JOURNAL OF AEROSPACE COMPUTING, INFORMATION, AND COMMUNICATION
Vol. 2, December 2005

Engineering Complex Embedded Systems with State Analysis and the Mission Data System

Michel D. Ingham,[*] Robert D. Rasmussen,[†] Matthew B. Bennett,[‡] and Alex C. Moncada[§]
Jet Propulsion Laboratory, California Institute of Technology, 4800 Oak Grove Dr., Pasadena, CA, 91109, USA
{michel.d.ingham, robert.d.rasmussen, matthew.b.bennett, alex.c.moncada}@jpl.nasa.gov

It has become clear that spacecraft system complexity is reaching a threshold where customary methods of control are no longer affordable or sufficiently reliable. At the heart of this problem are the conventional approaches to systems and software engineering based on subsystem-level functional decomposition, which fail to scale in the tangled web of interactions typically encountered in complex spacecraft designs. Furthermore, there is a fundamental gap between the requirements on software specified by systems engineers and the implementation of these requirements by software engineers. Software engineers must perform the translation of requirements into software code, hoping to accurately capture the systems engineer's understanding of the system behavior, which is not always explicitly specified. This gap opens up the possibility for misinterpretation of the systems engineer's intent, potentially leading to software errors. This problem is addressed by a systems engineering methodology called State Analysis, which provides a process for capturing system and software requirements in the form of explicit models. This paper describes how requirements for complex aerospace systems can be developed using State Analysis and how these requirements inform the design of the system software, using representative spacecraft examples.

I. Introduction

AS the challenges of space missions have grown over time, we have seen a steady trend toward greater automation, with a growing portion assumed by the spacecraft. This trend is accelerating rapidly, spurred by mounting complexity in mission objectives and the systems required to achieve them. In fact, the advent of truly self-directed space robots is not just an imminent possibility, but an economic necessity, if we are to continue our progress into space.

What is clear now, however, is that system complexity is reaching a threshold where customary methods of control are no longer affordable or sufficiently reliable. At the heart of this problem are the conventional approaches to systems and software engineering based on subsystem-level functional decomposition, which fail to scale in the tangled web of interactions typically encountered in complex spacecraft designs. A straightforward extrapolation of past methods has neither the conceptual reach nor the analytical depth to address the challenges associated with future space exploration objectives.

Furthermore, there is a fundamental gap between the requirements on software specified by systems engineers and the implementation of these requirements by software engineers. Software engineers must perform the translation of requirements into software code, hoping to accurately capture the systems engineer's understanding of the system

Received 17 January 2005; revision received 17 January 2005; accepted for publication 29 September 2005.

[*] Senior Software Systems Engineer, Flight Software Systems Engineering and Architecture Group, M/S 301-225, AIAA Member.
[†] Principal Engineer, Systems and Software Division, M/S 301-225.
[‡] Senior Software Systems Engineer, Flight Software Systems Engineering and Architecture Group, M/S 156-142.
[§] Systems Engineer, Flight Software Systems Engineering and Architecture Group, M/S 301-225.

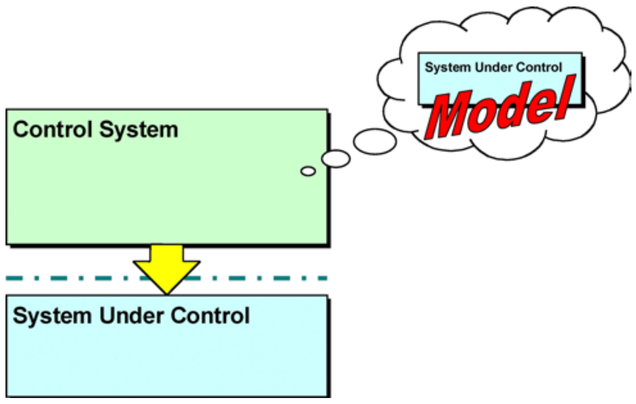


Fig. 1 The control system uses a model of the system under control.

behavior, which is not always explicitly specified. This gap opens up the possibility for misinterpretation of the systems engineer's intent, potentially leading to software errors.

In this paper, we describe a novel systems engineering methodology, called *State Analysis*, which addresses these challenges by asserting the following basic principles:

– Control subsumes all aspects of system operation. It can be understood and exercised intelligently only through models of the system under control. Therefore, a clear distinction must be made between the *control system* and the *system under control* (see Fig. 1).
– Models of the system under control must be explicitly identified and used in a way that assures consensus among systems engineers.
– Understanding state is fundamental to successful modeling. Everything we need to know and everything we want to do can be expressed in terms of the state of the system under control.
– As complexity grows, the line between specifying behavior and designing behavior is blurring. To the extent the software design reflects the systems engineer's understanding, the software will perform as the systems engineers desire. Thus, the manner in which models inform software design and operation should be direct, requiring minimal translation.

State Analysis improves on the current state-of-the-practice by producing requirements on system and software design in the form of explicit models of system behavior, and by defining a state-based architecture for the control system. It provides a common language for systems and software engineers to communicate, and thus bridges the traditional gap between software requirements and software implementation.

In this paper, we discuss the state-based control architecture that provides the framework for State Analysis (Section II), we emphasize the central notion of state, which lies at the core of the architecture (Section III), we present the process of capturing requirements on the system under control in the form of models (Section IV), and we illustrate how these models are used in the design of a control system (Section V). We then describe a State Database tool used for documenting the models and requirements (Section VI). Finally, we describe the Mission Data System (MDS), a modular multi-mission software framework that leverages the State Analysis methodology (Section VII). Clearly, a complete discussion of these topics is beyond the scope of a paper such as this; however, the overview we provide here highlights the essential features of State Analysis and MDS.

II. State-based Control Architecture

State Analysis provides a uniform, methodical, and rigorous approach for:

– discovering, characterizing, representing, and documenting the states of a system;
– modeling the behavior of states and relationships among them, including information about hardware interfaces and operation;
– capturing the mission objectives in detailed scenarios motivated by operator intent;

- keeping track of system constraints and operating rules; and
- describing the methods by which objectives will be achieved.

For each of these design aspects, there is a simple but strict structure within which it is defined: the state-based control architecture (also known as the “control diamond,” shown in Fig. 2). The architecture has the following key features:[1]

- *State is explicit:* The full knowledge of the state of the system under control is represented in a collection of state variables. State knowledge is updated in the form of continuous-time State Functions, to accurately reflect the fact that the system’s true state is defined at any point in time. We discuss the representation of state in more detail in Section III.
- *State estimation is separate from state control:* Estimation and control are coupled only through state variables. State estimation is a process of interpreting measurements and monitored commands to generate state knowledge; this process may combine multiple sources of evidence into a determination of state, supplied to a state variable as an estimate. Control, in contrast, attempts to achieve objectives by issuing commands that should drive estimated state toward desired state. Keeping these two tasks separate promotes objective assessment of system state, ensures consistent use of state across the system, simplifies the design, promotes modularity, and facilitates implementation in software.
- *Hardware adapters and data collections provide the sole interfaces between the system under control and the control system:* They form the boundary of our state architecture. Hardware adapters provide all the measurement and command abstractions used for control and estimation of physical states, and are responsible for translating and managing raw hardware input and output. Measurements can be used both as evidence for estimating the state of the hardware in the system under control (e.g., accelerometer, switch position, and temperature sensor measurements), and for holding science observations (e.g., camera images and spectrometer readings). The control system can directly inspect the data collections to determine the state of

