system a lot of flexibility in achieving goals. For example, a stuck switch is not necessarily a useless one. It can still successfully achieve goals as long as the switch happens to already be in the desired state. The system might also have at its disposal a command that can cause a switch to transition from STUCK to OK. (A realistic example would be a solid-state switching unit with a reset function.) In this case the system could automatically determine that it has to change the operational state of the switch before it can change its switching state.

## EXAMPLES

The simplest example of a goal consists of a constraint on a discrete-value variable to assume a specific value over a fixed time interval, e.g. "Make the power state of the camera be ON starting at 1:00 and ending at 2:00." There are several things to notice about this goal. First, it says nothing about the power state of the camera before 1:00 or after 2:00. If it is desired that the camera be on *only* between 1:00 and 2:00 and off otherwise then two additional goals are needed to specify that the camera should be off before 1:00 and after 2:00.

Second, if the camera is off then the command to turn on the camera must be issued before 1:00 for this goal to succeed. The goal criterion is strict: if the camera is off even for a fraction of a second between 1:00 and 2:00 then the goal fails. It is possible to specify more lenient goals by relaxing the goal's constraint. For example, one could specify that the camera should be on for 99% of the time between 1:00 and 2:00. Brief transient power outages would not cause this goal to fail until the outages exceeded 1% of the interval. The only limit on how complex constraints can be is the ability of a GAM to process them. There are no architecturally imposed limits.

Third, because instantaneous state transitions are physically impossible there must be gaps between goals with non-overlapping constraints. In the camera example, it is not possible to make the power state be OFF from 12:00 to 1:00 and ON from 1:00 to 2:00. As specified at least one of those two goals must fail. To allow them both to succeed one could, for example, specify that the camera should be OFF from 12:00 and 12:59. This would give the system a one-minute window of opportunity to switch the camera on. (Actually, the system could potentially switch the camera on and off an arbitrary number of times during this one-minute window. To prevent that one could specify an additional goal whose constraint is that the power state transitions from OFF to ON exactly once at some unspecified time during the one-minute interval. Such a goal is called a transitional goal.)

Goals can be issued on states over which the spacecraft has no direct control. For example, one can formulate a goal on the position of Jupiter. At first it might seem that such goals have no use since the position of Jupiter is not something that MDS has any influence over. But MDS has two tools at its disposal for achieving goals. The first, issuing commands, is only applicable for goals on states that are influenced by commands. The second tool is making choices about how time points are grounded. This second technique can be applied to non-controllable states.

For example, one could issue a goal for Jupiter to be in a particular position starting at an *ungrounded* time point T1. The system could achieve this goal by grounding T1 at a time when Jupiter is in the desired position. The utility of such a goal is that T1 can serve as a trigger for goals on controllable states by constraining the intervals on those goals relative to T1. This is the basis for doing event-driven control in MDS.

## **SUMMARY**

The MDS autonomous control architecture is designed to replace traditional sequencing-based approaches to controlling spacecraft. The MDS architecture is based fundamentally on the idea of a goal, which is a constraint on the value of a state over a (possibly flexible) time interval. Goals provide a unified model of fault protection, and a smooth transition from manual to autonomous operation. The MDS control architecture is part of the larger MDS effort to completely re-engineer the software infrastructure and development process for flying unmanned space missions.

## **ACKNOWLEDGEMENTS**

This paper describes work performed by the entire MDS design team and the MDS control architecture implementation team, including Bob Rasmussen, Tom Starbird, Kim Gostelow, Bob Keller, Winthrop Lombard, Ed Gamble, Weldon Smith and Dan Dvorak.

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

## **REFERENCES**

1. Barney Pell, et al. "An Autonomous Spacecraft Agent Prototype." *Autonomous Robots* 5(1), March 1998.
2. Dan Dvorak and Robert Rasmussen. "Software Architecture Themes in JPL's Mission Data System." *Proceedings of the IEEE Aerospace Conference*, March 2000.
3. Robert F. Stengel. *Optimal Control and Estimation*. New York: Dover, 1986.
4. Steve Chien, et al. "Autonomous Planning and Scheduling for Goal-Based Autonomous Spacecraft." *IEEE Intelligent Systems*, September/October 1998.
5. James Albus. "A Reference Model Architecture for Intelligent Hybrid Control Systems." *Proceedings of the 1996 Triennial World Congress, International Federation of Automatic Control (IFAC)*, San Francisco, CA, July 1996.