Comparison of Goal-Based Operations and Command Sequencing

Daniel L. Dvorak[1] and Arthur V. Amador[2] and Thomas W. Starbird[3]
Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA 91109 USA

In robotic space missions the purpose of any operations paradigm is to achieve specific objectives while protecting the health of the space vehicle(s). Unfortunately, in the dominant paradigm of command sequencing, the representation of such objectives, health constraints and other dependencies is left behind in the ground-based activity planning process and never carried into uplinked products where it can support context-specific status monitoring, resource allocation, fault protection, and general automation on the spacecraft. In contrast, in the paradigm of goal-based operations, operational objectives are ever-present, from the beginning of activity planning all the way to execution on a space vehicle. This paper examines the change in operations perspective from command sequencing to goal-based operations, with particular emphasis on the uplinked product—termed a *goal network*—and the design of *goal elaborators* needed to generate it.

Nomenclature

ACS = Attitude Control System
GBO = Goal-based operation
MDS = Mission Data System

I. Introduction

Robotic space vehicles have long been operated via a paradigm known as command sequencing whereby time-based command sequences are prepared and validated on the ground and then transmitted to a space vehicle for execution by a sequencer. This approach has worked well for space vehicles whose internal state and environmental conditions can be predicted with reasonable accuracy, and where safe-mode responses are adequate for most faults. However, this approach has been increasingly strained to accommodate more complex missions where vehicles are expected to operate autonomously in uncertain environments and in the presence of long communication delays with Earth. Flight system capabilities such as vehicle mobility with hazard avoidance, opportunistic science observations, and autonomous fault diagnosis and response all make the state of the vehicle less predictable, and thus more problematic for command sequencing.

An emerging paradigm, termed *goal-based operation*[1] (GBO), addresses these challenges by changing the basis of operations from imperative programming to a more declarative style of programming. A command sequence is an imperative program designed to change the state of a system in some desired way, though the desired state is typically not represented explicitly in the program, and determination of success or failure is typically done outside the program. In contrast, a *goal network* is a declarative specification of desired states, termed *goals*, which are linked together through dependencies and temporal constraints.

Fundamentally, goals are directives for closed-loop control, whether for short or long timescales and whether for ground or flight or both. In some respects, goal-based operation simplifies the job of operators because it focuses attention on specifications of *what* to do rather than *how* to do it. In other respects, though, goal-based operation gives operators new responsibilities to design and refine the how-to-do-it part, particularly for goals that must execute onboard, without opportunity for human-in-the-loop inspection and approval. Importantly, the how-to-do-it part can include context-sensitive fault responses, enabling a vehicle to recover autonomously or at least to degrade

[1] Principal Engineer, Planning & Execution Systems, M/S 301-270, AIAA Member.
[2] Section Manager, Planning & Execution Systems, M/S 301-250D, AIAA Member.
[3] Principal Engineer, Planning & Execution Systems, M/S 301-250D.
[4] SpaceOps 2008 paper #110966, version 3, updated April 5, 2008.

gracefully. From an operations perspective the real challenge is in how to move the how-to-do-it part from the ground environment to the flight environment in a way that is practical for both nominal and critical operations.

This paper describes key characteristics of goal-based operation and command sequencing, and compares the process of operations in the two paradigms. The paper summarizes a particular architecture for goal-based control, describes associated design tasks for operations engineering, and also addresses operational approaches that would be useful in managing the transfer of the responsibility for getting from intent to action from the ground to the flight environment.

II. Origins of Command Sequencing and Pressures for Change

Command sequencing became the basic operations paradigm in the early days of space exploration at a time when missions were relatively close to Earth and flight processors were relatively slow and memories were relatively limited compared to their modern counterparts. Spacecraft were commanded using linear sequences of commands, with very simple execution semantics. A command typically consists of an execution time, a command code and associated parameters, and each command is dispatched by an onboard sequencer when its execution time occurs. In the typical case, command execution is "open loop"; there is no feedback about whether the command worked and whether it achieved whatever was intended. In order for such command sequences to work as intended, it is essential to accurately predict the state of the spacecraft and its environment at the time of execution of each command. For many classes of missions, such predictability *has* been possible and safe-mode responses *have* been acceptable for handling the unpredictable, such as failures and unexpected environmental conditions.

Although modern flight processors are vastly more powerful than their early predecessors, and although missions are operating far from Earth in environments that are not as well modeled, command sequencing has remained as the dominant paradigm for operations, albeit with various extensions to accommodate less predictability and to react autonomously to a wider range of conditions. One type of extension is the introduction of multiple concurrent sequences. This permits concurrent execution of relatively independent activities but requires some form of synchronization where dependencies exist, usually done in an *ad hoc* manner. Another type of extension is a mark-and-rollback mechanism in the sequencer, often used in "critical sequences" where there is not enough time to consult an operator when things go wrong. However, developing and validating a critical sequence can cost orders of magnitude more than a non-critical sequence. Another area of extension has been in fault protection where, instead of simply detecting a problem and transitioning to safe-mode, it is designed to recover spacecraft capabilities as well as possible. However, in this approach the interplay between nominal sequencing and fault protection gets very complicated and therefore difficult to design and validate. Yet another type of extension is a "programmed behavior" in which a command can initiate a complex closed-loop procedure that reacts to local conditions in attempting to achieve some objective, and may take a variable length of time to complete. This approach entails more cost in flight software development, is more costly to modify than a command sequence, and complicates operations planning somewhat due to the variable execution time.

These extensions to the command sequencing paradigm point to three needs that have gained more importance since the early days of space exploration. First, there is a need to make more effective use of expensive spacecraft assets. This means more onboard autonomy to achieve mission objectives and react appropriately to local conditions, thereby reducing the number of situations that require costly, time-consuming intervention by the ground. Second, there is a need for onboard fault protection that works harmoniously with nominal activities. Fault responses must be more localized and context-specific so that they do not disrupt unaffected activities, as would happen with safe-mode responses. Third, there is continual pressure to reduce engineering costs in spite of increasingly complex requirements. That pressure—coupled with observations about complexity in design, verification and operations—motivates the search for architectural concepts that can better manage complexity.

Our intention in this paper is to show that goal-based operation—and its architectural concepts and mechanisms—addresses these needs, and to illustrate how it affects operations engineering. The next section shows that the very concept of goal-based control can be derived from a careful examination of a command sequence. In that sense, goal-based operation can be viewed as an evolution rather than a revolution.