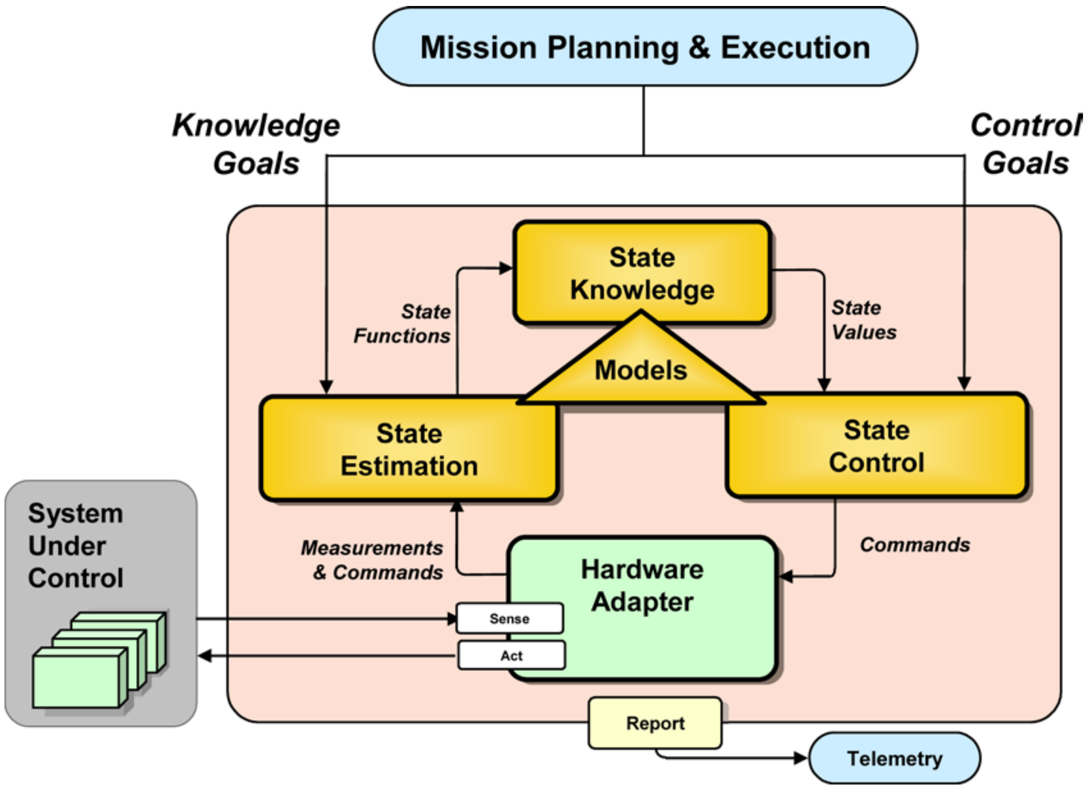


Fig. 2 The state-based control architecture.

the system data. Commands are directives that change the state of the system under control; these can be hardware commands (e.g., switch open/close and device operating mode commands) or data commands that are used for managing the data collections (e.g., data compression and data transport commands).

- *Models are ubiquitous throughout the architecture:* Models are used both for execution (estimating and controlling state) and higher-level planning (e.g., resource management). Whether overt and explicit, or hidden quietly in the minds of the engineers, models have always existed, since understanding and modeling are essentially the same thing. The key is that State Analysis requires that the models be documented explicitly, in whatever form is most convenient for the given application. In Section IV, we describe our process for capturing these models.
- *The architecture emphasizes goal-directed closed-loop operation:* Instead of specifying desired behavior in terms of low-level open-loop commands, State Analysis uses *goals*, which are constraints on state variables over a time interval. Goals are easier to specify than the actions needed to achieve them, and result in more compact specifications of desired behavior. Furthermore, goal-directed operation goes hand-in-hand with closed-loop control, because goals can be thought of as set points for onboard controllers, which are then given the latitude to decide how best to achieve the goals. In our architecture, goals are also used to specify the desired quality of state knowledge to be achieved by estimators, and to express operating constraints (such as resource and safety margins) and monitored conditions (such as failure modes and external events). In Section V, we discuss goals and their use in high-level system coordination.
- *The architecture provides a straightforward mapping into software:* The control diamond elements can be mapped directly into components in a modular software architecture, such as MDS,[1] which is described in Section VII of this paper.

In summary, the State Analysis methodology is based on a control architecture that has the notion of state at its core. In the following section, we describe our representation of state, and how we capture the evolution of state knowledge over time.

III. State Knowledge Representation

As discussed in the previous section, State Analysis is founded upon a state-based control architecture, where state is the momentary condition of an evolving system and models describe how state evolves. The state of a system and our knowledge of that state are not the same thing. The real state may be arbitrarily complex, but our knowledge of it is generally captured in simpler abstractions that we find useful and sufficient to characterize the system state for our purposes. We call these abstractions "state variables." The known state of a system is the value of its state variables at the time of interest.

Together, state and models supply what is needed to operate a system, predict future state, control toward a desired state, and assess performance. In this section, we focus on clarifying what we mean by "state," and describing how we represent state in state variables. More detail on our representation of state knowledge has been previously published.[2,3]

A. Defining "State"

A control system has cognizance over the system under control. This means that the control system is aware of the state of the system under control, and it has a model of how the system under control behaves. The premise of State Analysis is that everything we care about (to meet mission objectives) can be completely characterized as knowledge of state and its behavior—that no other information is required to control the system. Consequently, State Analysis adopts a broader definition of state than traditional closed-loop control theory, for example: in addition to position, attitude, temperature, pressure, power, and the like, we would also include as state any other aspects of the system that we care about for the purposes of control, and that might need to be estimated, such as device operating modes, device health, resource levels (propellant; volatile and non-volatile memory), etc. We would also include environmental states such as the motions of celestial bodies and solar flux.

Though we often think of state as describing the features of a system that change over time, we must include in our definition of state certain quantities that are actually static in nature. For example, the dry mass of a spacecraft may be essentially constant; however, our knowledge of it may very well change over the course of a mission. If one is actively engaged over the lifetime of a system in refining knowledge of some attribute of a system, then that attribute is best described with a state variable—not because the actual value is changing, but rather because what we do to estimate its value or respond to changes in its estimated value is fundamentally no different than for any other state variable. Similarly, our definition of state encompasses attributes that we would normally define as "parameters" in the vernacular of space systems, such as instrument scale factors and biases, structural alignments, and sensor noise levels. Their variation over time may be slow; in fact, they may even be treated as constant under normal circumstances. But changes happen to them, nonetheless—even if it requires a failure to cause the changes—and a control system must be cognizant of such changes to the system.

Another type of state variable of great importance is nevertheless likely to come as a mild surprise to those who have not yet encountered it: the state of a data collection. Lower level data management and transport functions are generally considered to be part of the system under control. These include the cataloguing functions that capture the collections of data gathered from instruments and other devices. Therefore, in its cognizance of the system, the control system must also be aware of the state of these data collections. That is, there must be state variables that represent the state of these collections. On further consideration, this should come as no surprise at all. The concept of state is concerned with change. The difference between a camera before taking a picture and that same camera afterward is primarily the presence of the newly recorded image. This is not only a change we can describe, but it is a vital change—one that motivates the entire existence of the camera. And it is not just any image we care about, but the right image. Therefore, the change of state must be described in a manner sufficient to discriminate between the right image and the wrong image. The state variable we need, then, is one that represents our knowledge of what data has been collected, including the conditions under which it was collected, the subject of the data, or any other information pertinent to decisions about its treatment.

We note, however, that the internal state of the control system is not represented by state variables. This is in keeping with a basic principle of State Analysis that distinguishes clearly between the control system and the system under control (recall Section I). A control system may indeed have internal state; in fact, it usually does. These might include control modes, records of past operation, and so on. But this state is not maintained in state variables. A state variable is the representation of a state; it is knowledge of that state, but it is not the state itself. Therefore, if control system state were maintained in state variables, there would be both control system states and control system state variables to represent them, which is redundant. This would lead to problematic self-reference, with the control system issuing goals on its own internal state variables.

B. Representing State

Now that we have defined what "state" means, we consider how to represent it. An important part of the State Analysis process is to select and document an appropriate representation for each state variable in the system. As previously indicated, state variables can have discrete values (e.g., a camera's operational mode can be "off," "initializing," "idle," or "taking-picture") or continuous values (e.g., a camera's temperature might be represented as a real value in degrees Celsius). Whether continuous- or discrete-valued (or a composite of such values), all state variables represent state as a piecewise function of continuous time, rather than as a history of time-stamped samples. This representation is true to the underlying physics, where state is defined at every instant in time. Our architectural decision to update state in the form of continuous-time State Functions (see Fig. 2) has important implications on the form of the software requirements produced through State Analysis. It is therefore worthwhile to introduce the notion of *state timelines* as the conceptual repositories for state knowledge, which also map into state value containers in the MDS software architecture.

State Analysis assumes that state evolution is described on state timelines (Fig. 3), which are a representation of a system's past and future history. This representation is complete, to the extent that it captures everything the control system has chosen to remember about the state, subject to storage limitations. State knowledge is "forgotten" by explicitly replacing it with knowledge of lesser quality (e.g., a summarized representation, or simply the value "unknown"), rather than deleting it. As a result, querying a state timeline at any time will always return a stored value, even if that value may be merely "unknown." State timelines provide the fundamental coordinating mechanism for any control system developed using State Analysis, since they describe both knowledge and intent. This information, together with models of state behavior, provides everything the control system needs to predict and plan, and it is available in an internally consistent form, via state variables. This internal consistency is due to architectural rules disallowing competing local versions of information about the same state.

