An Architectural Pattern for Goal-Based Control

Matthew Bennett, Daniel Dvorak, Joseph Hutcherson, Michel Ingham, Robert Rasmussen, David Wagner
Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109
818-393-6426
{ matthew.b.bennett, daniel.l.dvorak, joseph.o.hutcherson, michel.d.ingham, robert.d.rasmussen, david.a.wagner }
@jpl.nasa.gov

Abstract—Time-based command sequencing is the traditional paradigm for control of spacecraft and rovers in NASA's robotic missions, but this paradigm has been increasingly strained to accommodate today's missions. Goal-based control is a new paradigm that supports time-driven and event-driven operation in a more natural way and permits a melding of sequencing and fault protection into a single control paradigm. This paper describes one approach to goal-based control as an architectural pattern in terms of purpose, motivation, structure, applicability, and consequences. This paper is intended to help flight and ground software engineers understand the new paradigm and how it compares to time-based sequencing. [1][2]

TABLE OF CONTENTS

1. INTRODUCTION

Much software in aerospace systems is devoted to making a system do what its operators intend. These intentions range from high-level mission plans to low-level hardware modes and from long timescales to short timescales. Fundamentally, the software for carrying out such intentions must perform closed-loop control, with some control loops closed through automation and others closed through human-in-the-loop analysis and decision-making. This "control software" must also coordinate activities at multiple timescales and multiple levels of system and subsystem decomposition and must respond appropriately to failures.

For nominal operation at the system level, the dominant paradigm for such control in today's interplanetary missions is time-based command sequencing, where there has been considerable engineering investment in flight software (sequence management, sequence execution) and ground software (tools for planning, scheduling, resource modeling, and flight-rule checking for activities and command sequences). Fault protection generally exploits sequencing capabilities to some extent when responses are necessary, usually at the expense of planned activities. For the most part, however, fault protection is independent in design, character, and execution from sequencing.

The predominant feature of this approach is that correct operation of sequences depends on a mixture of ground-based a priori predictions and a posteriori checks, in addition to on-board health and safety checks that are typically linked only weakly to planned activities. The driving intent of a sequence is largely absent from the final uplinked product, and the models essential to its structure are also typically left behind, replaced by timers. That is, if the system does not perform as predicted, discrepancies may go unnoticed by the flight system, which is unaware of the overarching intent. If an anomaly *is* detected, the flight system does not have sufficient knowledge of this intent or of system behavior to safely restore operation. Sometimes, the actual real-time progression of sequences is conditioned on observations made during execution, but more usually these sequences proceed based purely on time.

This paradigm has been increasingly strained to accommodate today's more complex missions, which require more localized capabilities such as autonomous resource management, vehicle mobility with hazard avoidance, opportunistic science observations, and so on. In these missions, keeping the ground in the loop may be impractical or even impossible, plans may be frequently subject to change given a dynamic situation, and fault responses may need to restore planned activities rather than halting them.

An emerging paradigm—known as goal-based control, which is a form of closed-loop control—can provide such capabilities in a more natural way while still preserving the option to "command" a system at a very detailed level. A goal, in contrast to a command or command sequence, is inherently a closed-loop directive since it specifies an intention on the state of the system under control over a period of time—an intention that is monitored continuously during execution and therefore knowingly succeeds or fails.

[1] 1-4244-1488-1/08/$25.00 ©2008 IEEE.
[2] IEEEAC paper #1542, Version 9, Updated December 20, 2007.

Note that a goal specifies *what* to do but not *how* to do it, thus leaving options open to the control system—options that may be exercised in both normal operations and in fault responses.

Since a goal specifies a constraint on state over a period of time, its execution can not only be time-driven but also event-driven. Moreover, the uniform and complete description of intent as state and time constraints provides a basis for model-based reasoning that permits, for the first time, a melding of sequencing and fault protection into a single, highly flexible control paradigm.

The objective of this paper is to describe one particular approach for goal-based control to software engineers as an architectural pattern. As a pattern, the description focuses on types of elements and relationships, their mechanisms of interaction, and rules for combining them. This paper describes a set of smaller patterns that are composed into the larger architectural pattern, emphasizing several architectural principles including:

- 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 reactive and deliberative processing
- Distinction of concepts:
  - distinction between goal and command
  - distinction between measurement and state estimate

Major elements of the pattern include state variables, goal elaboration, goal scheduling, goal status monitoring, goal timeline execution, state estimation, controllers, and hardware adapters. As a pattern, the ideas are language-independent, so software engineers can implement them in any programming language and begin to experiment with goal-based control. In fact, C++ and Java versions already exist and are available for use.

2. BACKGROUND

2.1 Architectural Patterns

Architectural design involves making decisions that have system-wide impact. Architectural patterns help architects understand the impact of the architectural decisions at the time these decisions are made, because patterns contain information about consequences and context of the pattern usage. The patterns used here are based roughly on the software design pattern methodology described in [11]. Section 3 of this paper describes an architectural pattern using a template of the following five elements:

- *Purpose*. What does the design pattern do? What problem does it address?
- *Motivation.* This is usually a scenario that illustrates how the pattern solves a problem.
- *Structure*. Includes descriptions of the participants and collaborations between them.
- *Applicability*. Conditions in which this pattern applies, or does not apply.
- *Consequences*. What are the trade-offs and results of using the pattern? What are its limitations?

2.2 Separation of Control System and System Under Control

Fundamental to these control patterns is the concept of a separation between the *Control System* and the *System Under Control.* The System Under Control is what the Control System is intended to control. A clear boundary is essential to defining clear models of behavior and establishing clear control authority. Control system designers can choose to define the boundary at a hardware interface, or a subsystem interface, or anywhere else, as long as the boundary is formally defined and the boundary rules described in these patterns are adhered to.

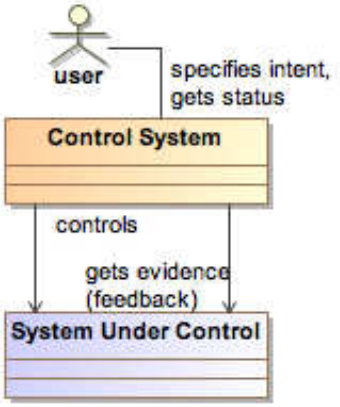

Figure 1 – Separation of Control System and System Under Control

2.3 State-Based Control

Control systems are designed to translate some notion of user intent into actions that cause that intent to be achieved. State-based control systems make a clear distinction between the intent and the actions that the control system may perform to achieve the intent. Intent expresses a *desired outcome* in some physical state of the system under control, rather than a script or sequence of actions needed to achieve it.

Control theory defines a *closed-loop* control system as one in which direct feedback from the system under control can be used to determine the effectiveness of control actions, allowing the control system to actively compare the state of the system with the intent, and perform control actions that

attempt to keep the system within a range of acceptable behavior. This is distinct from an *open-loop* system where control actions are performed without feedback. Feedback allows the control system to react to unpredictable or unexpected effects in the physics of the system, and compensate for them.

Intent defines what the external users want the system to accomplish. Intent is generally defined as a range of acceptable behavior that may optionally be augmented with additional performance measures. Intent is expressed through a *Goal*. A goal is intended to constrain the state of the physical system, but in order to do this a representation of that physical state must exist in the control system. As shown in Figure 2, a *Software State Variable* in the control system represents the value of *a Physical State Variable* in the system under control. Formally, then, a goal specifies a constraint on the values of a software state variable over an interval of time. All control decisions are based on the relationship between estimated state and desired state (goals). Patterns for estimation and control using this basic paradigm are described later in this paper.