III. Viewing Command Sequences as Goal Networks

The design of a spacecraft control system—whether via command sequences or goal networks—requires the same kinds of knowledge. Specifically, it requires an analysis of the system under control in terms of its physical states to be controlled and models of behavior. That knowledge then informs the design of the control system. The example below examines a simple physical system and an easy-to-understand command sequence for controlling it. As we delve into the knowledge and thinking that went into the design of the command sequence, and start to

represent that knowledge in a more explicit form, we will begin to see a command sequence as a degenerate form of a goal network. Subsequent sections will illustrate the value of retaining this knowledge in the uplinked product (termed *goal network*) to facilitate verification and enable autonomous responses to unexpected situations.

Consider the simple case of a camera on a scan platform, as shown in Figure 1. The camera rotates on a gimbaled platform for pointing to a target, and picture data from the camera is stored separately in a data recorder. When the camera is powered off, a heater is used to keep it acceptably warm, but when the camera is powered on, the heater is to be turned off to avoid overheating. Figure 2 shows a command sequence for taking a picture. At the specified starting time of 1:00 PM the sequence issues the first command to turn the heater off, followed shortly by another command to power on the camera. Still later, another command is issued to begin turning the scan platform to a target. That turning motion takes a variable amount of time, and completion of the turn is signaled to the command sequencer via a semaphore. The camera is then commanded to take a picture, followed by clean-up commands to turn off the camera and turn on the heater.

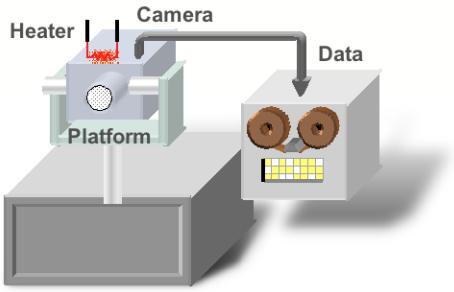

Figure 1. A simple system for comparing command sequencing and goal-based operation.

Time	*Command*
1:00 PM	Camera Heater Off
+2 m	Camera On
+8 m	Turn platform to target
Turn done	Take picture
+1 m	Camera Off
+2 m	Camera Heater On

Figure 2. Sample command sequence.

Although it's not explicit in the sequence, engineers understand that certain states of the system under control are being controlled through the effects of commands. For example, the "heater on/off" commands affect the state of the camera heater and its consequent heating effect. Similarly, the "camera on/off" commands affect the power state of the camera electronics, the "turn" command affects the pointing of the scan platform, and the "take picture" command affects the camera operating mode. Figure 3 illustrates the command sequence in terms of these state variables and makes it easy to see command pairs such as "camera power on" followed later by "camera power off". Figure 3 also makes it easy to see that some things are underspecified and other things are overspecified. For example, the sequence is *underspecified* in that it does not say what the intended state of the camera power is in the period between "camera power on" and "camera power off". Of course, everyone *expects* it to be powered on, but an expectation cannot be represented in a command sequence and is therefore not present to support planning-time verification or execution-time monitoring. The sequence is also *overspecified* in the sense that other timings or orderings would also work. For example, the command to turn the platform could have been the first or second command in the sequence rather than the third. The choice was arbitrary, forced by the fact that all commands in a sequence must be ordered along a single command timeline. This limitation can be problematic in large systems because operators may be afraid to alter command timing and/or ordering if the real constraints are not readily available.

In order to verify this command sequence—or any command sequence—it is necessary to "fill in the gaps" between what a sequence *says* and what it *intends,* and also predict system *behavior* given the commands to be issued. Figure 4 again depicts the command sequence shown in Figure 3, but now fills in some details. It assumes start states, imposes end states, and uses models of behavior to fill in state gaps (such as the prediction that the heater remains off after commanded off, until commanded on), and includes side effects such as that the camera

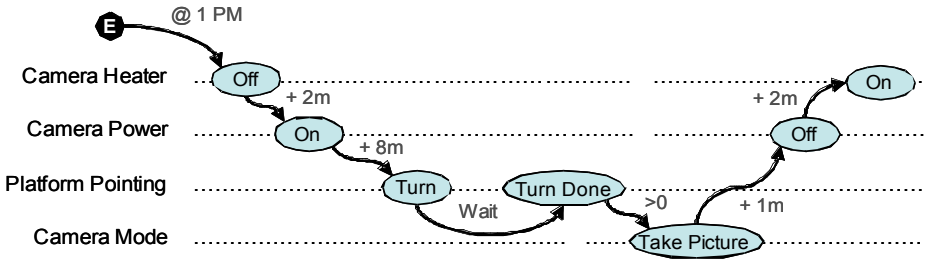

Figure 3. Command sequence shown as instantaneous effects on four state variables.

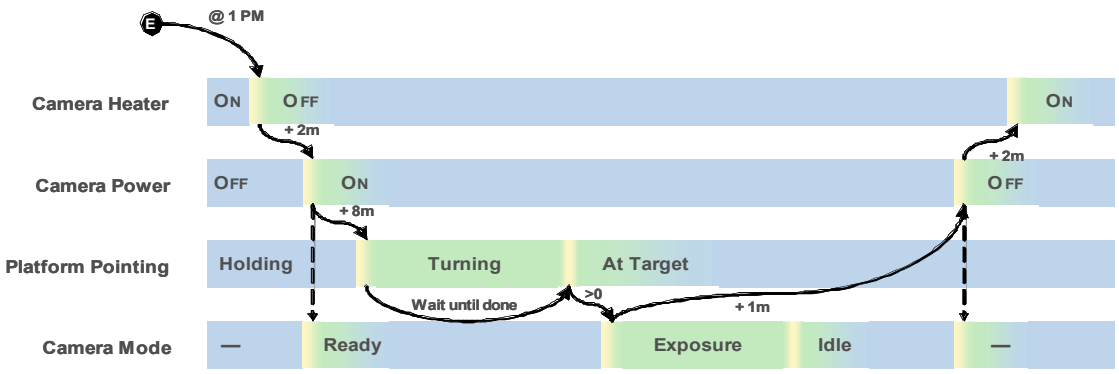

Figure 4. This state representation supports sequence checking by assuming start states, imposing end states, filling in state gaps between events, and depicting side effects.

powers up in "ready" mode. With this extra information, the sequence can now be checked for compliance with rules, such as checking that the platform remains stationary during camera exposure and that enough time is allowed for exposure before turning off the camera.