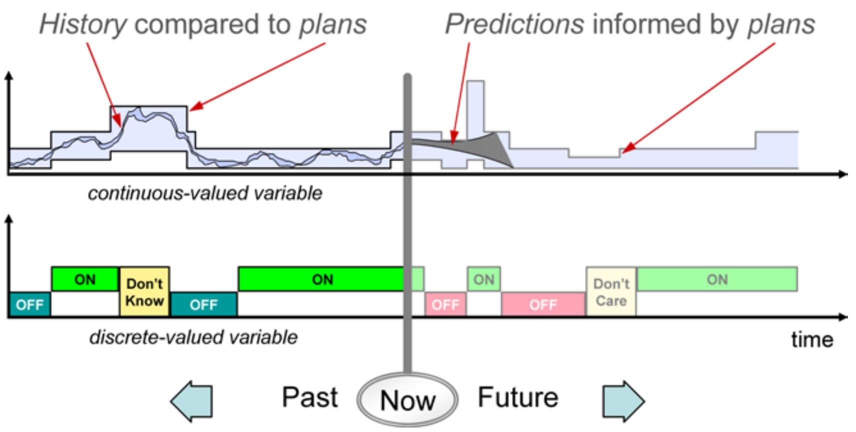


Fig. 3 Timelines are used to capture state knowledge (past estimates and future predictions) and intent (past and future constraints on state).

State timelines also provide a control system with an efficient mechanism for transporting data between the ground system and the spacecraft. For instance, telemetry can be accomplished by relaying state histories to the ground, and communication schedules can be relayed as state histories to the spacecraft. Timelines are a relatively compact representation of state history, because states evolve only in particular and generally predictable ways. That is, they can be modeled. Therefore, timelines can be transported much more compactly than conventional time-sampled data.

Because of our adoption of a continuous-time representation of state in the form of State Functions on a timeline, a state and all of its derivatives can and should be modeled using a single state variable, to ensure consistency of representation (thus avoiding the possibility of returning inconsistent values for a state and its derivative).

C. Representing Uncertainty

In a real system, we never really know physical states with complete accuracy or certainty—only a simulator "knows" state values precisely. The best we can do is to estimate the value of the state as it evolves over time.

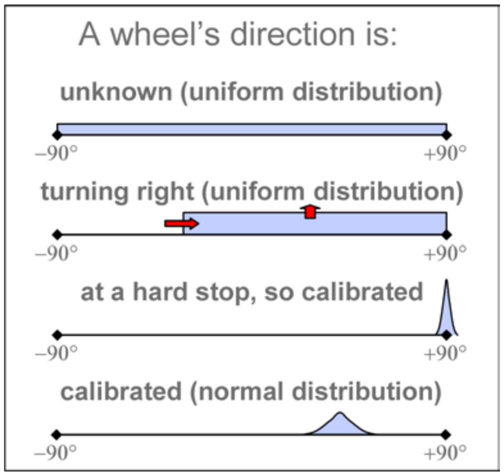

A wheel's direction is:
unknown (uniform distribution)
–90°
+90°
turning right (uniform distribution)
–90°
+90°
at a hard stop, so calibrated
–90°
+90°
calibrated (normal distribution)
–90°
+90°

Fig. 4 Uncertainty representation can be polymorphic, e.g., given a uniformly uncertain rover wheel direction, we can calibrate our knowledge of direction by turning the wheel (narrowing the range of the uniform distribution) until it hits its stop (at which point our knowledge of state has very little uncertainty). As a result of rover operations, the level of uncertainty in the wheel direction estimate increases over time (represented as a normal distribution with gradually increasing variance).

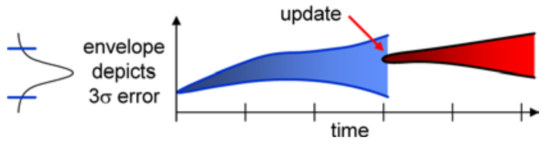


Fig. 5 The level of uncertainty associated with a state estimate generally grows over time, and can decrease with the receipt of additional evidence by the estimator.

These estimates constitute state knowledge; it is what we know, and, equally important, how well we know it. That is, it makes no sense to represent the estimated value of a state without also representing the level of certainty of the estimate. Although State Analysis asserts that uncertainty must be explicitly represented along with the state value, it imposes no restriction on how uncertainty should be represented. It can be represented in many ways, e.g., enumerated confidence tags, variance in a Gaussian estimate, probability mass distribution over discrete states, etc. State Analysis even allows for polymorphic representations of state and uncertainty (see Fig. 4).

There are multiple benefits to explicitly representing uncertainty. First, it leads to a more robust software design, in which estimators can be honest about the evidence, increasing the uncertainty in their estimates for conflicting evidence, noisy evidence, missing evidence, and 'old' evidence (see Fig. 5). Furthermore, it enables controllers to exercise caution, and modify their actions during periods of high uncertainty. Finally, it allows human operators to be better informed about the quality of knowledge of the state.

We recall that data collections are included as part of the system under control. We note that the state of data collections is known with complete certainty, since the state of system data can be directly inspected. Thus, data states are represented without uncertainty.

Now that we have defined our notion of state and described our representation of it, we next turn to the issue of modeling the behavior of the system under control.

IV. Modeling the System Under Control

State Analysis provides a methodology for developing a model of the system under control. This model represents everything we need to know for controlling and estimating the state of the system under control. We note that traditional systems engineering approaches capture most of this information in multiple disparate artifacts, allowing for potential inconsistencies. By making the model explicit, the State Analysis approach consolidates all this information rigorously in a consistent unambiguous form.

Our model of the system under control is composed of:

– *State Models* describing how each state in the system under control evolves over time and under the influence of other states;
– *Measurement Models* describing how each measurement is affected by various states in the system under control; and
– *Command Models* describing how states are affected by each command (possibly under the influence of other states in the system under control).

This model describes the behavior of all elements in the system under control, including most of the hardware plus any software assigned to the system under control (e.g., in hardware adapters or data management functions), as well as the relevant behavior of any external systems (e.g., environmental effects). It is important to note that these models are expressed in terms of *true state*, and that consideration of uncertainty in the state estimates is only folded into the estimation and control algorithms that are informed by the model. This will be discussed further in Section V.

A. The Modeling Process

State Analysis provides an iterative process for discovering state variables of the system under control and for incrementally constructing the model. The steps in this process are as follows:

1. Identify needs—define the high-level objectives for controlling the system.
2. Identify state variables that capture what needs to be controlled to meet the objectives, and define their representation.

3. Define state models for the identified state variables—these may uncover additional state variables that affect the identified state variables.
4. Identify measurements needed to estimate the state variables, and define their representation.
5. Define measurement models for the identified measurements—these may uncover additional state variables.
6. Identify commands needed to control the state variables, and define their representation.
7. Define command models for the identified commands—these may uncover additional state variables.
8. Repeat steps 2–7 on all newly discovered state variables, until every state variable and effect we care about is accounted for.
9. Return to step 1, this time to identify supporting objectives suggested by affecting states (a process called 'goal elaboration,' described later), and proceed with additional iterations of the process until the scope of the mission has been covered.

This modeling process can be used as part of a broader iterative incremental software development process, in which cycles of the modeling process can be interwoven with concurrent cycles of software implementation.

It should be noted that State Analysis provides a methodology for documenting significant states and effects *as well as the rationale for dismissing others*. If a state or effect is purposely omitted because it is insignificant, or if it is greatly simplified through abstraction, the reasons should be documented.

B. Example

We now present a simple example to illustrate this iterative process. Consider the problem of powering up a rover's navigation camera (step 1). One of the key state variables associated with this activity is the Camera Power State (step 2). We select an appropriate state representation for the Camera Power State: real-number values in Watts for mean and standard deviation. For the purposes of this example, we choose a simple state model for the behavior of this state variable, described by the following textual representation (step 3):

- If Camera Power Switch Position is Open (or Tripped-Open) or if Power Bus Voltage is less than `threshold`, Camera Power State = 0 Watt;
- Otherwise,
  - if Camera Health = Healthy, Camera Power State = 10 Watts;
  - if Camera Health = Short-Circuit, Camera Power State > 10 Watts.

(Note that this model is highly simplified for the purposes of illustrating the modeling process; a real model for Camera Power State would undoubtedly be more complex.)

This state model makes reference to three other states of the system under control: 'Camera Power Switch Position', 'Power Bus Voltage' and 'Camera Health'. In this example, we assume there are no direct measurements or commands associated with Camera Power State (steps 4–7). This completes our first iteration of the modeling process. Figure 6 shows a graphical representation of the states and effects we have documented thus far. This representation, which we call a State Effects Diagram, provides a convenient view of the state variables in the system under control, and the physical effects between these state variables.