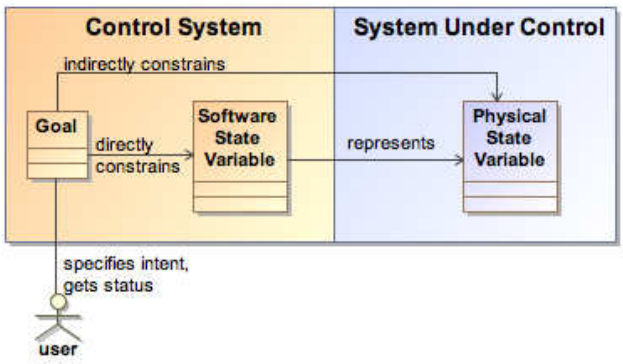

Figure 2 – State Variables and Goals

Real control systems, of course, usually control multiple elements in the system under control, with multiple concurrent objectives. The physical state variables of those elements may be physically coupled, or interdependent. Further, some physical state variables may not be directly measurable, and many are only indirectly controllable. Even as the complexity of the system increases, the challenge for designers is to control the whole system in a coordinated way.

Real control systems often employ both *reactive* and *deliberative* control, according to the timescale for reacting to feedback and the form of reasoning applied in achieving intent. In relative terms, reactive control operates on faster timescales and makes control decisions based on a narrow scope of awareness. Deliberative control operates on slower timescales and makes control decisions based on a wider scope of awareness. Deliberation is driven by the need to anticipate future demands and to attempt to ensure that conditions appropriate to those demands will have been established, where such conditions cannot be achieved instantaneously, either due to the speed of system dynamics or to potential conflicts.

The design patterns presented here are intended to provide architectural solutions for achieving closed-loop control in these increasingly complex systems. Section 3 first develops the patterns for reactive control involving a single physical state variable. Later, the section develops the patterns for deliberative control, addressing the complexity introduced when the control intent demands coordination of multiple physical states variables.

3. DESIGN PATTERNS

3.1 State Estimation

Purpose

Define an architectural pattern for estimating the values of physical state variables based on available evidence; cleanly separate estimation from control.

Motivation

It is a common mistake in control system engineering to make control decisions based on incomplete knowledge of the state of the physical system as described in raw measurements. Measurements can be noisy, and intermittent. Filters are commonly applied to raw measurements, but if the results are buried in a control algorithm, they cannot easily be reused by other controllers. Worse, two different users of the same raw measurements, using different filters, may arrive at different estimates of the state of the physical system, resulting in control conflicts. Having a single explicit representation of any physical state variable of the system under control, using a single estimator, ensures consistent representation of that variable in the control system.

Structure

The primary structural elements that participate in this pattern are described in Figure 3.

First, the *State Variable* provides an explicit representation in the control system of a corresponding physical state variable of the system under control. This is also known as the *software* state variable to make the distinction clear. State variables are first-class entities in this pattern for three reasons. First, a direct representation in software of the physical state being controlled makes the software more readily understandable. In addition, telemetry based on state variable values is generally more informative than raw measurements because they refer to a physical state being monitored and possibly controlled. Second, the existence of the state variable permits a separation of concerns between estimation logic and control logic. Third, the existence of a single software state variable for each physical state variable ensures that there is one definitive source for estimates of a given physical state in the control system, and only one way

to access it. This avoids the common situation where two different controllers each have their own private-but-inconsistent estimates of a physical state, leading to surprising and potentially hazardous interactions.

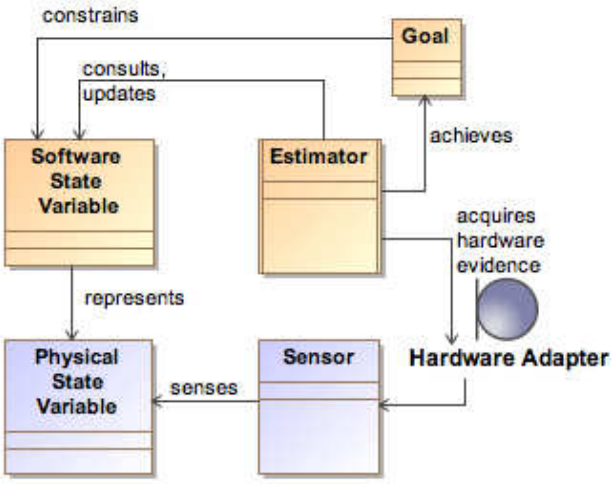

Figure 3 – Estimation Pattern (minus command evidence)

An *Estimator* is responsible for actively providing values to populate the state variable with the best estimate of its value from available evidence. In the simplest case an estimator may have only one source of evidence, such as measurements from a single sensor, but in the general case there are multiple sources of evidence: measurements from multiple sensors, commands sent to multiple actuators, and estimates of other state variables. The role of the estimator is to combine that evidence into a "best guess" of the value of the physical state, known as an *estimate*. Estimators must deal with discrete and continuous values, noisy, missing or corrupted measurements, and inconsistent evidence from multiple sources. These characteristics underscore why state estimation deserves special attention, quite apart from control.

The *Hardware Adapter* is simply a formal interface to the system under control. It provides a command interface for components that can be directly commanded (actuators), and a measurement interface to components that provide measurements of the system state (sensors). Its main role is to formalize the interface, but it can also serve to normalize the interface (like a device driver) and buffer data.

Measurements are raw samples delivered from sensors to an Estimator via a Hardware Adapter. They can have any form, since this is often determined by the sensor hardware. They should have time tags to eliminate timing ambiguity. It is important to remember that measurements are not state estimates; measurements are a type of evidence used by estimators to generate state estimates.

Commands are another type of evidence used by estimators, though not shown in Figure 3. Specifically, a command issued to an actuator affects one or more physical states, and can therefore provides evidence about the values of those physical states. Thus, estimators may acquire not only measurement evidence from sensor hardware adapters but also command evidence from actuator hardware adapters.

State Variables store information about the system state in the form of *State Value Functions*. These are distinct from Measurements in that State Value Functions must be continuous over time, and explicit about uncertainty. Measurements are readings at discrete points in time, and usually provide a single uncalibrated value. The process of Estimation (the role of the Estimator) involves calibration, smoothing, or noise elimination, and application of system models to determine and express uncertainty. State values can explicitly represent the fact that the system state may be unknown in situations where measurements are not available (e.g., if a sensor is powered off or failed).

Estimators produce state knowledge and repeatedly update software state variables. The precision and certainty of that state knowledge depends is driven by need, typically the need to control one or more physical state variables to a desired accuracy. Goals are used in this pattern to express constraints on the desired quality of the state knowledge, which may vary over time. Thus, estimators can be viewed as "achievers" for these goals.

Applicability

This pattern applies in any control system where knowledge of the target control states must be inferred from sensors or other indirect evidence.

Consequences

The existence of software state variables as first-class citizens in the architecture encourage a separation of concerns between estimation and control. The State Estimation pattern—and the State Control pattern that follows—formalize this separation. This separation is important because it decouples two concerns that have often been intertwined in control system software, making each concern easier to design, implement, verify, and reuse.

Also, the role of a software state variable as the sole source of information for estimates of its corresponding physical state variable eliminates the potential problem of multiple, private-but-inconsistent estimates within a control system.

This pattern also makes a clear distinction between measurements and state estimates. This is an important distinction for robust control systems because there are often multiple sources of evidence about the state of any single physical state variable—sources that should be examined and reconciled before making control decisions.

3.2 State Control

Purpose

Define an architectural pattern for exercising control over a given target system in a way that directly uses knowledge of the state of the system under control.

Motivation

Consider a simple thermostatic temperature control system. The system under control includes a temperature sensor and a heater which can be controlled by a switch. The goal is to maintain the temperature within a target range, or within a target range. Designing and implementing a software control system for this is straightforward. However, what if the underlying system changed (such as a change to the sensor) after the software was written, or you had to port the control system to different hardware? How hard would it be to pick apart the various models, assumptions, and algorithms from the code?

The closed-loop control pattern is intended to address this problem by defining placeholder elements for each of the key roles in a control loop, and rules governing separation of responsibilities between these elements.

Structure

The elements of this pattern are shown in Figure 4.