The next step in the evolution from command sequence to goal network is shown in Figure 5. Here, the *intention* of the original sequence—but not the commands themselves—is represented as state constraints over time intervals on each of the four state timelines. Those time intervals are demarked by time points, and the timings in the original sequence are represented as temporal constraints between time points. For example, the 2-minute delay between "camera heater off" and "camera power on" is represented as a 2-minute time constraint from time point 1 to time point 2. A time point may exist on more than one timeline to indicate simultaneous change, as shown for time point "8" where turning the camera off has the simultaneous effect of making camera mode undefined.

This representation in Figure 5, termed a goal network, is noteworthy for several reasons. First, it focuses operational attention on *what* to do rather than *how* to do it. In the simple example discussed thus far, there's little difference between the two, but in more complicated examples having multiple levels of goal elaboration (to be discussed in Section V), there is considerable value in the distinction. Second, the focus on states of the system under control is fundamental because the purpose of operation is to change the states of the system in coordinated and safe ways. Coverage of system states is broad, including dynamics, device status, resources, data repositories, parameters, and environment. Third, state constraints are remarkably expressive in that they can represent not only control targets (such as a temperature range) but also device modes, resource allocations and required resource

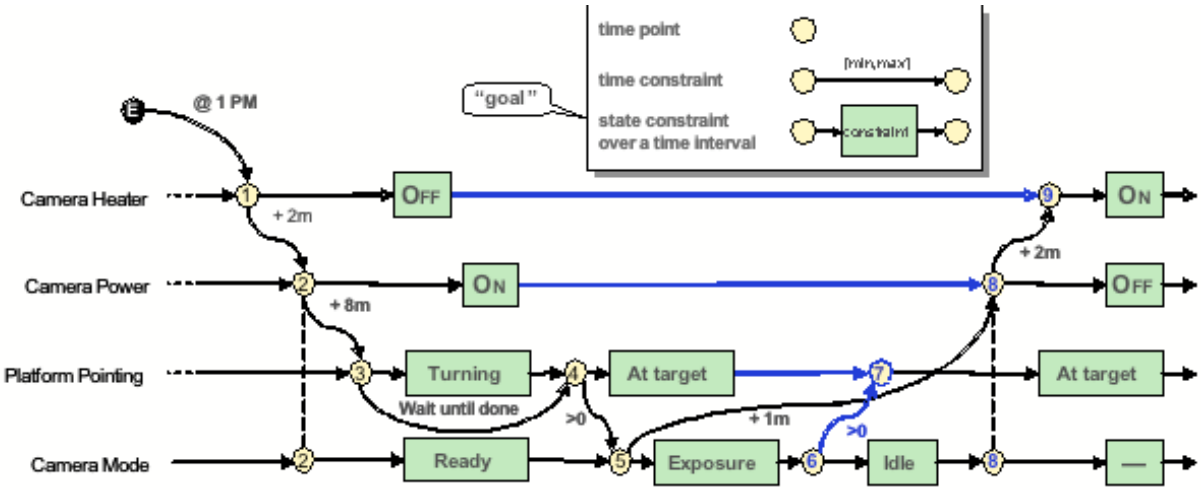

Figure 5. The original command sequence is now represented in a constraint network containing state constraints for required or expected state values, and time constraints for the start, end, and duration of those state constraints. Time points mark the start and end of state constraints, and can be shared across timelines to represent simultaneous effect.

margins, necessary preconditions, and even required quality of state knowledge. This uniformity in representation enables verification of multiple concerns as constraint checking on a single structure (the goal network). Fourth, the existence of state constraints in the uplinked product enables execution-time monitoring of estimated state versus desired state. That, coupled with onboard goal elaborators, enables onboard fault protection that uses the same mechanisms as nominal control, thereby simplifying the flight software architecture. Fifth, the use of explicit time constraints in a goal network eliminates the necessity (in command sequences) of committing to an arbitrary ordering of commands. Although the example shown thus far has inflexible time constraints, this is not a restriction; a time constraint can specify a range, permitting flexibility in the timing of goal start, goal end, and goal duration. Sixth, the explicit representation of intent on state values facilitates verification in a way that command sequences cannot. In a sequence-driven system, if one sequence turns on a device "intending" to keep it on for an hour, but another sequence turns it off a minute later, nothing has been violated in the semantics of command sequences. In a goal network, however, any attempt to levy conflicting goals on the same state variable is immediately detectable as an unsatisfiable constraint. Overall, the greater expressive power of goal networks addresses many limitations of command sequencing.

IV. Architectural Concepts of Goal-Based Control

In the paradigm of command sequencing, a sequence is prepared by operators and uplinked, executed onboard at the prescribed time, and telemetry is down-linked to operators for analysis, where control loops are ultimately closed. The comparatively simple semantics of a command sequence permits a relatively simple onboard execution system, commonly called a *sequence engine*, which issues commands to lower-level software based on time or events. However, the richer semantics of goal networks calls for more architectural support. This section overviews one architecture specifically designed for goal-based control, namely, the Mission Data System (MDS). This section describes MDS in terms of architectural concepts, with attention to goal representation and processing mechanisms, but leaves many details to other papers[6,7].

A. Goals as the Basis for Operation

The distinction between a command and a goal marks one of the greatest departures of the MDS architecture from conventional approaches. A command is momentary, and any lasting effect it may have is due to functions or behaviors expressed elsewhere in the system. Thus, one cannot tell from a command alone whether it contradicts the intent of its predecessors or interferes with the intent of its successors. In systems that run multiple concurrent sequences, one sequence might power on a device with the intention of it being powered on until its corresponding "power off" command is issued, but another sequence might power off that device for its own reasons, unaware of its interference with the intent of the first sequence. The absence of intent not only complicates verification of command sequences but also validation of results. For example, if a device is intended to be on for some period of time, but spontaneously turns off, there is no automatic detection and reporting of that unintended event because there *is* no representation of intent. For the same reason, the control system cannot intelligently react to such an event because it doesn't know what was intended over what time interval.