Let us consider a second iteration, focusing on the Camera Power Switch Position state variable (step 2). The representation for this state variable is discrete, where the switch can be Open, Closed, or Tripped-Open. We choose to

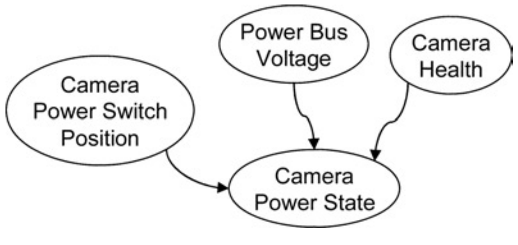
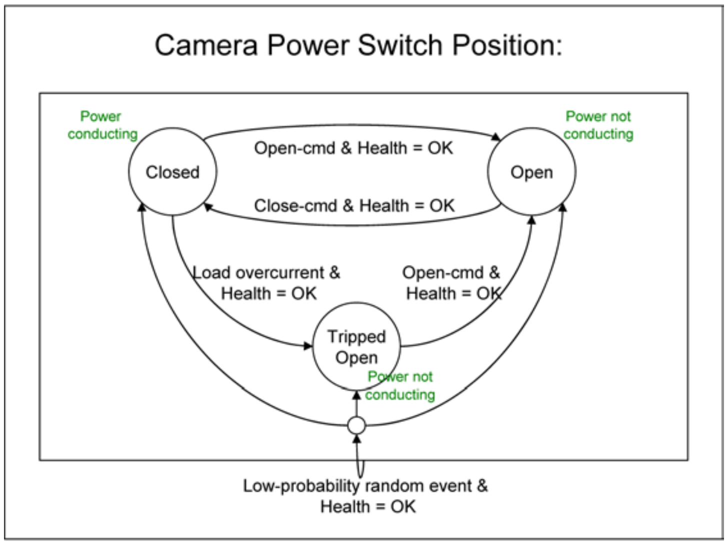


Fig. 6 State Effects Diagram after one iteration of the modeling process. State variables of the system under control are represented as ovals, and state effects are denoted by arrows.



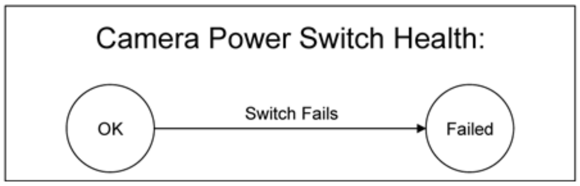


Fig. 7 StateCharts for the Camera Power Switch Position and Camera Power Switch Health state variables. The three discrete state values for Camera Power Switch Position each have an associated power conduction behavior. This model allows for a low-probability random event that can change the switch position state arbitrarily, presuming the switch health is not Failed (i.e., stuck). This type of a single-event upset behavior, although rare, should be captured as a possible response of the switch.

specify the state model for this state variable in the form of a StateChart[4] (step 3), which is a convenient representation for discrete state models that are fairly commonly used by systems engineers (Fig. 7). We note that the behavior in this state model is affected by two state variables, 'Camera Power State' (the "load overcurrent" condition on the transition from Closed to Tripped-Open corresponds to Camera Power State >10 Watts) and 'Camera Power Switch Health' (the "Health $=$ OK" conditions on all of the transitions, which imply that nominal transitions between switch position states require that the switch be healthy, and not stuck). These effects are depicted in the updated State Effects Diagram in Fig. 8. The effects of the commands shown in the StateChart will be discussed when we define the command model in step 7.

Note that we could have decided to combine Camera Power Switch Position and Camera Power Switch Health into a single state variable, 'Camera Power Switch Position & Health'. A reasonable state model for this combined state variable is depicted in Fig. 9. Deciding whether to combine state variables or keep them separate boils down to a tradeoff between the complexity of a combined model and the number of separate models that would be necessary to capture the same behavior. As a guideline, we would generally consider combining state variables if:

- intricate couplings make for a highly-connected State Effects Diagram;
- they affect many of the same measurements;
- they are affected by many of the same commands; or
- they affect the results of many of the same commands.

Conversely, we generally use separate state variables when we wish to transport, estimate, or control them separately (typically for efficiency, when they tend not to change together).

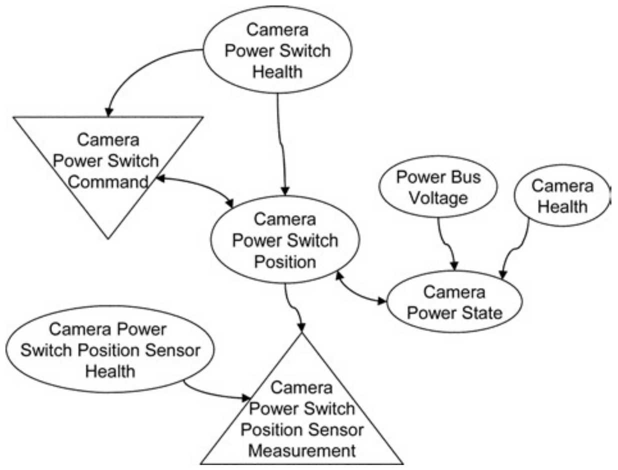

Camera Power Switch Health
Camera Power Switch Command
Camera Power Switch Position
Power Bus Voltage
Camera Health
Camera Power State
Camera Power Switch Position Sensor Health
Camera Power Switch Position Sensor Measurement

Fig. 8 State Effects Diagram after two iterations of the modeling process.

We assume that the power switch has an associated sensor that provides a measurement of the switch position, either "open," "tripped-open," or "closed" (step 4). We define the measurement model (measurement expressed as a function of its affecting states), as follows (step 5):

– if Camera Power Switch Position Sensor Health is Healthy, Measurement = Camera Power Switch Position;
– if Camera Power Switch Position Sensor Health is Stuck-Reading-Open, Measurement = Open (independent of the Camera Power Switch Position);
– if Camera Power Switch Position Sensor Health is Stuck-Reading-Closed, Measurement = Closed (independent of the Camera Power Switch Position);
– if Camera Power Switch Position Sensor Health is Stuck-Reading-Tripped-Open, Measurement = Tripped-Open (independent of the Camera Power Switch Position).

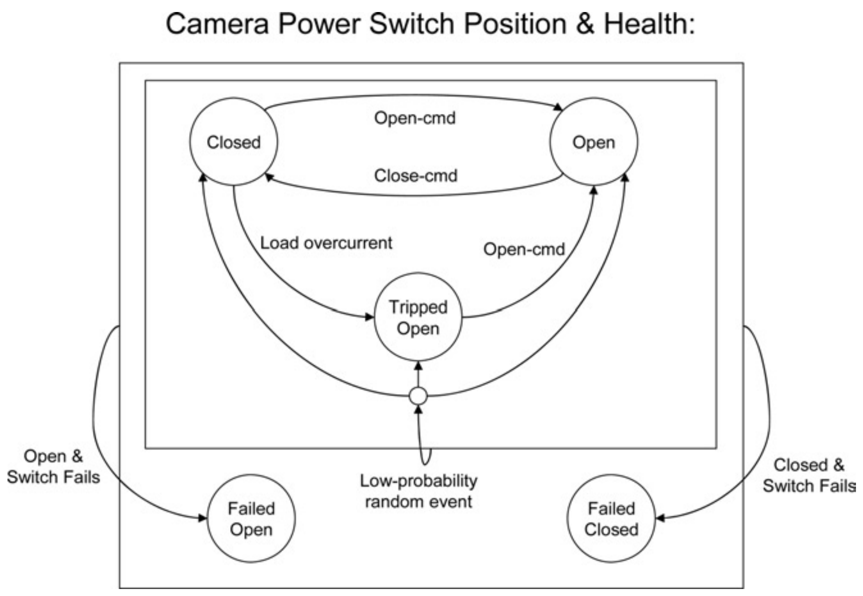
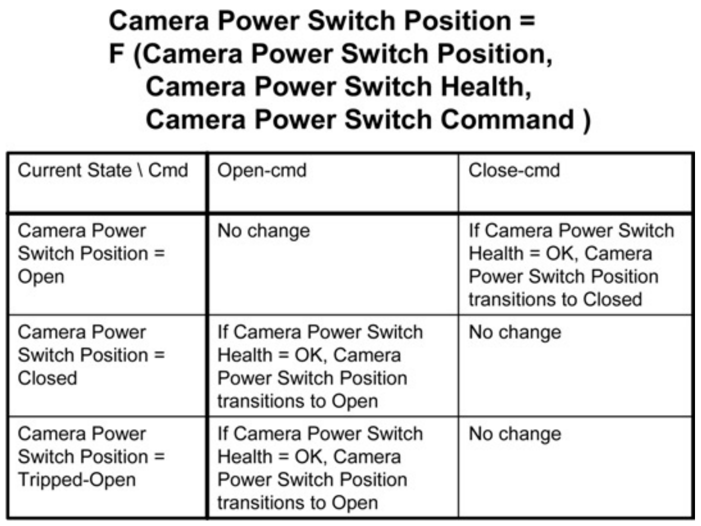


Fig. 9 State model for the combined Camera Power Switch Position & Health state variable (StateChart representation).