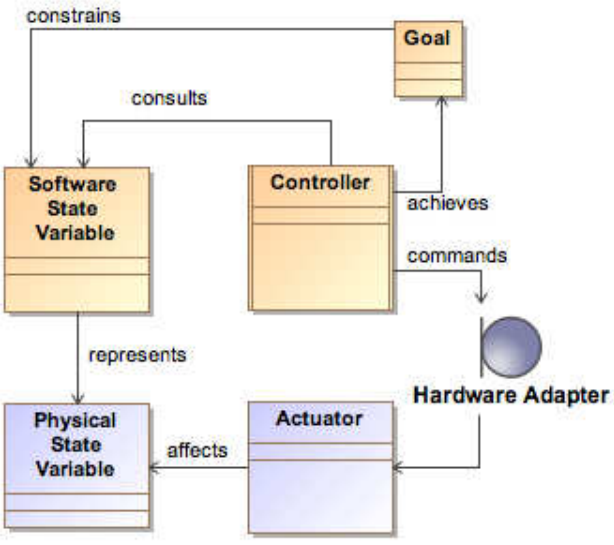

Figure 4 – State Control Pattern

As in the State Estimation pattern, a Hardware Adapter provides a line of separation between an *Actuator* in the system under control, and a *Controller* in the control system.

Control intent is expressed through the use of *Goals*, which express a constraint on the target state over an interval of time.

A *Controller* is responsible for any direct interactions with the system under control required to change or control the target physical state. The controller can issue commands to the target system through a Hardware Adapter. A controller is goal-directed in the sense that it issues commands as needed in order to drive the state of the physical system into agreement with the goal, or desired state. Note that the controller bases its decisions on the comparison between the goal and state knowledge provided by state variables. In other words, the controller never examines raw measurements to make control decisions, i.e., it never performs any internal state estimation.

Applicability

This pattern applies in situations where the control intent (the goal) can be expressed as a constraint on state over a time interval, or as a sequence of such constraints, and where the target state can be explicitly described in a state variable, and where the target state is directly controllable.

This pattern is typically limited to primitive states of the system under control that can be affected through actuators. The controller may rely on models of the system under control to determine appropriate control actions when the target state can only be indirectly controlled.

Consequences

This pattern, like the State Estimation pattern, supports the separation of concerns between estimation and control, and therefore makes control software easier to design, implement, and verify because control logic is cleanly separated from estimation logic.

This pattern places responsibility for control of a physical state variable within a single controller. As such, a controller may issue commands to multiple actuator hardware adapters that have an effect on the physical state being controlled.

3.3 Reactive Closed-Loop Control

Purpose

Define an architectural pattern for exercising simple closed-loop control over a given target system in a way that directly represents knowledge of the state of the system under control, distinguishes between raw evidence and state estimates, cleanly separates state estimation from control, and bases all control decisions on the relationship between estimated state and desired state.

Motivation

Consider a simple thermostatic temperature control system. The system under control includes a temperature sensor and a heater which can be controlled by a switch. The goal is to maintain the temperature within a target range, or within a target range. Designing and implementing a software control system for this is straightforward. However, what if

the underlying system changed (such as a change to the sensor) after the software was written, or you had to port the control system to different hardware? How hard would it be to pick apart the various models, assumptions, and algorithms from the code?

The reactive closed-loop control pattern is intended to address this problem by defining placeholder elements for each of the key roles in a control loop, and rules governing separation of responsibilities between these elements.

Structure

The structure of this pattern is a simple composition of the state estimation pattern and the state control pattern as shown in Figure 5.

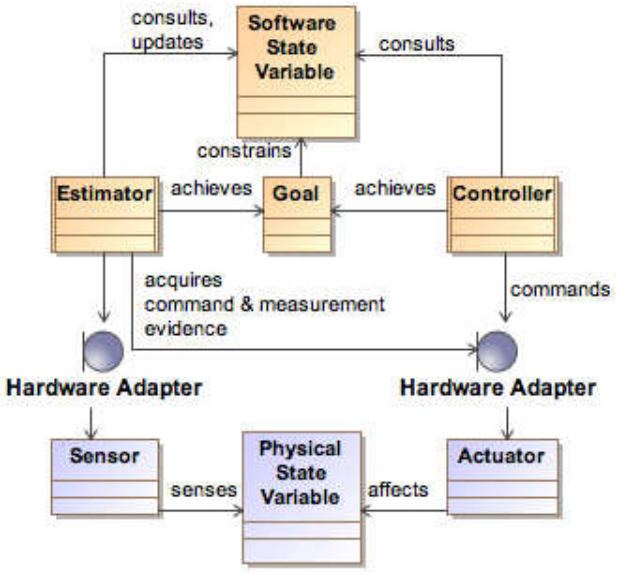

Figure 5 – Reactive Closed-Loop Control

What is important to note in the structure of this pattern is how it can be composed from the sub patterns due to the clean separation between estimation and control. Estimation and control are separate functions that only interact through the state variable.

Applicability

This pattern applies in situations where the intent (the goal) can be expressed as a constraint on state over a time interval, or as a sequence of such constraints. In the simplest case, the goal may be statically built into the system. In the more general case, an external sequencing mechanism delivers goals in order, as described later in the section 3.7, Executive Control.

Consequences

Some states can only be controlled indirectly. In this case the pattern may extend, and control loops may overlap one another via common state variables.

As long as the system is accurately modeled and estimation and control algorithms are faithfully executed, this pattern works for all control problems where the intent can be expressed as a single constraint, or at least a single constraint at a time. The pattern can be extended to support more complex behaviors in the following ways:

* Complex constraints (e.g., trajectory) – Here the goal includes timing information that describes a path through state space over time. An example of this is a *transition goal*, which is defined as a goal that allows for the transition from one stable state value to another. Transition goals express intent to have the state arrive at a target value, yet avoid a determination of failure if the state is not immediately being satisfied. For example, a transition goal on a temperature state variable might be defined so that it is succeeding as long as the temperature is moving toward the target value, whereas a *maintenance goal* would be defined so that any excursion from the constrained value range would be considered a failure.

* Hierarchical layering of achievers – goal achievers can be organized in a control hierarchy whereby a higher-level achiever issues goals to subordinate achievers to coordinate their actions in real time. An example is a position & heading controller for a Mars rover that issues real-time goals to the multiple driving and steering controllers.

* External sequencing of constraints – this approach is commonly used in robotic systems not only to sequence the constraints on a single state, but also to coordinate the application of goals applied to many states. See section 3.7, Executive Control.

These patterns can be combined in various ways to implement quite complex behaviors. A common limitation, though, is the limited tolerance for faults. In particular, the sequencing of goals into complex activities will typically describe one plan or script with all events ordered in time, or possibly sequenced according to states being achieved. If something breaks, or something unexpected happens, these scripts have only a limited ability to recover because there is no explicit representation of the higher-order intent, and no formal mechanism for expressing alternative methods to accomplish them. This limitation motivates the Deliberative Closed-Loop Control Pattern described in section 3.9.

3.4 Goal Network

Purpose

Define an architectural pattern to represent the relationships between a set of goals on a set of state variables that specify control coordination across states and over time.

Motivation

The primitive patterns described thus far provide the means to control state variables individually. In order to coordinate

control of multiple states, a way is needed to represent relationships among goals on different state variables.

Structure

A *Goal Network* (see Figure 6) is primarily a container for a set of goals and their associated software state variables. To make any sense as a plan, goals must be temporally related with one another. This is done using *Time Points* and *Temporal Constraints.*

A time point represents an abstract event. Every goal associates with exactly one starting time point, and one ending time point. However, time points can be shared by many goals.

Time points carry no internal relationship to time. Instead, all temporal relationships are represented through *Temporal Constraint* objects, which also associate with one starting time point and one ending time point. A temporal constraint can specify a minimum and maximum duration allowed between two time points, or simply a sequential ordering constraint.

A *Goal Network* contains goals, time points, and temporal constraints, as shown in Figure 6. The term "goal network" is used because the topology of the container is that of a directed graph where the time points are the nodes, and the goals and temporal constraints are the edges. The "parent" relationship shown in the figure means that each goal has a link to its parent goal. This parent/child relationship is populated during goal elaboration, as explained in section 3.5, Goal Elaboration.