In contrast to a command, which is inherently momentary, a goal explicitly represents intended state over a time interval. At planning time, concurrent goals on the same state variable are allowed only if they are mutually compatible (e.g., two goals that both want the same device powered on). At execution time, violation of intent is trivially detected by comparing estimated state to intended state and reported to an operator. Furthermore, goal violations can trigger appropriate onboard fault responses because the intended state is known. A goal-driven control system has a contract to achieve (or maintain) some condition, or report that it has failed to do so. This is the architectural concept of *cognizant failure*[8].

B. Goal as a Specification of Intent

A goal represents intent, but that begs the question of *how* to represent intent. Since goals are used to operate a system, they must have clear semantics that can be processed by a computer. By itself, this requirement would allow a kind of goal such as "execute procedure *X* with parameters *a*, *b*, and *c*, and return success or failure", but such goals are specific to the design of a control system, with no broader semantics (see § IX.D). With goal-based operation we want to also use goals for interoperation and coordination of a system of systems, with elements potentially built by different teams, so goal semantics should be independent of control system design. Specifications of intent should come naturally from the problem domain and should have obvious meaning to systems engineers and operators. In this vein, "intent" must be about the *system under control*, not the control system. For example, a goal may specify a desired spacecraft attitude since that is a constraint on a state of the

system under control, but a goal must not specify a mode of an attitude controller since that is an implementation-specific state of a controller in a control system.

The MDS control architecture defines a goal as a constraint on the value history of a state variable during a time interval, as depicted in Figure 6. The term "state variable" refers to an element of the control system that corresponds to a physical state of the system under control. For example, a power switch in the system under control will physically be in an opened state or closed state or tripped state, and the control system will use evidence such as sensor measurements to estimate the state of the switch as opened, closed, or tripped. A goal on the state of the switch is specified as a constraint on the switch's estimated state. For example, a goal could specify that the value of the state variable must be 'closed' continuously from 1:00 PM to 2:00 PM. A different goal could require that the switch be closed 90% of the time between 1:00 PM and 2:00 PM, thus tolerating temporary switch transitions. In all cases, a goal specifies a requirement on a state history that must be satisfied. If the goal's constraint is violated, the control system detects the violation and handles it in ways to be described later.

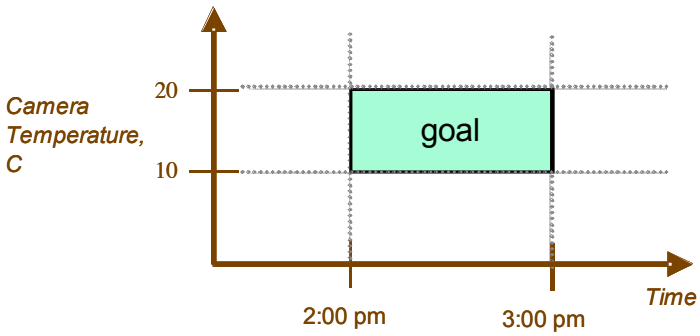

Figure 6. A goal is a constraint on the value history of a state variable over a time interval, and can be visualized as a region of acceptable behavior in value and time, as depicted by the box.

Goals can also be specified on uncontrollable states. For example, the light level from the Sun is not controllable, but activities that depend on some minimum light level—such as picture-taking—can include a goal on light level. Such a goal will not be ready to start until the estimated light level satisfies the goal's constraint. When such a goal is coupled appropriately with dependent goals, then the starting or ending time of whole activities can be based on such conditions. In this sense goals represent requirements, and forward progress in execution can be conditioned on events.

C. State Variables and Their Timelines

The preceding definition of 'goal' begs the question of whether all types of operational intent can really be specified as constraints on the values of state variables. We believe the answer is "yes" given that the whole purpose of operations is to change the state of a system under control in specified ways. Examples of physical states include device status (configuration, temperature, operating modes), dynamics (vehicle position and attitude, gimbal angles, wheel rotations), resources (power and energy, propellant, data storage, bandwidth), and data (science observations, engineering measurements). Given the wide variety of kinds of state variables, goals can represent low-level objectives, as we've seen with the power switch goal, and they can also represent high-level objectives that drive an entire phase of a mission, such as "spacecraft is landed on Mars within ellipse x by time t". High-level goals get decomposed into a myriad of supporting goals, to be described in Section V.

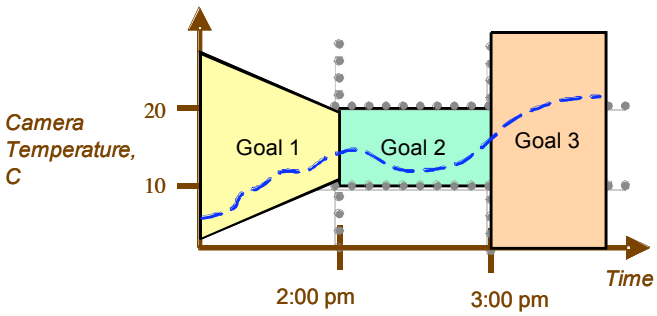

Figure 7. Every state variable contains two timelines: intent and knowledge. The intent timeline holds goals, ordered according to time, as depicted by the colored polygons. The knowledge timeline holds estimated state values, as depicted by the dashed blue line. A goal succeeds during execution if its constraint is satisfied by the estimated state history during its time interval.

As mentioned earlier, every state variable in the control system corresponds to a physical state in the system under control. Each state variable contains two timelines: an *intent timeline* that holds the goals, ordered according to time, and a *knowledge timeline* that holds estimated state up to the present time (see Figure 7). During execution a goal that extends from time t_1 to time t_2 succeeds if the estimated state history between t_1 and t_2 satisfies the goal's constraint. Suitable displays of these timelines enable an operator to examine the past, present, and future, and see how the system under control behaved, insofar as it can be estimated from available evidence.

An important aspect of the state knowledge timeline is that estimates contain not only the 'best estimated value' of the physical state but also some representation of the amount of uncertainty. This inclusion of uncertainty is very important for robust control because it acknowledges that sensors and actuators are imperfect as well as our models of devices and the environment. If the evidence available to a state estimator is noisy or conflicting or unavailable, then the uncertainty of its estimate must be correspondingly higher. This aspect of estimate uncertainty is important in two ways: it allows an estimator to be honest about the evidence and not pretend that its estimates are facts, and it enables a controller to exercise caution when its control decisions depend on highly uncertain estimates. If an activity depends on some minimum level of certainty of state estimates, then a goal can specify such a constraint since uncertainty is part of the value representation of every state variable.