This measurement model specifies the dependence of the measurement not only on the Camera Power Switch Position state variable, but also on another as-yet-unspecified state variable: the ‘Camera Power Switch Position Sensor Health’. This simple model assumes three different possible failure modes for the sensor, corresponding to the sensor readings being “stuck” at one of the three possible outputs. In a real model, we might also want to allow for the possibility that the sensor could exhibit other failure modes, such as intermittent random readings. Measurements are depicted on the State Effects Diagram as triangles, as shown in Fig. 8, representing the state of the data collections associated with the measurements. The diagram shows incoming effect arrows from all state variables that appear in the measurement model.

The camera power switch is, by definition, an actuator. We therefore specify a command that will allow us to affect a change in the camera switch position state. We define this command to include a parameter, to be set by the appropriate controller, which indicates the desired operation: “Open-cmd” or “Close-cmd” (step 6). Associated with this command we define a command model, which specifies how the Camera Power Switch Position state variable changes in response to the command (step 7). Command models are used to describe *instantaneous* changes of state; we ascribe cascading effects and delayed behavior to the state model. Tables, such as the one in Fig. 10, are a convenient way to represent command models for discrete commands like the camera power switch command. However, we note that this command model was previously fully specified by the StateChart in Fig. 7; such model representations can be used to concisely capture state, measurement and command models in a single form. Commands are depicted on the State Effects Diagram (see Fig. 8) as inverted triangles, with an outgoing arrow pointing to the commanded state variable (Camera Power Switch Position, in this case), and incoming arrows from the state variables that have an impact on the effects of the command (Camera Power Switch Position and Camera Power Switch Health, in this case). The double-headed arrow between Camera Power Switch Position and Camera Power Switch Command indicates that the switch position is affected by the command, and that its value prior to issuing the command affects the results of the command. Just as for measurements, a command in the State Effects Diagram represents the state of the data collection associated with the command.

We have just stepped through two iterations of the modeling process. Further iterations on all newly discovered state variables would eventually produce a State Effects Diagram like the one depicted in Fig. 11. This diagram reflects the combined Camera Power Switch Position & Health state variable, as discussed above. There are state variables in this figure that require further modeling, so this is not the end of the process. As we have illustrated, our modeling approach can lead us a long way from the states we started from, but this is a good thing: it allows us to quickly ascertain the scope of the problem.


Camera Power Switch Position =
F (Camera Power Switch Position,
Camera Power Switch Health,
Camera Power Switch Command)

Current State \ Cmd	Open-cmd	Close-cmd
Camera Power Switch Position = Open	No change	If Camera Power Switch Health = OK, Camera Power Switch Position transitions to Closed
Camera Power Switch Position = Closed	If Camera Power Switch Health = OK, Camera Power Switch Position transitions to Open	No change
Camera Power Switch Position = Tripped-Open	If Camera Power Switch Health = OK, Camera Power Switch Position transitions to Open	No change

Fig. 10 Command Model associated with the Camera Power Switch Command, in tabular form.

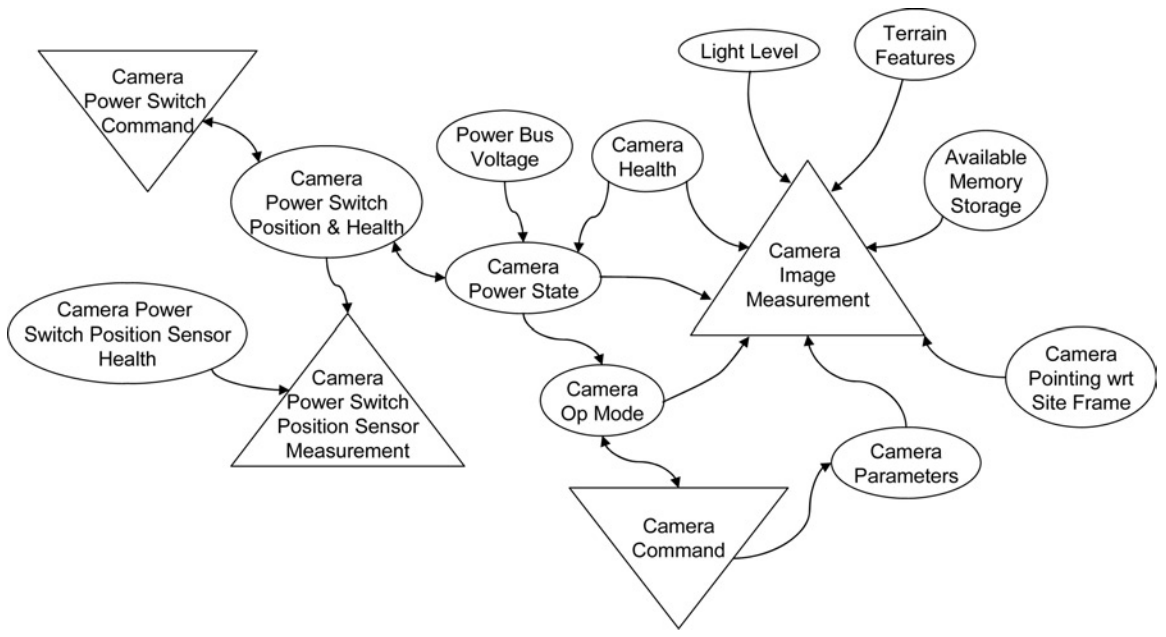

Camera Power Switch Command
Light Level
Terrain Features
Power Bus Voltage
Camera Health
Available Memory Storage
Camera Power Switch Position & Health
Camera Power State
Camera Image Measurement
Camera Power Switch Position Sensor Health
Camera Power Switch Position Sensor Measurement
Camera Op Mode
Camera Pointing wrt Site Frame
Camera Parameters
Camera Command

Fig. 11 State Effects Diagram for simple camera example.

In addition to illustrating the iterative modeling process we adopt in State Analysis, this simple example also shows how State Analysis promotes early consideration of component health states and fault modes. This is in contrast with traditional systems engineering practice, where consideration of off-nominal behavior ("failure modes and effects analysis") is commonly postponed until later in the spacecraft design process, and can lead to ad-hoc fault protection implementation. In State Analysis, fault behaviors are included in the state models and are treated just like any nominal state; as a result, fault detection, diagnosis, and recovery become integral aspects of the design of the system architecture.

Furthermore, our example shows how State Analysis is flexible with respect to model representation. We provide systems engineers with broad latitude to capture the state, measurement and command models in a form that is most convenient for their specific application.

In summary, State Analysis provides a modeling process that produces requirements on system and software design in the form of explicit models of the system under control. These models capture the systems engineer's understanding of the state-level behavior and interactions in the system. In the next section, we will discuss how State Analysis uses information from the models in the design of the mission software.

V. Using the Model to Design the Control System

The state, measurement and command models defined as part of the State Analysis process (described in the previous section) are used throughout the control system. In this section, we outline how state, measurement and command models are used to inform the software design. In particular, we discuss the design of the Mission Planning and Execution functions, and the Estimation and Control algorithms (recall Fig. 2).

A. Mission Planning and Execution

As mentioned in Section II, one of the key features of State Analysis is that it emphasizes goal-directed closed-loop operation. The control architecture in Fig. 2 includes a Mission Planning and Execution[5] function whose role is to produce and execute plans for accomplishing high-level mission objectives. Unlike the traditional "open-loop" approach to space mission planning and operation, where spacecraft operator intent is translated into sequences of low-level commands, we specify plans as temporally-constrained networks of goals (*goal networks*). Goal-directed operation represents a logical evolution of the spacecraft control paradigm, allowing operators to generate closed-loop sequences that implicitly account for system interactions. It enables (but does not impose) flexible autonomous operations, by freeing the ground controllers from having to worry about the exact state of the spacecraft, empowering the spacecraft to accommodate surprises without the need for ground intervention and improving reliability, despite uncertainty in our knowledge of the environment. Fault responses have always been goal-directed to some extent, out of necessity, and recent space missions, including the Cassini and Mars Exploration Rover spacecraft, have demonstrated a fair amount of goal-directed behavior for nominal operations. However, this powerful control paradigm has not yet been consistently applied across a mission in a way that allows it to be fully exploited by an onboard or ground-based reasoning system.