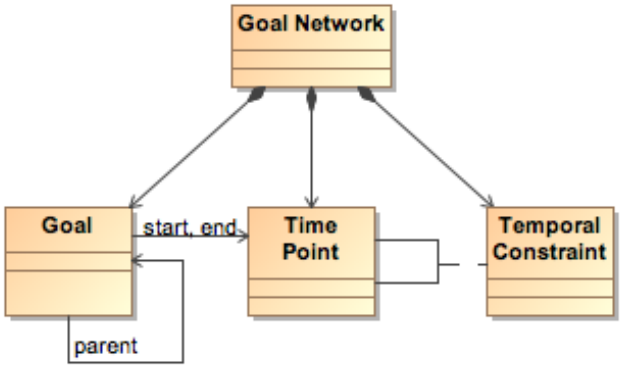

Figure 6 – Goal Network

Since time points can be associated with different goals on different state variables, they enable coordination of goals across different state variables. Figure 7 depicts an example goal network. The yellow circles represent time points. The green boxes represent goals aligned along state time lines. Time points joined by vertical lines indicate that those time points are shared, representing events that connect the state time lines. The arcs between time points represent temporal constraints, in this case indicating a minimum and maximum duration allowed between the given time points.

Earlier, a goal was defined as a constraint on the value of a state variable over an interval of time. Note that the goal's relationship with time is indirect, through its relationship with a starting and ending time point. Constraints on the duration of the goal are specified through temporal constraints on the bounding time points, and not as part of the goal itself. This separation of concerns allows for goals that do not have any temporal constraints, but it also allows temporal constraints that are not elaborated from goals to be added as part of the scheduling process, which will be described later.

Every goal instance in the network associates with a specific software state variable that it constrains. The set of goals associated with a single state variable can be computed into a sequential timeline through the process of ordering the time points into a topological ordering that satisfies all of the temporal constraints. This may result in overlapping goals on the same state variable. Thus, goals must have the property that allows them to be combined, or *merged*. The process of merging two goals may result in a new, more constrained goal. Merging occurs as part of the scheduling process described later.

A goal network can exist in two states. When initially constructed, an *unscheduled network* is simply the aggregation goals, time points, and temporal constraints representing a proposed plan. An *executable plan* has undergone scheduling and verification (described later) to merge and order goals according to temporal constraints, and verified that the proposed plan is achievable.

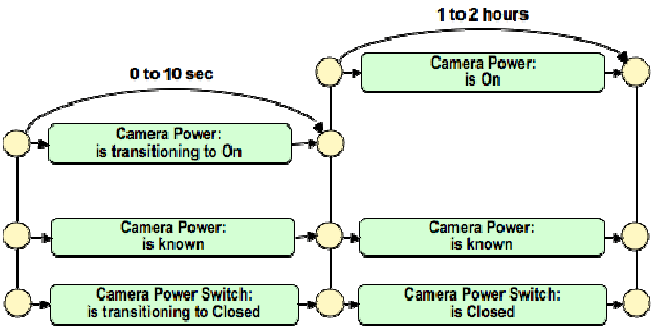

Figure 7 – Example goal network depiction

Applicability

This pattern becomes applicable as soon as coordinated control over multiples states is required.

Consequences

The use of temporal constraints to indirectly constrain event times allows for temporal flexibility in the plan. Contrast this with sequences having fixed event times, or purely sequential ordering.

3.5 Goal Elaboration

Purpose

Define an architectural pattern to represent causal dependency relationships among a set of goals.

Motivation

Coordinated control is about controlling several component states to accomplish some higher intent. In order to achieve intent on a given state variable, the control system may need to control other state variables that are causally related. In other words, goals may beget other supporting goals.

The elaboration pattern provides a formal mechanism by which a goal (an expression of intent) can specify dependencies on other supporting goals needed for their own achievement.

Structure

Elaboration is defined as the process of generating the additional supporting goals that would be needed in the same plan in order to accomplish the "parent" goal. A goal can associate with an Elaborator (see Figure 8) whose job it is to provide the additional plan elements needed to achieve the given goal. This set of supporting goals—plus any needed time points and temporal constraints—is known as a *Tactic*. Tactics are small goal networks defined in support of a particular parent goal. A goal can have more than one tactic, i.e., there may be more than one set of supporting goals that can help achieve the parent goal.

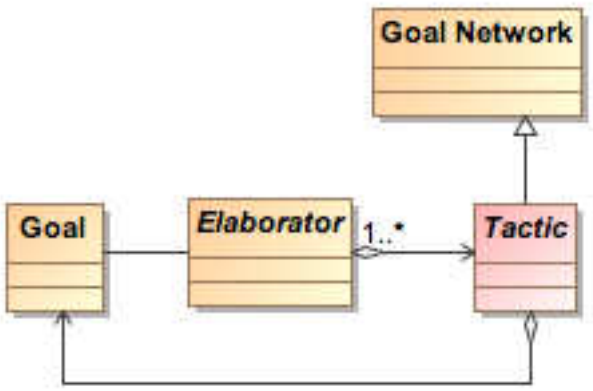

Figure 8 – Goal Elaboration Pattern

When the elaborator provides multiple tactics, only one can actually be used at a time in a single plan. An elaborator determines the tactic to apply depending on a variety of possible conditions, including current state variable values, scheduling failures, and failures of supporting goals during execution. In the latter case, goal elaboration in response to execution failures is called re-elaboration, and is described further in section 3.8, Goal Monitoring and Fault Response. In the case of a scheduling failure, a schedule using one tactic is determined to be unachievable, so the elaborator tries a different tactic if one is available.

The elaborator is separate from its goal mainly to separate specification of intent (the goal) from planning behavior that may or may not need to exist where the goals are executed.

The process of elaboration is performed at the level of an entire goal network. The initial set of goals is elaborated and then their supporting goals are elaborated recursively until the process bottoms out with goals having no elaborators (i.e., having no need for supporting goals).

Goal elaborations are normally defined such that supporting goals are on state variables that are either the same state variable as the parent goal, or are state variables that affect the parent goal's state variable [12].

The elaborator class can be more than a simple container of a set of predefined tactic sub-networks. It can use information available in the network context including current and historic states of the system (from state variables) to compute tactics appropriate to the given situation.

Applicability

This pattern applies to any system where coordinated control across multiple state variables is needed.

Consequences

A goal represents a desired outcome, and that encourages operators to think in terms of the outcome rather than in how it will be achieved. Of course, somebody still has to design the tactics to achieve the outcome, but that is done once, and then appropriate tactics are selected thereafter via goal elaboration.

The elaboration process can be invoked prior to execution (at plan design time) to elaborate operator-specified goals into the complete set of goals needed to accomplish the intent. A modified version of the process can be used during execution to respond to goal failures. (A goal with failing tactics can be re-planned by removing its current tactics from the goal network, and elaborating and scheduling an alternate tactic.)

3.6 Goal Planning and Scheduling

Purpose

Define a pattern for automatically preparing a goal network for execution.

Motivation

The power of the elaboration process is that it makes it possible to describe a high-level goal and all of the supporting goals it needs to be achieved. Maximum flexibility is achieved if the elaborations specify the fewest temporal constraints. Additional constraints need to be added to the goal network by the planning and scheduling process to create an executable goal network that is known to be "achievable". Achievability is determined by the planner by checking the executable goal network against the capabilities of the control system, and the physics of the system under control.

Structure

The planner/scheduler shown in Figure 9 represents the object performing planning and scheduling. A planner/scheduler is basically a constraint solver. Given a set of proposed goals, and temporal constraints (edges in a directed graph) the planner first elaborates all goals recursively to populate a complete set of goals needed to achieve the proposed goals. The planner then merges concurrent portions of overlapping goals on the same state variable. Merges that result in unachievable goals are rejected.