D. Goal Network and Temporal Constraint Network

To achieve coordinated control, goals must be organized so that temporal dependencies are honored and that incompatible goals are not scheduled for concurrent execution. The structure in which a coordinated schedule of goals is arranged is called a goal network, consisting of goals situated in a temporal constraint network. Every goal has a starting and ending time point, and goals that must begin or end simultaneously share the same time point. Every time point has a time window which designates the earliest possible and latest possible times for the time point to fire during execution. This window—which may be as small as an instant of time—is a determined by the temporal constraints in the network. A temporal constraint specifies minimum and maximum time duration between two time points. The time windows of time points are updated by a temporal propagation algorithm as temporal constraints are added or modified during scheduling and as time points fire during execution.

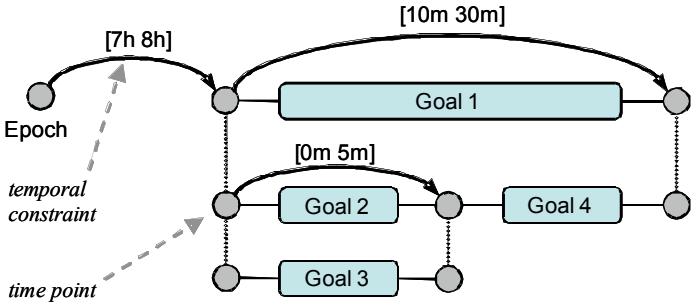

Figure 8. Example goal network. A goal network consists of goals situated within a temporal constraint network (TCN), with its starting and ending times governed by the time point attached to its left and right sides, respectively. Time points connected by a dashed line are the same time point, meaning that multiple goals may start or end simultaneously. The Epoch is a special singleton time point designating a predefined reference instant in time.

E. Layered Control Architecture

A canonical architecture has emerged in the research community based on the notion of layered control loops that address control problems at various timescales and level of abstraction[9]. The layers run concurrently and asynchronously to provide a combination of responsiveness and robustness. Figure 9 shows the MDS architecture consisting of four layers: Control, Execution, Planning, and Presentation. Note that the Presentation layer includes human operators and their decision-making *as part of* the control system. It is here that operational intent is first specified in the form of goals and it is here that the longest control loops are closed.

Goals can exist at very different levels of abstraction, from high-level goals such as "Be landed on Mars" to low-level goals such as "Maintain switch 27 in closed state". Goals also address control problems at various timescales, meaning that some goals have relatively short durations and/or stringent requirements on starting or ending times, while other goals have relatively long durations and/or flexible starting and ending times. The lowest layer—the Control Layer—provides reactive control with real-time responsiveness while the highest automated layer—the Planning Layer—generates globally consistent plans, ready for execution. The Execution layer executes the current

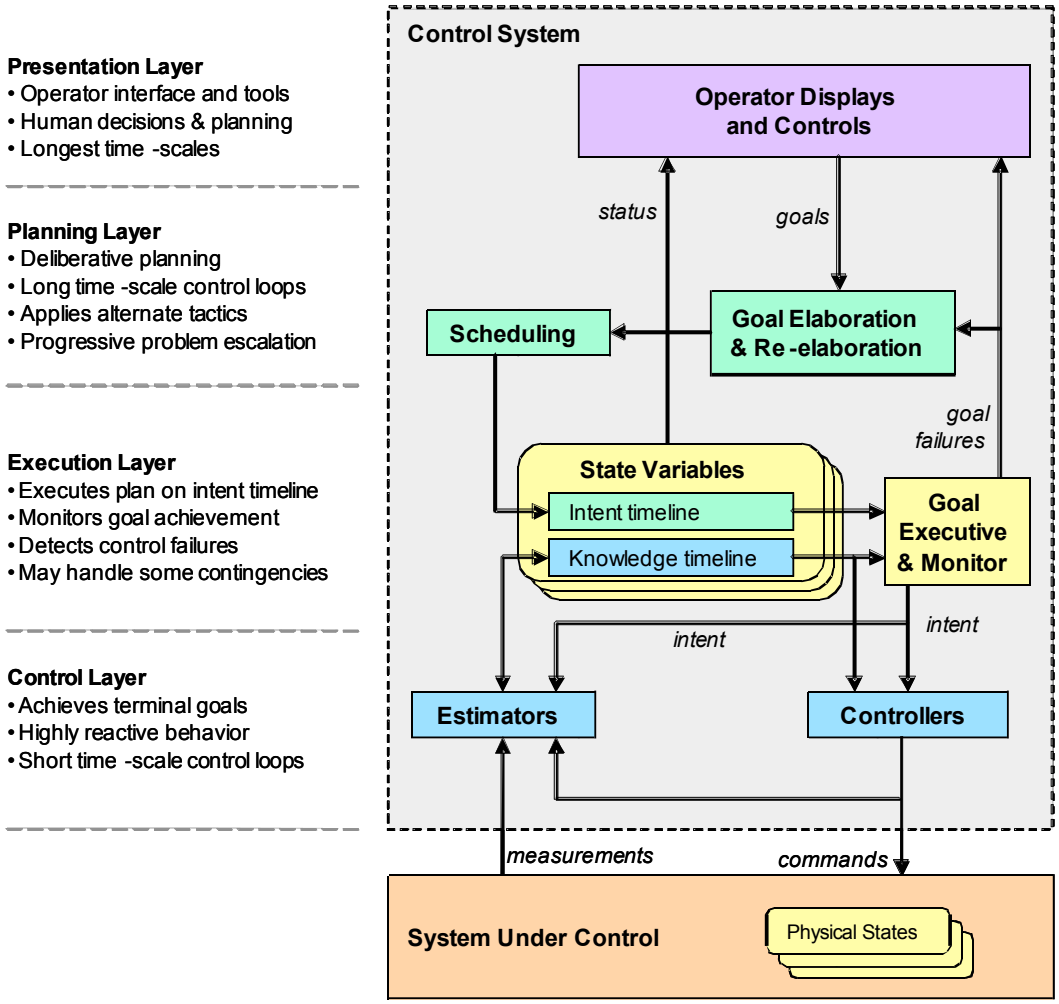

Figure 9. Software architecture layers in MDS. All layers operate asynchronously and concurrently to achieve goals. The control layer operates on short time-scales, reacting to local observations, while the planning layer operates on longer time-scales, coordinating system-wide behavior, including system-level fault responses. State variables in the Control System correspond to physical states in the System Under Control.

goal network, determining when to advance to the next goal on each intent timeline, consistent with temporal constraints and goal transition conditions. The Presentation Layer provides the displays and controls used by operators, where the highest level decision-making occurs.