In order to enable goal-directed operation, systems engineers must define the types of goals that can be issued, the groups of goals that achieve higher-level goals (traditionally referred to as “blocks” or “macros”), and the system-specific logic needed to correctly plan and execute goals. In this subsection, we first define our notion of goal; we then show how the model of the system under control is used to elaborate goals into the fundamental building blocks of goal networks; and finally, we briefly address how these building blocks can be assembled and scheduled into goal networks for onboard execution.

1. Goals

In State Analysis, a goal is defined as a *constraint on the value history of a state variable over a time interval*. The start and end of each time interval is called a “time point,” a potentially variable moment in time. As part of the State Analysis process, a systems engineer specifies a dictionary of *goal types*, each with parametric state constraints and unspecified temporal constraints (which we represent diagrammatically, as shown in Fig. 12). A goal is specified by instantiating a goal type in the goal dictionary. Spacecraft operators construct activity plans in the form of goal networks, by interconnecting goals. Interconnections are made either by sharing time points among goals (e.g., one goal ends at the same time point as another goal starts), or by adding additional temporal constraints among time points (e.g., one goal starts an hour after another starts).

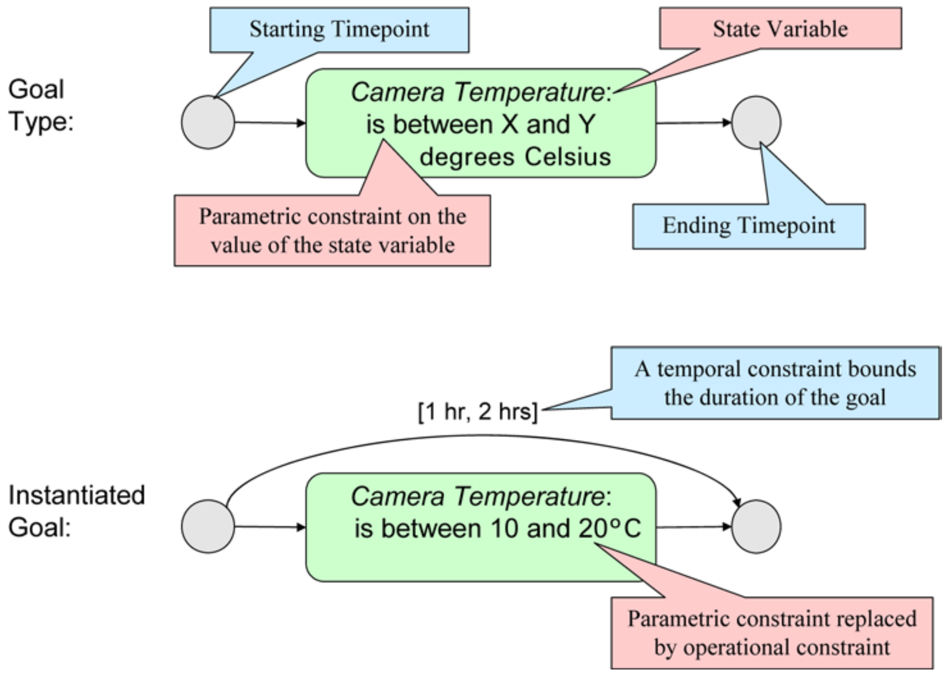

Starting Timepoint
State Variable
Goal Type:
Camera Temperature: is between X and Y degrees Celsius
Parametric constraint on the value of the state variable
Ending Timepoint
A temporal constraint bounds the duration of the goal
[1 hr, 2 hrs]
Instantiated Goal:
Camera Temperature: is between 10 and 20°C
Parametric constraint replaced by operational constraint

Fig. 12 The anatomy of a goal type and an instantiated goal. Every goal has a starting time point and an ending time point. A goal can be instantiated with a flexible temporal constraint on its duration, indicated by an arrow from its starting to its ending time point, labeled with a [min, max] duration window.

A goal is expressed as an assertion whose success or failure can be evaluated with respect to its state variable's value history (state timeline). We note that commands are not the same thing as goals. For example, if our objective were to bring a camera to an acceptable operating temperature, "At 2:00 pm, issue the close-switch command to the camera heater power switch" would not be a valid goal; what if we were to issue the close-switch command, immediately followed by an open-switch command? What if the switch failed to close? What if it tripped open? Clearly, we would not have achieved our underlying objective of heating the camera, even though we did issue the close-switch command as specified.

Similarly, actions in support of a higher level objective are not the same as a goal to achieve that objective. For example, "The Camera Heater is powered on from 2:00 pm to 3:00 pm" is a valid goal, per our definition; however, what if we power the heater, but an unanticipated cold Martian wind results in a net decrease of the camera temperature? Again, we would not have achieved our high-level objective of heating up the camera, even though the supporting action was properly executed.

Finally, settings of controller and estimator modes are also not the same as goals. For example, "The camera temperature controller shall be in Thermostatic Mode from 2:00 pm to 3:00 pm" is not a valid goal. The definition of a goal requires there to be a state variable to put the goal on. The state variable in this case would have to be the controller mode, but state variables are not defined for anything but states of the system under control. This avoids self-reference, preserving the principle of separation between control system and system under control discussed in Section I. Furthermore, by imposing a controller mode, we have dictated the actions of the control system. Perhaps there is a control mode that would both achieve the objective and lead to lower power consumption in the heater. By explicitly setting the control mode, we are not giving the control system the freedom to adjust its actions to the circumstances at hand.

Clearly we would prefer to coordinate control in a way that focuses on the state we really care about controlling, and that allows the control system the flexibility to decide how best to affect that state. This is exactly what our definition of goal implies. Goals specify what to achieve within the system under control, not how to achieve it within the control system; they express conditions that should persist over some time interval, and provide a statement of operational intent.

In our example, what we really care about is the Camera Temperature state. An appropriate goal for accomplishing the objective would be "The Camera Temperature is between 10 and 20°C from 2:00 pm to 3:00 pm" (instantiated from the goal type in Fig. 12). Given this goal, the software controller associated with the Camera Temperature state variable has the flexibility to change modes or issue commands as necessary to satisfy our intent.

We note that the intent of this goal is to maintain the camera temperature within a range; however, this goal will fail unless the temperature is already in range at 2:00 pm. This is because the temperature cannot be changed instantaneously at 2:00 pm to meet the goal's condition after 2:00 pm (per the physics specified in the Camera Temperature state model). Consequently, a "maintenance" goal of this sort is generally immediately preceded by a "transition" goal that achieves the appropriate precondition. In this case, an appropriate transition goal would be "Camera Temperature is transitioning to be between 10 and 20°C no later than 2:00 pm."

Since our representation of state knowledge is continuous in time, goals can express constraints on the time-varying behavior of the state value history, e.g., "Camera temperature is between 10 and 20°C from 2:00 pm to 3:00 pm, and its rate of change does not exceed 1 degree Celsius per minute during this time period." In another example, goals can express constraints on time-averaged behavior, e.g., "Camera Temperature is between 10 and 20°C between 2:00 pm and 8:00 pm, over at least 80% of each hour." Further examples can be found in goals that embody the sort of system safety constraints typically associated with fault monitors in traditional software, where persistence tests are a feature. For goals of this sort, success of the goal cannot be determined by looking at the instantaneous value of the state variable, but rather, only by following the progress of the state over time. This justifies our earlier definition of a goal as a constraint on the value history of a state variable (and not just on its value at any specific point in time).

The examples of goals we have presented thus far have focused on *control goals*, i.e., goals that express constraints on desired nominal values of state variables. As we mentioned in Section II, goals are also used to specify desired quality of state knowledge, to be achieved by estimators. We refer to these types of goals as *knowledge goals*, e.g., "Camera temperature standard deviation is less than 0.5 degree Celsius from 1:00 pm until 5:00 pm," or "Camera power switch position is known with 95% certainty or better from 1:00 pm until 5:00 pm."

We have introduced control and knowledge goals as distinct, for illustrative purposes. However, it does not always make sense, for certain representations of state knowledge, to speak of constraints on the state value separately from constraints on the quality of state knowledge. Thus, a single goal can specify constraints on *both* state value and quality of knowledge, e.g., “Camera temperature mean value, plus or minus three sigma, is in the range 10–20°C [$10 \leq \text{mean} - 3\sigma \leq \text{mean} + 3\sigma \leq 20$], from 2:00 pm to 3:00 pm.”

Finally, another type of state we described in Section III is the state of a data collection. We can therefore specify goals on these data states. Goals of this type are called *data goals*, e.g., “Camera Temperature measurement data collection state contains at least one measurement less than 10 seconds old, from 1:00 pm until 5:00 pm.”