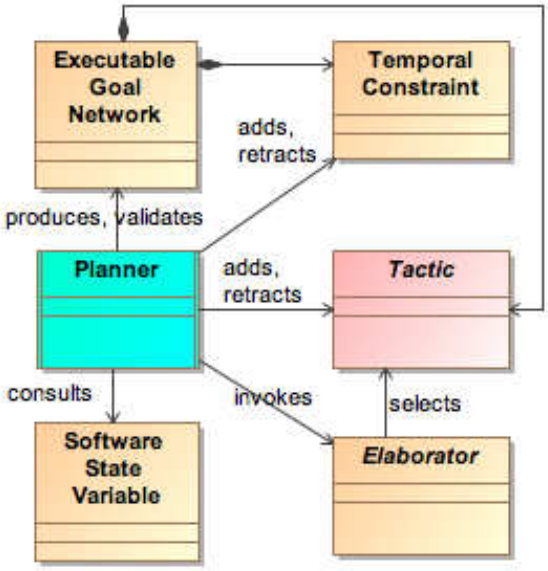

Figure 9 – Planner/Scheduler Interactions

Scheduling a goal network is the process by which an elaborated goal network is prepared for execution. At the end of elaboration, each state variable has goals and time points defined on it. Scheduling picks an ordering of the time points for each state variable. Goals that overlap over time intervals are merged. If merging results in an inconsistent goal, then a different time ordering is selected by the scheduler. In addition, the temporal constraints in the goal network are propagated to determine if the goal network is temporally consistent.

Before a scheduled goal network is ready for execution is must be validated. *Validation* of a scheduled goal network checks that sequential goals on state variable are consistent and that state predictions based on the ordered and merged goals meet the intent of the ordered and merged goals. Sequential goals are checked against transition achievability criteria to determine if a goal can begin executing when the previous goal's end condition is met. Predictions are computed using a mechanism called state projection which takes into consideration models for the effects of goals on affecting states, initial state variable values, physical models of state variable behavior, the behavior of the control system when it executes goals, and temporal constraints on the goals. If a consistency check for sequential goals or a state prediction check fails, the scheduled goal network is rejected, and the scheduler attempts a different ordering of time points. If all consistency checks succeed, then the ordered and merged goal net is promoted for execution as an executable goal network. The projections for each merged goal are saved with that merged goal in what is called an executable goal. If no ordering of time points results in a valid goal network, the planner/scheduler backtracks to choose another elaboration tactic.

Applicability

Needed if goal elaborations allow for temporal flexibility

Consequences

A key advantage of the planning and scheduling pattern is that problems can be detected before they happen by checking predictions for planned executable goals.

An executable goal network has been validated against models to ensure that every goal is achievable, and every transition from one goal to the next is achievable. Although the ordering in which goals are executed along any given state variable timeline will be fixed by this process, the network may still permit flexibility in the order in which events occur on different timelines, and the firing of time points.

3.7 Executive Control (Timeline Execution)

Purpose

Define an architectural pattern for execution of a planned and scheduled network of goals (an executable goal network) that will execute goals associated with planned activities according to a time-driven, and state-driven schedule.

Motivation

Given that operator intent is captured within a goal network as a series of goals placed upon state variables, how is this translated into activities performed by the system under control? Using the simple thermostatic control example, the

switch to the heater must be turned on at time t_0 and turned off at time t_1. During that span of time the heater must remain within a certain temperature range. In this example there are two events that must occur; turning the switch on then turning it off. During the period of time the switch is on the temperature of the heater must be monitored to ensure it remains within the range specified by the goal. The Executive Control pattern ensures the events occur within their temporal windows.

Structure

The purpose of the *Goal Executive*, as depicted in Figure 10, is to carry out the intent represented in an executable goal network by dispatching goals for execution at the appropriate times. The executive relies on the fact that goals express a continuous intent on a target state variable as long as they are in effect, and it is the responsibility of the control system to continue to try to achieve each assigned goal for each state variable until the next goal is dispatched for that state variable.

The executable goal network specifies the intent timelines for each of the state variables modeled within the control system. An intent timeline for a state variable is represented in the executable goal network as of a series of time points connected by merged, executable goals. Scheduled time points can retain some temporal flexibility as allowed by the set of temporal constraints in the goal network. As time is advanced by the executive, it is the responsibility of the executive to continually propagate the temporal constraints to refine the schedule of each time point.

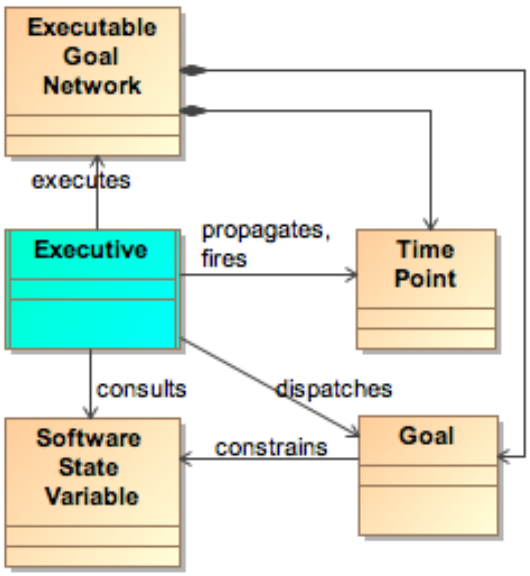

Figure 10 – Executive Pattern

Like goals, executable goals are bounded with starting and ending time points, some of which may have been generated during the scheduling process due to partially overlapping goals. A time point represents a time at which the executive must perform an action. The temporal constraints of the contributing goals determine the valid range of times, or window, in which the time point is considered open, or eligible to fire. To fire a time point, the Goal Executive checks that all the goals that have this time point as their starting time point are "ready" to start executing; that is, the post-conditions and pre-conditions associated with the transition from the current executable goal to the next executable goal on the timeline have been satisfied [2]. When the Goal Executive fires a time point it becomes "grounded" in time, removing any temporal flexibility it may have had, and the next executable goal's constraint is dispatched to the control system for execution. The Goal Executive will honor a not-ready transition status while within the eligible window of the time point and not dispatch the next executable goal; however once past the window the Goal Executive will fire the time point and issue the next executable goal even if it is not ready for transition. Thus a temporal problem in execution will be manifested as a potential goal failure by the goal that was not ready to transition.

Consequences

Executive Control provides for the sequencing of activities on individual state variable time lines and the coordination of events across all state variables modeled within the system. As an independent functional entity, the Goal Executive may continue to execute the latest mission plan while other planning activities occur. It provides an intermediate rate of execution between potentially long-term planning activities and rapid execution cycles of a reactive control system. As such, care must be exercised when choosing a rate of execution for the Goal Executive.

3.8 Goal Monitoring and Fault Response

Purpose

Define a pattern for monitoring the execution of goals in order to respond to goals that cannot be achieved (goal failures).

Motivation

Time continues moving forward regardless of what happens in the system. Although a reactive control system, with the knowledge of intent available in a goal, may be able to compensate for some unexpected events, things can still fail. Since the current goal network was planned using a specific set of tactics to achieve certain goals, there may be other goal networks (using alternate tactics) that could still achieve the plan's intent.

For example, consider a goal to drive a mobile robot from point A to point B through city streets. The set of available routes is constrained, and a given plan may choose one route. However, after executing part of the route, an obstacle is encountered, preventing further advance along

that route. Now the only option is to give up the current plan, and try another route.

Since the current plan may also contain goals that are still relevant, the executive and the goal achievers cannot just stop – they must continue trying to achieve the current plan until a new plan can be produced. So, a separate mechanism is required to notice that the plan is failing, and notify the planner to do something about it.

Structure

The *Goal Monitor* is a separate element of the control system that monitors the status of all currently executing goals. The Goal Monitor consults each executable goal's associated state variable to check the estimated state against the intent of the goal. The Goal Monitor may also check temporal constraints and projections to determine if a goal can still be satisfied. If the state variable reports that a given merged executable goal is no longer satisfiable, the Goal Monitor will then initiate a fault response.