As a goal is being executed in the Control Layer, its status is independently monitored in the Execution Layer by evaluating its constraint against state estimates on the associated knowledge timeline. If the goal is still satisfiable then execution proceeds normally, but if not, the goal status monitor reports the failed goal to the Planning Layer while the Control Layer continues to try to achieve the now-failed goal. The Planning Layer responds to the goal failure through the process of re-elaboration, as described in the next section.

V. Goal Elaboration

A goal that is directly executable by the Control Layer can simply be submitted by an operator and the Control Layer will try to achieve it. However, few goals can be achieved in isolation without considering their implications on other related states of the system, and many goals simply cannot be achieved without the support of other goals on other state variables that have an effect on the state of the primary goal. For example, a goal on spacecraft attitude has implications on antenna pointing and camera pointing, so it has effects that might be inconsistent with goals on those *affected* states. Likewise, a goal on spacecraft attitude can only be achieved if goals on *affecting*

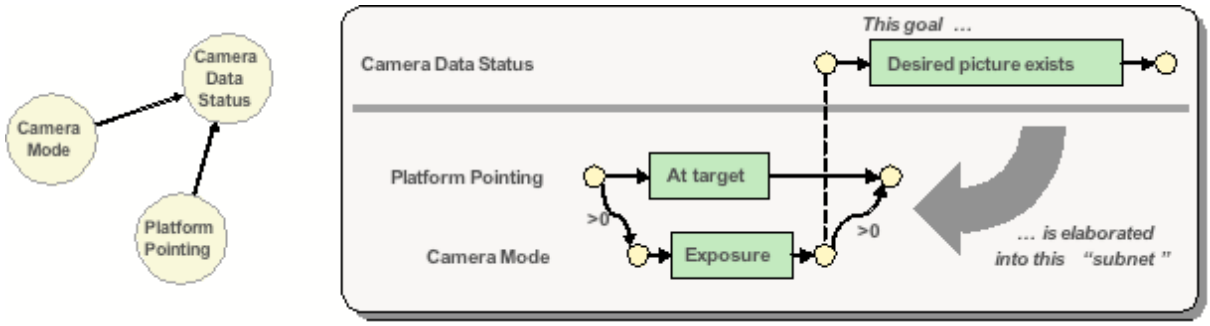

Figure 10. State effects diagram and goal elaboration. A state effects diagram, produced during analysis of the system under control, shows what physical states affect other states. A goal elaboration, produced during operations engineering, shows a goal above the double line and its supporting goals below the line, reflecting the state-to-state effects that must be managed (or required) to achieve the original goal.

states are achieved or maintained, such as adequate power and propellant, warm catalyst beds for the thrusters, and healthy inertial measurement units.

The engineering knowledge of what states affect other states in the system under control is represented in a *state effects diagram*, as shown in Figure 10, and that knowledge guides the design of fundamental "blocks" of goals that can be assembled into plans that respect the causality among state variables in the system under control. These fundamental blocks, called goal elaborations, specify supporting goals on related state variables that need to be satisfied to achieve the original goal, or make the original goal more likely to succeed. Each type of goal has an associated elaborator that generates its supporting goals (if any), and those supporting goals may themselves have elaborations with supporting goals, so goal elaboration is a hierarchical process that finishes when no more goals have elaborations. The design of goal elaborations is based on the state effects diagram and the application of a few rules[12].

Since there may be more than one way to achieve a goal, an elaborator may contain multiple tactics. For example, if the pointing platform had failed, but was part of a spacecraft, then an alternate tactic would be to change the orientation of the spacecraft in order to point the camera at the target. Alternate tactics may be explored during initial planning and scheduling and also in response to goal failures.

VI. Goal-Based Fault Tolerance

Most spacecraft designed for science missions contain a bifurcated control architecture that consists of a time-based command sequencer for nominal, pre-planned activities, and an event-driven monitor-and-response mechanism for unplanned events, typically for hardware faults. Traditionally, these two different control mechanisms are engineered by different teams, and the fault protection system is integrated late in the development lifecycle. The integration is often time-consuming because of the difficulty of coordinating the actions of these two concurrent control mechanisms. Rasmussen explains that this bifurcated architecture results in unwarranted

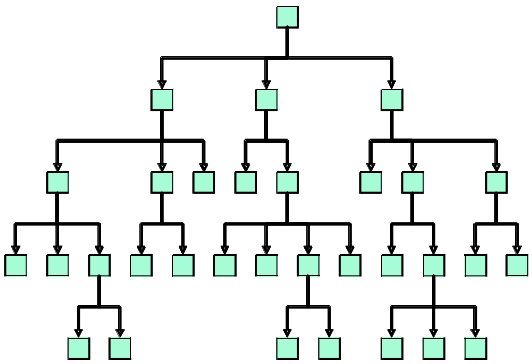

Figure 11. Hierarchical goal elaboration. The end result of goal elaboration is a tree of supporting goals with the "leaf goals" being goals that have no further elaboration. The goals still need to be scheduled into a plan (a goal network) and then executed. All of the goals in the tree—not just the leaf goals—are monitored during execution since each goal represents intent with respect to a particular state variable whose value is actively estimated during execution.

complexity, poorly understood behavior, incomplete coverage, brittle design, and loss of confidence[5]. In the MDS architecture, fault protection employs the same mechanisms as nominal control, and so coordination of fault responses with normal activities is the same as coordination of normal activities alone. Further, since fault response logic is programmed in goal elaborators, it can be designed alongside the logic for nominal elaborations.