2. *Goal Elaborations*

As we discussed in Section IV, our model of the system under control captures the physical cause-and-effect relationships between state variables. Because of these interactions between state variables, it is clear that there is more to controlling than simply asserting a goal on a state variable of interest, and expecting it to be achieved in stand-alone fashion, without considering its implications on other related states in the system. Furthermore, many goals simply cannot be achieved without also asserting supporting goals on other state variables that impact our state variables of interest.

Part of the State Analysis methodology is the specification of fundamental “blocks” of goals, which can be assembled into plans and which account for the causality between state variables in the system under control. We call these fundamental blocks *goal elaborations*. A goal’s elaboration specifies supporting goals on related state variables that may need to be satisfied in order to achieve the original goal, or alternatively, may simply make the original goal more likely to succeed.

Goal elaborations are defined based on engineering judgment, our model of the system under control, and the following five rules:

1. A goal on a state variable may elaborate into concurrent control goals on directly affecting state variables (concurrent goals share the same start and end time points).
2. A control goal on a state variable elaborates to a concurrent knowledge goal on the same state variable (or they may be specified jointly in a single control and knowledge goal).
3. A knowledge goal on a state variable may elaborate to concurrent knowledge goals on its affecting and affected state variables.
4. Any goal may elaborate into preceding goals (typically on the same state variable). For example, a “maintenance”-type goal on a state variable may elaborate to a “transition”-type goal on the same state variable, which has an ending time point coincident with the starting time point of the “maintenance”-type goal.
5. A goal’s elaboration can include uncertain temporal constraints to reserve time in the schedule for actions required by the goal.

Goal elaborations are defined locally for each goal by considering only direct effects (that is, effects of states that are only a single step away in the State Effects Diagram). It is also important to note that goal elaborations are different from traditional macro expansions, in that *the elaborated goal is not replaced by the goals in its elaboration*, but rather, these goals are added to the plan along with the elaborated goal. That way, the original intent of the expansion is not lost. The elaborated goal can be monitored for success along with all its supporting goals, allowing the system to determine how it is doing with respect to the original intent.

Let us consider the simple camera power example to illustrate how to apply the above rules in the elaboration of goals. We assume for our purposes that the scope of the simple model is as shown in Fig. 8 (as opposed to the more complete model in Fig. 11). Consider the following goal on the Camera Power State: “Camera Power State is 10 ± 1 Watts.” Applying the elaboration rules, our models of the system under control and some reasonable engineering judgment, this goal can be elaborated as shown in Fig. 13. Per Rule #1, our control goal on Camera Power State elaborates into concurrent supporting control goals on the affecting state variables (recall the State Effects Diagram in Fig. 8): Camera Power Switch Position, Camera Health, and Power Bus Voltage. Per Rule #2, it also elaborates into a knowledge goal that asserts that the uncertainty in the Camera Power State must be limited to a standard deviation of 0.2 Watts or less. Finally, per Rule #4, our goal that constrains Camera Power to be maintained at 10 ± 1 Watts

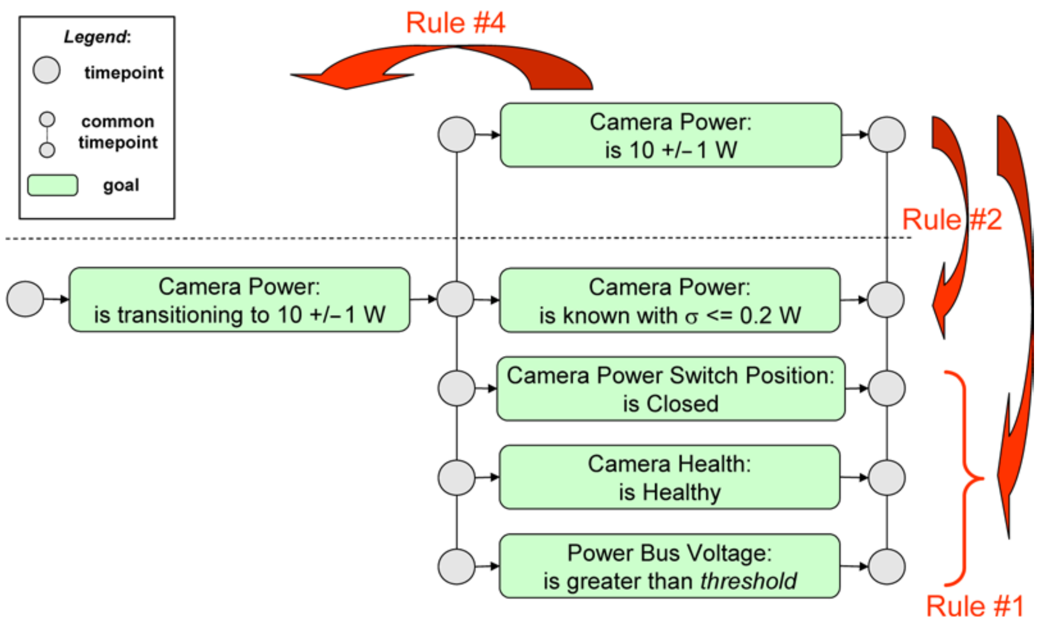

Legend:
timepoint
common timepoint
goal
Rule #4
Camera Power: is 10 +/– 1 W
Rule #2
Camera Power: is transitioning to 10 +/– 1 W
Camera Power: is known with σ <= 0.2 W
Camera Power Switch Position: is Closed
Camera Health: is Healthy
Power Bus Voltage: is greater than threshold
Rule #1

Fig. 13 The elaboration for the "Camera Power State is 10 ± 1 Watts" goal.

must be immediately preceded by a goal that results in Camera Power reaching the 10 Watt level. Since our goal is a control goal, Rule #3 does not apply.

Note that an elaboration is nothing more than a goal network appended to the original goal at its time points. Goal elaboration is an iterative process, so supporting goals that appear in an elaboration are, in turn, elaborated. Thus, elaboration works by iteratively adding goals into an increasingly larger goal network. The elaborations chain together to encompass the full set of relevant state variable interactions. As a second example, consider the elaboration of one of the supporting goals from the first elaboration: "Camera Power is known with $\sigma \leq 0.2$ Watt" (see Fig. 14). In this case, the application of Rule #1 does not result in the elaboration of the knowledge goal into any control goals

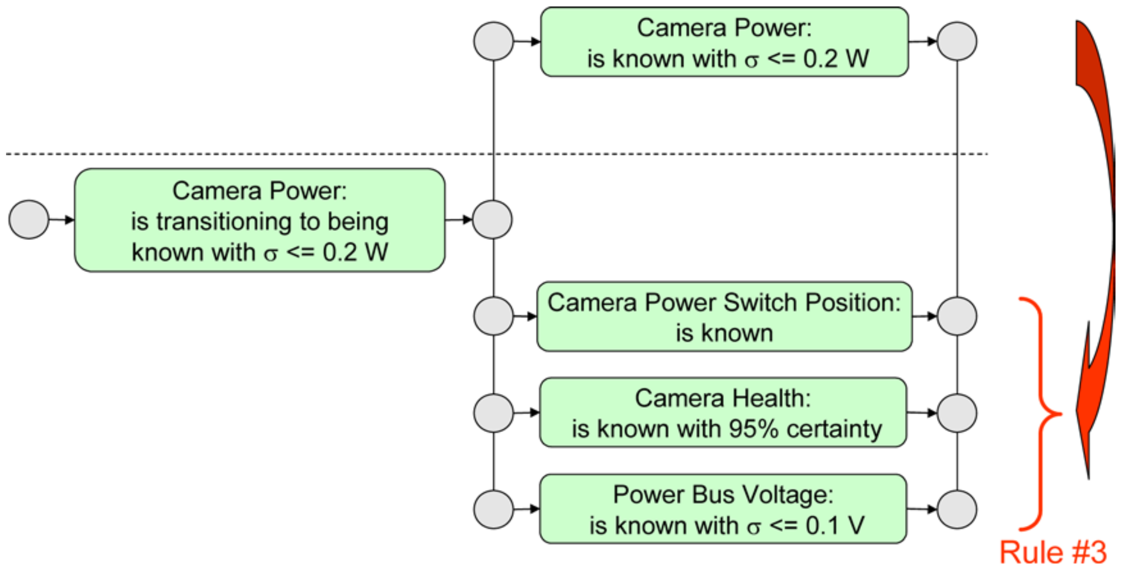

Camera Power: is known with σ <= 0.2 W
Camera Power: is transitioning to being known with σ <= 0.2 W
Camera Power Switch Position: is known
Camera Health: is known with 95% certainty
Power Bus Voltage: is known with σ <= 0.1 V
Rule #3

Fig. 14 The elaboration for the "Camera Power State is known with $\sigma \leq 0.2$ Watt" goal.