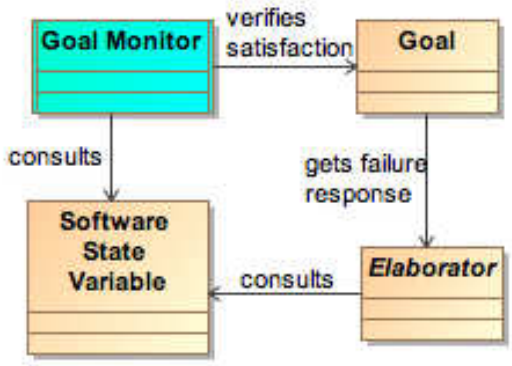

Figure 11 – Goal Monitor

First, it will attempt to determine which of the contributing goals merged into the failing executable goal have failed. To do so, it will query each of the contributing goals to see if it is still satisfiable. For each failing goal it then finds that goal's parent goal (using relations in the goal network), and notifies the parent goal's elaborator, which in turn determines an appropriate fault response.

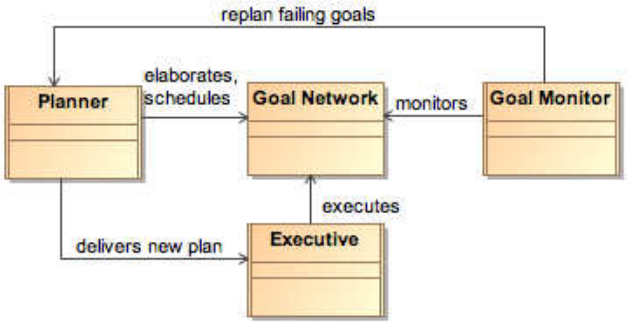

Figure 12 – Fault Response

The parent goal's elaborator has several options. It can decide to do nothing (i.e., just let the plan continue to execute and hope for the best); it can assert an error condition that would stop and safe the system; it can propose a change of plan by invoking re-elaboration of a different tactic; or, it can "fail up" by consulting its parent for a fault response. The process of failing up the goal elaboration hierarchy allows a fault to propagate up to the level of intent at which it can be appropriately dealt with.

Consequences

Separating the goal monitor from the executive allows the executive to continue trying to achieve the current plan as best it can. Separating the goal monitor from the planner/scheduler allows the monitor to continue checking the status of goals even after a fault is detected and a response initiated. If a second fault occurs, the monitor and planner/scheduler can then prioritize their response based on relationships between the failing goals. For example, if several goals are all failing at the same time (a likely situation if their state variables affect one another), then the goal monitor, or planner can determine that they are all children of the same parent goal, and then only have to replan that one parent goal. Or, it can determine that the goals are entirely independent, and re-elaborate and reschedule them separately.

3.9 Deliberative Closed-Loop Control

Purpose

Reactive control is very useful for many situations when control decisions can be made without looking far into the future. However, sometimes the determination of what should be accomplished in the present depends on what is planned or predicted for the future. Because reactive control systems have no knowledge of future plans beyond the activity they are currently trying to accomplish, there is a need for a mechanism to control systems that must consider the future. The deliberative closed-loop control pattern provides such a mechanism.

This mechanism constructs, monitors, and revises goal networks that take into consideration requirements on what needs to be accomplished in the future. The deliberative closed-loop control pattern monitor function responds to unpredictable or unanticipated events as they occur during execution.

Motivation

Consider the problem of maintaining a battery state of charge through a series of activities that both consume and produce energy. One can represent the requirement to maintain the battery state of charge above a minimum limit as a goal on a battery energy state variable. The activities, represented as a series of goals, need to be ordered in time into a plan such that the battery state of charge does not fall below the minimum limit. The goals that affect the battery

state of charge in the plan are used to predict the battery state of charge, and validate that the plan does not violate the minimum battery state of charge limit. The goal to maintain the state of charge can be monitored during execution, and activities can be shed if the use an unexpected amount of power.

Applicability

This pattern is applicable to situations in which:

(1) A large number of state variables need to be controlled in parallel;

(2) The control strategy involves a series of activities organized into a long term plan;

(3) The activities can be expressed as goals on state variables;

(4) The state variables must be controlled to meet user-defined goals; and

(5) The plan needs to be able to be changed automatically in response to unanticipated or unpredictable events.

Structure

This pattern is a composition the following patterns described previously:

(1) Goal Network

(2) Goal Elaboration

(3) Goal Planning and Scheduling

(4) Executive Control (Timeline Execution)

(5) Goal Monitoring and Fault Response

Construction of a goal network includes the elaboration of operator-specified goals, scheduling the resulting goal network, and validating the result as an executable goal network. The executable goal network is executed by the goal executive, and as each executable goal executes it is monitored by the goal monitor. The goal monitor notifies the planner when an executable goal fails, allowing the planner to modify the plan to respond to goal failures.

Combining these patterns enables the kinds of complex behaviors made possible by traditional sequencing and fault management mechanisms, but in addition, it accommodates dynamic changes to the plan. Specifically, it provides a coordinated mechanism for responding to faults or other unexpected deviations from the plan.

Consequences

A key advantage of the deliberative closed loop control pattern is that problems can be detected before they happen by checking predictions for executable goals. Corrective action can be taken before serious consequences ensue. For example, if battery energy is being used faster than predicted, the goal network may be revised to shed lower priority energy-consuming goals. Or it may schedule new goals to charge the battery.

The deliberative closed loop control pattern may require significant computing resources and time for performing scheduling. This can be ameliorated by ensuring that goal networks are scheduled for a limited time horizon, avoiding the computational expense of long-term planning. Also, pre-scheduled networks can be quickly swapped in if a fast response is required. An example may be a “safe-mode goal network” that puts the system into a safe state.

This pattern needs good models of physics and achiever behavior to validate scheduled goal networks. However, models only need to be as good as necessary to achieve objectives. Many times conservative simple models are

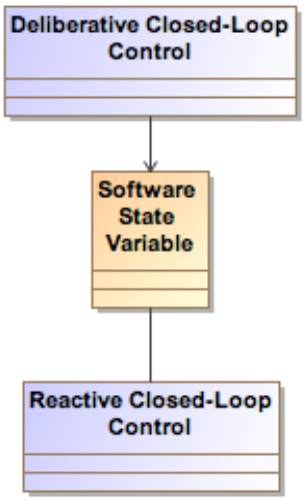

Figure 13 – Deliberative and reactive closed-loop control patterns are connected through software state variables.

adequate.

3.10 Deliberative and Reactive Closed-Loop Control

Purpose

Reactive and deliberative closed-loop control patterns are combined into a single pattern to allow for highly flexible and robust control system behavior.

Motivation

Control systems may need to be both reactive to small changes in the system under control, as well as being able to

plan and execute a long-range series of tasks. For example, a Mars rover needs to be able to deliberatively plan a safe path across rocky terrain and also reactively control its wheel rotations to accommodate slippage while maintaining forward progress.

Applicability

This pattern is applicable to most embedded and robotic control systems, which require both deliberative and reactive control.

Structure

This pattern is a composition the following patterns described previously:

(1) Reactive Closed-Loop Control

(2) Deliberative Closed-Loop Control

These two patterns are connected through software state variables, as shown in Figure 13. State variables are estimated and controlled by the reactive control system in response to executable goals metered out by the deliberative closed loop control system. The deliberative control system sequences and validates the plans for goal execution, and detects goal execution failures as the reactive control system acts on the goals. The deliberative control system responds to goal failures through goal re-elaboration and scheduling to produce a modified executable network. Figure 14 shows major data flows within this combined control pattern.

Consequences

The integration of deliberative and reactive control brings some complexity in terms of interactions between the two patterns, but this complexity is largely inherent in the challenging control problems for which it is applicable. The intent of specifying this architectural pattern is to provide a structured means of dealing with this complexity.