As described earlier and as depicted in Figure 9, every goal is actively monitored during execution and if its constraint is violated, the failed goal is queued for re-elaboration. Specifically, the failed goal is delivered to the goal elaborator that generated it, namely, the elaborator of the *parent* of the failed goal. Depending on the logic programmed in that elaborator, it can produce one of four kinds of responses. First, it can choose to do nothing, effectively continuing on a best-effort basis since the achiever of the failed goal continues trying to achieve the goal until told otherwise. This option may be used when there is absolutely nothing better to do. Second, it can choose to re-elaborate, which involves retracting the failed goal, its supporting goals, and potentially its siblings, and replacing them with a different set of goals, all in an effort to achieve the parent goal using a different tactic. This option permits a localized fault response that avoids disrupting unaffected activities. Third, it can escalate the problem to *its* parent. This option is used when the elaborator has no knowledge of how to respond to the particular failure. If escalation propagates all the way to the top, then safe-mode is invoked. The fourth and final option is that the elaborator can immediately invoke safe-mode, thereby bypassing any higher level consideration of responses and shortcutting the response time. This option is useful when it is clear that no other response is feasible.

VII. Verification & Validation

Verification—whether applied to command sequences or goal networks—determines if a given set of objectives can be achieved, given models of how the system under control works and predictions of its state. Between these operational paradigms there are two main differences with respect to V&V. The first difference is in how verification is performed at planning time. As Morris describes[2], verification of command sequences employs a "product flow" approach from tool-to-tool, starting from an observation plan, and then adding resource constraints, then moving to an activity plan, followed by sequence expansion, and ultimately resulting in a command sequence that has been checked in different ways by different tools. The variety of tool-specific representations results in a complicated uplink process. In contrast, goals represent both activities and resources, so actions and effects can be reasoned about side-by-side. Goals can overlap in time because they are eventually merged by the planning system, allowing for optimization by consolidation of redundant activities. State effects and elaborations allow the operators to trace resource violations back to the source. Thus, when verification detects a violation, it is easier to trace back to the conflict in a goal network than in a command sequence.

The second difference between the two approaches is in the availability of objectives and models to support run-time validation. In other words, how do we know if objectives are being satisfied during execution? In a goal-driven system the objectives (the goals) can be actively monitored because they are part of the uplinked product. Goals represent activity-specific requirements—control objectives, resource requirements, causal dependencies and flight rules—and if any are violated during execution, the control system notifies the appropriate elaborator for a context-specific response. In contrast, a command sequence contains no explicit representation of its objective, so protection against misbehavior falls to a fault protection system that checks general flight rules and often issues responses (such as safe-mode) that are often disruptive to activities unaffected by the fault. A goal-based system *will* go to safe-mode, but only as a last resort when no activity-specific response is available.

VIII. Evolving the Role of Operations

Regardless of the operations paradigm, the most important thing for operators is that they understand the system under control and the effects of any directives given to it, and that they can reconcile observations with expectations. Obviously, this is most challenging when things don't go as planned. GBO helps in this regard because raw measurements are interpreted into state-based telemetry, goal failures are detected and reported automatically, and parent-child relationships between goals can be traced in either direction to help understand causes and effects. Nonetheless, it's important that operators understand the state effects model because that describes how the system is expected to behave, not only in nominal situations but also in failure modes. Furthermore, since fault response logic is contained in goal elaborators, and is supposed to be consistent with the state effects model, it's important that operators understand that too. The surest way to do that is for operators to have ownership of goal elaborators and be responsible for their design, including updates to the state effects model when found to be wrong.

IX. Common Questions and Concerns

A. Observability

One commonly-expressed concern about GBO is that it may decrease an operator's observability into the behavior of the system. In fact, GBO allows for *improved* visibility because it emphasizes good estimates of system state. In addition to the low-level measurements that typically make up the bulk of spacecraft engineering telemetry streams today, the operator has access to more intuitive state-based telemetry. Furthermore, availability of the success/failure status of executed goals makes it easier for operators to quickly focus on activities that have not executed nominally. In command sequencing the operator's job of trying to understand the state of the system is much harder, especially when things go wrong. The operator must examine long streams of low-level telemetry and interpret it into state. This process often depends on "hand-me-down" understanding, consultation with resident flight software experts, and/or guess work.

B. Controllability

Another commonly-expressed concern about GBO is that it may decrease an operator's ability to exert detailed control over the system. This concern stems from a common interpretation of the word "goal" as a relatively high-level objective to be achieved, and knowing that in any mission, unusual situations may arise that require operators to take over detailed control of a vehicle. Such control may require execution of very low-level sequences, even down to the level of opening/closing switches. In fact, GBO allows such control because goals at *any* level of detail can be specified by operators, scheduled on the ground, and uplinked to the vehicle. During normal operations, of course, low-level goals are typically generated automatically from elaborations of higher-level goals, but this is not a requirement; operators can specify low-level goals directly and *not* specify higher-level goals, if desired. In general, operators can choose any level of controllability, from very low levels that exert very detailed and predictable control with minimal autonomy, to very high levels that permit *in situ* reactions to a range of variations and failures.

C. Predictability and Trustworthiness

Time-based command sequencing has an appeal of predictability in that specific commands are executed at specified times. If conditions are nominal and if operators have high-fidelity models of the spacecraft and its environment, then they can predict the state of the spacecraft with accuracy, even in the absence of telemetry. From this perspective, *any* autonomous control system—regardless of operational paradigm—seems less predictable because the timing of commands is determined onboard in reaction to local observations. Engineers sometimes erroneously say that such systems are nondeterministic, but it's a misuse of the term. Software systems are deterministic, meaning that given the same conditions, they will behave the same way. What they really mean is that command timing is unpredictable.

The concern over command predictability presents an irony. Every command sequence is designed for a purpose, but a command sequence is *less* likely to achieve its purpose than a goal network simply because the former is open loop and the latter is closed loop. Although both a command sequence and a goal network are prepared ahead of time based on predictions of state for the planned time of execution, the latter has the ability to condition its commanding on locally estimated state in real time. A goal-driven system can change command timing, or change a parameter value of a command, or even change what command to issue next. Also, as discussed earlier, when faults occur, the integrated nature of goal-based fault protection makes it easier to devise context-sensitive fault responses that allow unaffected activities to continue uninterrupted.