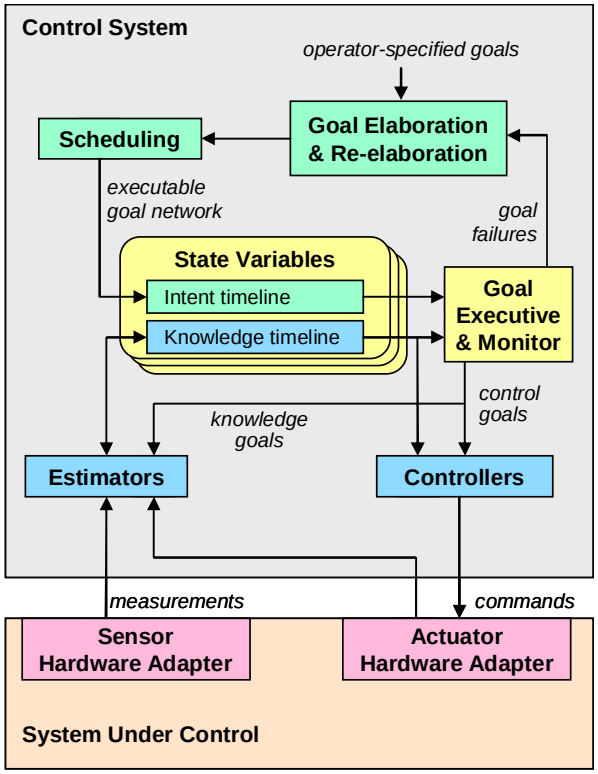

Figure 14 – Major data flows within combined architectural pattern for goal-based control

4. RELATED WORK

The idea of operating systems at the level of explicit intent is not a novel concept. For example, thermostats have been used to control the temperature of building interiors for over a hundred years. The thermostat's set point is a form of goal in that it specifies the desired temperature that the building's heating, ventilating, and air conditioning control system must achieve and maintain. In the context of space exploration, goals have actually been used for decades in limited fashion, particularly in the context of spacecraft attitude and articulation control systems, for the purposes of pointing science instruments, communication antennae, solar panels, etc. Vehicle reorientation and gimbal angles are "commanded" by specifying trajectories of desired angles and rotation rates; these state trajectories are explicit representations of intent. Until recently, however, such representations have not been used consistently across all spacecraft subsystems, and have not been integrated into coherent system-level control architectures and operations processes. This section provides a brief overview of related work in goal-based control architectures and goal-based operations (GBO), highlighting a number of significant achievements from the space exploration domain.

One of the first full-scale (system-level) applications of goal-based control architecture was the Remote Agent (RA) Experiment [3], which was flight-validated in 1999 on the Deep Space One (DS1) spacecraft, the first deep space mission in NASA's New Millennium Program. RA is a model-based, reusable, artificial intelligence (AI) software system that enables goal-based spacecraft commanding and robust fault recovery. A simplified view of the RA software architecture is shown in Figure 15. RA consists of general-purpose reasoning engines (both deductive and procedural) and mission-specific domain models. One of its key characteristics—and a main difference with traditional spacecraft commanding—is that ground operators can communicate with RA using goals (e.g., "During the next week take pictures of the following asteroids and thrust 90% of the time") rather than with detailed sequences of timed commands. RA determines a plan of action that achieves those goals; actions are represented as tasks that are decomposed on-the-fly into more detailed tasks and, eventually, into commands to the underlying flight software. The RA Experiment provided an invaluable proof-of-concept and lessons learned in a number of areas, including benefits and challenges associated with autonomous goal-based operations. These lessons have been documented in the Remote Agent Experiment DS1 Technology Validation Report [4].

NASA's Mars Exploration Rovers (MER) [5], Spirit and Opportunity, also employ a certain degree of goal-based operations capability, in both the ground system and onboard the rovers. In the ground system, operators use the Mixed-initiative Activity Plan GENerator (MAPGEN [6]) tool to plan each rover's science and engineering activities on a sol-by-sol basis. Given a set of user observation goals and their priorities, this tool enables operators to construct a plan that satisfies these goals and schedule the activities in the plan such that conflicts between incompatible activities and oversubscription of limited resources are avoided. MAPGEN leverages the automated planning and scheduling engine that was flight-validated as part of RA on DS-1, integrating it into a GUI environment that enables operators to incrementally build and edit their plans. With this tool, a plan is refined through iterations of automated computation and judicious hand-editing based on domain expertise, eventually converging to a final plan that the operator finds appropriate. Onboard each rover, the flight software is programmed to accept a combination of abstract goal-like directives, such as 'drive to waypoint', and lower-level commands. In a remote and unpredictable environment like the Martian surface, the rovers robustly achieve their ambitious science objectives by taking advantage of their ability to make certain decisions in-situ, and execute flexibly-specified plans in an event-driven fashion. These are fundamental characteristics of goal-based systems.

The most recent and comprehensive space-based application of a goal-based control architecture is the Autonomous Sciencecraft Experiment (ASE) [7, 8]. ASE is a software system currently flying onboard the EO-1 spacecraft, which has demonstrated several integrated autonomy technologies that together enable science-directed autonomous operations. The ASE software includes onboard continuous planning, robust task and goal-based execution, and onboard machine learning and pattern recognition, and has more recently been augmented to demonstrate model-based diagnosis capabilities with RA heritage. Like RA, ASE began as a technology experiment within NASA's New Millennium Program, as part of the Space Technology 6 project. Early tests had the goal-directed planning and execution capabilities deployed as part of a ground-based

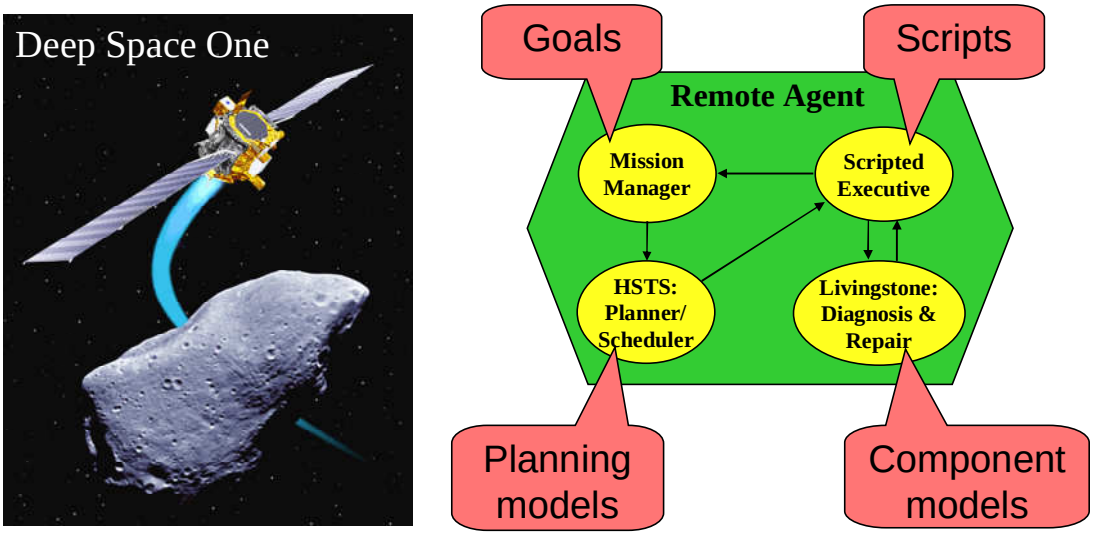

Figure 15 – Remote Agent Experiment on Deep Space One demonstrated goal-based operations in 1999. In this architecture the Mission Manager sends high-level goals to the Planner/Scheduler which then generates detailed tasks (lower-level goals), and the Executive executes scripts associated with the lower-level goals, issuing commands as needed.

sequencing system; the success of these tests built up confidence in the technology in preparation for ultimate deployment of the capabilities onboard the spacecraft. The technology was declared fully validated in May 2004. The ASE software now runs full-time onboard the EO-1 satellite, and has become its primary mission planning and control system. Through automation of the operations process, ASE has contributed operational savings of approximately $1M per year, compared to EO-1's nominal operations cost prior to ASE deployment. It has resulted in dramatic increases in science return, thanks to its intelligent downlink selection and autonomous retargeting capabilities, and increased flexibility in operations, thanks to the resulting streamlining of human-operator-in-the-loop activities. Another long-term benefit of the ASE project is documentation of the lessons learned which will certainly be invaluable to future applications of onboard autonomy and goal-based operations.

Not surprisingly, NASA is not alone in its desire to exploit the benefits of goal-based autonomous control architectures. In 2001, the European Space Agency (ESA) launched its first Project for On-Board Autonomy [9] (PROBA-1) spacecraft. The PROBA-1 technology validation mission successfully demonstrated both onboard and ground-based automation, including the ability to convey high-level goals (user requests) to the spacecraft via the Internet. ESA is also investigating the use of goal-based control and on-board planning and scheduling for ExoMars [10], a Mars rover anticipated to be the first flagship mission in ESA's Aurora Exploration Programme.

Although this paper's focus is on spacecraft applications, the goal-based control approach has broad applicability to other domains, such as industrial robot control and autonomous unmanned air/underwater/ground vehicles. For example, the Defense Advanced Research Projects Agency (DARPA) has sponsored Grand Challenges, which have stimulated the development of various goal-directed planning and execution techniques and technologies. More broadly, goal-based control architecture is the focus of much research and development in academic, governmental and industrial organizations.

5. SUMMARY

The patterns for goal-based control provide the general organizing principles that allow intent to be preserved through the planning, execution, and fault response phases of system operation. These are only the general patterns, and real control systems present many special circumstances and situations that call for specializations or adaptations of these patterns. Reference [12] describe several pattern specializations for more complex estimation, or state representation patterns (distillation, graph state variables, value histories); patterns for dividing control systems across physical boundaries (proxy state variables and hardware adapters); patterns for managing data (data state variables, and data controllers); and patterns for smoothly transitioning from the execution of one plan to the next (promotion).

6. FUTURE WORK

Additional architectural patterns have been developed for the following capabilities, and could be described in one or more follow-on publications:

- *Delegation* is a pattern that enables one achiever to send goals directly to another achiever that enables a goal-based version of reactive control;
- *Measurement Distillation* is a pattern that converts measurements into measurements that retain only the essential information required for state estimation;
- *State Variable Timelines* are an abstraction of state variable representations for three kinds of state variable information: knowledge, intent, and projection;
- *Proxy State Variable* and *Proxy Hardware Adapter* are copies of state variable information and command and measurement histories available to a deployment that is remote from the deployment in which they were created,
- *Data State Variables*, *Data Controllers*, and *Data Commands* are special representations and control mechanisms to control the content and transport of value histories using the reactive and deliberative control mechanisms described in this paper,
- *Promotion* is the mechanism by which a scheduled and validated executable goal net is installed and placed into execution by the goal executive.

Current work is investigating the interfaces between the control system and its human users in an attempt to improve the ways people interact with control systems. Many of the active roles described in these patterns can be, and traditionally are, performed by people. Using the pattern interfaces may allow for a more seamless interaction between systems and their users.

Although these patterns have been applied in large distributed systems, they present some interesting questions in a systems-of-systems context. It is possible to use these patterns recursively, so that systems could distribute goals to subsystems, where the subsystems perform their own tactical planning and execution.

ACKNOWLEDGEMENT

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. Thanks to Nicolas Rouquette for help with the UML diagrams.

REFERENCES

[1] Constellation Program Web site, National Aeronautics and Space Administration, http://www.nasa.gov/mission_pages/exploration/main/index.html.

[2] 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, Vol. 2, No. 12, Dec. 2005, pp. 507-536.

[3] Bernard, D., et al., "Design of the Remote Agent Experiment for Spacecraft Autonomy," Proceedings of the IEEE Aerospace Conference, Aspen, CO, 1999.

[4] Bernard, D., et al., "Final Report on the Remote Agent Experiment", Proceedings of the New Millennium Program DS-1 Technology Validation Symposium, Pasadena, CA, February 2000.

[5] 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.

[6] Ai-Chang, M., et al., "MAPGEN: Mixed Initiative Activity Planning for the Mars Exploration Rover Mission", Proceedings of the 13th International Conference on Planning & Scheduling (ICAPS '03), Trento, Italy, June 2003.

[7] Chien, S. et al., "Using Autonomy Flight Software to Improve Science Return on Earth Observing One", AIAA Journal of Aerospace Computing, Information and Communication, Vol. 2, No. 4, April 2005, pp. 196-216.

[8] Chien, S., et al., "Lessons Learned from Autonomous Sciencecraft Experiment", Proceedings of the Autonomous Agents and Multi-Agent Systems Conference (AAMAS 2005). Utrecht, Netherlands, July 2005.

[9] Proba: Observing the Earth Web site, European Space Agency, http://www.esa.int/esaMI/Proba_web_site/index.html.

[10] Woods, M., et al., "On-board Planning and Scheduling for the ExoMars Mission", Proceeding of the DASIA (DAta Systems In Aerospace) Conference, Berlin, Germany, 22-25 May 2006.

[11] Gamma, E., et al, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, 1995.

[12] Bennett, M., Dvorak, D., Ingham, M., Morris, J.R., Rasmussen, R., and Wagner, D., "State Analysis for Software Engineers Training Course", https://pub-lib.jpl.nasa.gov/docushare/dsweb/View/Collection-68.

BIOGRAPHY

Matthew Bennett *is a Senior Software Systems Engineer in the Flight Software & Data Systems section at the Jet Propulsion Laboratory. He has interests in model-based engineering, software architecture, and spacecraft autonomy. He has developed mission software for fault protection, guidance and control, science data collection, performance analysis, and simulation. He holds an MS from the University of Washington in Computer Science, and a BS from the University of California at San Diego in Computer Engineering.*

Daniel Dvorak *is a principal engineer in the Planning & Execution Systems section at the Jet Propulsion Laboratory. His research interests include software architecture, model-based engineering, and operation of autonomous systems. Prior to 1996 he worked at Bell Laboratories on the monitoring of telephone switching systems and on the design and development of R++, a rule-based extension to C++. Dan holds a Ph.D. in computer science from The University of Texas at Austin, an MS in computer engineering from Stanford University, and a BS in electrical engineering from Rose-Hulman Institute of Technology.*

Joseph Hutcherson *is a senior software engineer in the distributed systems technologies group at the Jet Propulsion Laboratory. He has helped develop information distribution systems for the Navy and Marine Corps using R/F and satellite communications. Most recently he has helped to develop a Java – based version of MDS. He has a BS in General Engineering from Harvey Mudd College.*

Michel Ingham *is a senior software system engineer in the Flight Software Systems Engineering and Architecture Group at the Jet Propulsion Laboratory. His research interests include model-based methods for systems and software engineering, software architectures, and spacecraft autonomy. He earned his Sc.D. and S.M. degrees from MIT in Aeronautics and Astronautics, and a B.Eng. in Honours Mechanical Engineering from McGill University in Montreal, Canada.*

Robert Rasmussen *has been a systems engineer at JPL since 1975 after receiving his Ph.D in Electrical Engineering from Iowa State University. He has contributed broadly to several planetary missions, including Voyager and Galileo, and was the lead engineer for the Cassini Attitude and Articulation Control Subsystem. Bob has long been interested in spacecraft control and autonomy, helping to initiate the award winning Remote Agent experiment on DS-1. He has also been Technologist for the Information Technologies and Software Systems Division, and Architect for the Mission Data System project, developing a unifying architecture and model-based engineering methodology for complex autonomous systems. He is presently a JPL Engineering Fellow and Chief Engineer for the Systems and Software Division.*

David Wagner *is a senior software system engineer in the flight software applications group at the Jet Propulsion Laboratory. He has a BS in Aerospace Engineering from the University of Cincinnati, and a MS in Aerospace Engineering from the University of Southern California.*