The concern about predictability is more likely a concern about trustworthiness. Operators have long accepted uncertainty in command timing in attitude control systems (ACS). Operators don't know exactly when or for how long an ACS will fire each thruster. For deep space missions it's a detail that *has* to be handled onboard since the dynamics and light-time delays preclude Earth-in-the-loop control. Nonetheless, ACS has earned respect as a reliable form of automation. In large part, its reliability stems from its grounding in control theory, which inspires some of the architectural principles of MDS: state variables and behavioral models as first-class architectural elements, separation of state estimation from control, and control actions based solely on estimated state and goals. The real challenge is to demonstrate that goal-based control systems can be as trustworthy as attitude control systems.

D. Goal-like Commands

Modern space missions increasingly employ commands that are goal-like in the sense that they specify an objective to be achieved and/or maintained within a time interval. Examples of goal-like commands are pointing a spacecraft antenna at Earth, driving a rover to a waypoint, and capturing an image of a particular target with a

particular filter-wheel setting. In all cases such commands perform closed-loop control, making local decisions—about what commands to issue and when to issue them—based on local estimates of state, in order to achieve a desired state. Goal-like commands are created for the same reasons that motivate GBO: they perform tasks that simply cannot be done from Earth (e.g. attitude control at Mars), they increase science return by eliminating round-trip communication delays with Earth (e.g. drive rover to target), and they simplify operations by leaving the details to "programmed behaviors" in flight software. Of course, these benefits come at the price of increased flight software complexity and testing complexity, but it's a good tradeoff for the mission as a whole.

Given the benefits of goal-like commands, one might ask if there's much value in a goal-based architecture, such as described in this paper. If you look only at individual commands in isolation and have only a few goal-like commands that don't interact, then it's possible to use *ad hoc* techniques for representing and processing goals. The real value comes when goals are elevated to be the principle basis for operations, including resource management and fault protection[4]. Then, architectural structure and mechanisms become essential to predictable, coordinated control. Perhaps the three most important architectural principles for goal-based operations are: (1) representation of operational intent in terms of the system under control, independent of control system design; (2) separation of concerns (separation of control system from system under control, separation of state knowledge from intent, separation of state estimation from control, separation of goal achievement from goal status monitoring, and separation of reactive and deliberative processing); and (3) a natural modularization of estimators, controllers, and goal elaborators according to the physical states of the system under control.

E. Operator Expertise

In any semi-automated system there is a valid concern that operators may become so reliant on automation that they become less able to knowledgeably intervene in the event of an off-nominal situation. This is an issue associated with automation in general and is not peculiar to GBO. Part of the solution to this problem lies in a combination of things: operator interfaces that provide visibility into system state at different levels of detail; rigorous training that includes simulated failures and that exercise more complex diagnostic procedures; and operations procedures that involve periodic confirmation of consistency between low-level telemetry and state estimates. The most important part of the solution for GBO, though, is probably direct operator responsibility for the design and maintenance of goal elaborators because that requires a detailed understanding of the system being controlled.

X. Conclusions

Time-based command sequencing has served space missions well for decades, but its limitations have become more evident as missions levy requirements for more autonomous activities such as opportunistic science observations, vehicle hazard avoidance, and fast localized fault responses. Command sequencing has difficulty accommodating such requirements because it is inherently an open-loop control paradigm. Goal-based operation (GBO), in contrast, is inherently a closed-loop paradigm that more readily accommodates such requirements in an architecturally principled way.

Although the semantics of a command sequence are very simple, its intended effects are not knowable without considerable information that gets discarded during its preparation. In contrast, the intended effects of a goal network are present for all to see since they are represented explicitly in state constraints on state timelines, with time constraints for ordering and timing. This representation not only facilitates verification at planning time, but it is also carried into the uplinked product where it supports run-time validation and fault detection.

The versatility of the concept of 'goal' enables seemingly different things to be represented and processed in a more uniform way. Defined as a state constraint over a time interval, goals can represent not only control targets but also resource allocations and preconditions for activities, including dependencies on the values of uncontrollable states. The expressive power of goals holds the prospect for more consistency across tools used in the uplink process, eliminating the need for conversion scripts[2].

In a GBO paradigm, goal elaborators should become a focus of operations engineering because they contain the "how to do it" knowledge for each kind of goal. Elaborators are designed in accord with the state effects models, which contain essential system-wide knowledge that operators need to know and understand, particularly when called to intervene in unanticipated situations. Regardless of whether some elaborators execute on the ground and some in flight, operators should be the designers.

Acknowledgments

The work described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. The authors acknowledge Robert D. Rasmussen as the architect of MDS, which defines the concept of goal-based operations presented in this paper.

References

[1]Dvorak, D. L., Ingham, M. D., Morris, J. R., and Gersh, J., "Goal-Based Operations: An Overview," Proceedings of AIAA Infotech@Aerospace Conference, AIAA, Rohnert Park, CA, May 2007.

[2]Morris, J. R., Ingham, M. D., Mishkin, A. H., Rasmussen, R. D., and Starbird, T. W., "Application of State Analysis and Goal-Based Operations to a MER Mission Scenario," Proceedings of SpaceOps 2006 Conference, Rome, Italy, June 2006.

[3]Bennett, M., Dvorak, D., Hutcherson, J., Ingham, M., Rasmussen, R., and Wagner, D., "An Architectural Pattern for Goal-Based Control," Proceedings of 2008 IEEE Aerospace Conference, Big Sky, MT, March 2008.

[4]Rasmussen, R. D., "Goal-Based Fault Tolerance," Proceedings of IEEE Aerospace Conference 2001, Big Sky, Montana, March 2001.

[5]Rasmussen, R.D., "GN&C Fault Protection Fundamentals," 31st Annual AAS Guidance and Control Conference, Breckenridge, CO, February 2008.

[6]Ingham, M., Rasmussen, R., Bennett, M., and Moncada, A., "Engineering Complex Embedded Systems with State Analysis and the Mission Data System", AIAA Journal of Aerospace Computing, Information and Communication, Dec. 2005.

[7]Barrett, A., Knight, R., Morris, R., Rasmussen, R., "Mission Planning and Execution Within the Mission Data System", Proceedings of the 4th International Workshop on Planning and Scheduling for Space (IWPSS 2004), Darmstadt, Germany, June 2004.

[8]Gat, E., "Non-Linear Sequencing", Proceedings of the 1999 IEEE Aerospace Conference, Snowmass at Aspen, CO, March 1999.