(implying that maintaining the quality of knowledge of the Camera Power State requires no explicit control over its affecting states). Rule #3 results in the elaboration of our knowledge goal into supporting goals constraining the knowledge of the affecting states. We see that different representations of uncertainty are accommodated: for Camera Power State and Power Bus Voltage we use standard deviations (in Watts and Volts, respectively), for Camera Power Switch Position we distinguish simply between "known" and "unknown," and for Camera Health we compute a percentage certainty level. The specific constraints in the supporting goals may be either directly inferred from the state models or determined based on engineering judgment and analysis of the state models. We also see that our "maintenance of knowledge" goal on Camera Power State elaborates into a preceding transitional goal on the same state variable (Rule #4).

Similar elaborations are specified for the remaining supporting goals, for each of their own supporting goals, and so on. We can manage the complexity and scale of the iterative elaboration process by making judicious engineering decisions to identify "terminal" goals that require no further elaboration. The issue of how to deal with loops in the elaborations is not discussed here, but clearly must be addressed, by either engineering the elaborations to explicitly avoid loops, or adopting an iterative elaboration algorithm that converges to the final elaborated goal network. We can also leverage automated algorithms to assemble goal networks from the elaborations and schedule them, as we will discuss in the following subsection.

We support alternative ways of accomplishing a goal by allowing the definition of multiple alternative goal elaborations, called *tactics*. Thus, should any of the supporting goals in an elaboration fail, it becomes possible to re-elaborate with an alternate tactic to accomplish the same objective. Context-specific elaboration can be enabled by conditioning alternative elaboration tactics on different state constraints.

Currently, systems engineers produce goal elaborations by hand, using the aforementioned elaboration rules. We note that the existence of an explicit model opens up the possibility of automatic generation of goal elaborations from the state models. Further work is needed in the areas of model representation and model-based reasoning before such a capability can be implemented. We see recent progress in the compilation of model-based programs[6] as a potential solution to this problem; this is an area for future work.

In the MDS software architecture, described in Section VII, elaborations are performed within a coding framework. In that context, essentially arbitrary implementation is possible, but for ease of design a compact language called GEL (Goal Elaboration Language) has been provided. Graphical tools for elaboration design are also planned and have been prototyped.

Before we move on to address the topic of goal networks, we introduce a mechanism that enables "reactive" coordination of activity, as opposed to the more "deliberative" (pre-planned) coordination we have introduced via elaboration of goals into supporting goals with explicit constraints. Reactive execution-time coordination is needed during activities like rover driving and steering, or attitude control thrusting, for which it would not be appropriate to specify explicit goals on individual rover wheels or thrusters at plan-time. For these types of activities, we expect one controller (or estimator) to coordinate the control of one or more hardware components within very short reaction times.

In State Analysis, the mechanism we use for this is called *delegation*, because it involves one state variable delegating the authority over its controller to another state variable's controller or estimator. Not surprisingly, we specify delegation relationships in terms of our model of the system under control. Delegation from state variable B to state variable A is an option when the value of state variable B affects the value of state variable A (see Fig. 15). To enable the exchange of reactive goals between the controller (or estimator) of A and the controller of B at run-time, we must plan ahead: during elaboration, the delegator (B) authorizes the delegate (A) to send reactive goals to B subject to an allocation requested in the elaboration of a goal on A. This allows our planning system to allocate and account for the level of delegator "resource" (state variable B) that might be needed to achieve the goal on A. During execution, the controller (or estimator) of A can then send goals "on-the-fly" to the controller of B, as necessary to achieve its goal, as long as these reactive goals honor the resource allocations established at elaboration time. Figure 16 shows how delegation is enabled via goal elaboration, for a rover driving and steering example.

In the following subsection, we discuss how the goal elaborations are used in the construction and scheduling of goal networks.

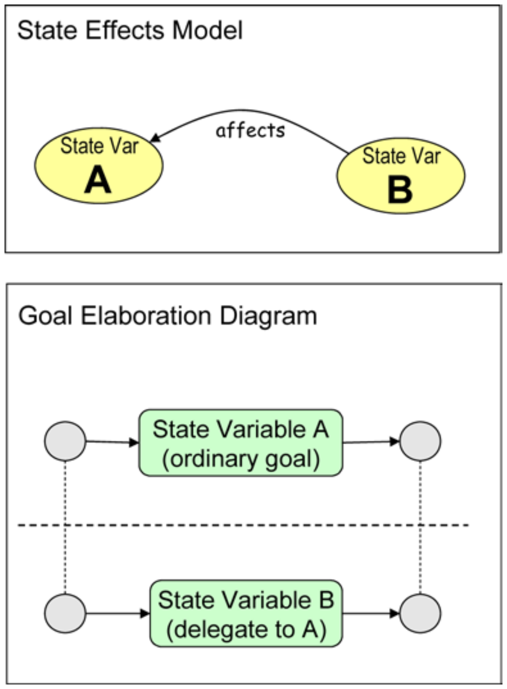


Fig. 15 Reactive execution-time coordination via delegation is enabled during goal elaboration.

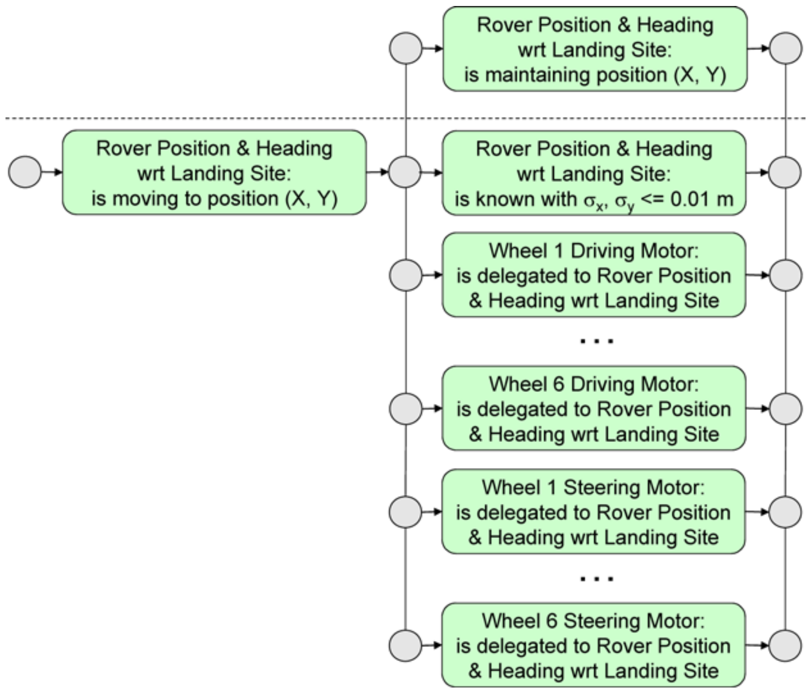

Rover Position & Heading wrt Landing Site: is maintaining position (X, Y)
Rover Position & Heading wrt Landing Site: is moving to position (X, Y)
Rover Position & Heading wrt Landing Site: is known with σ_x, σ_y <= 0.01 m
Wheel 1 Driving Motor: is delegated to Rover Position & Heading wrt Landing Site
...
Wheel 6 Driving Motor: is delegated to Rover Position & Heading wrt Landing Site
Wheel 1 Steering Motor: is delegated to Rover Position & Heading wrt Landing Site
...
Wheel 6 Steering Motor: is delegated to Rover Position & Heading wrt Landing Site

Fig. 16 Goal elaboration that establishes delegation of wheel motors for driving and steering to Rover Position & Heading controller.

3. Goal Network Scheduling

Once the necessary set of goal elaborations has been defined, they can be encoded into the ground and flight software, enabling ground operators to simply specify desired behavior in terms of high-level goals on the state variables of interest, and allowing the Mission Planning and Execution system to automatically:

- elaborate these goals into the set of appropriate supporting goals;
- add these elaborated goals into the current goal network, which includes all background goals (capturing flight rules and constraints) and previously-scheduled activities; and
- schedule the augmented goal network to satisfy any specified temporal constraints and to eliminate any conflicts that arise, and verify the consistency of the full goal network that results.

This is an iterative search process that may require backtracking, and the use of heuristics for efficiency, to guide the search. The details of the Mission Planning and Execution system have been previously published.[5] Here we focus on providing a high-level overview of its main scheduling-related functions and describing how they leverage the model of the system under control.

We recall from Section III that state timelines are used for representing intent. More specifically, for each state variable in the system under control, the intent is captured as a sequence of executable goals ("x-goals") on its timeline. These x-goals are created through a process called *scheduling*, which involves four steps:

1. adding appropriate temporal constraints to the goal network to ensure that all potentially concurrent goals for each state variable are consistent (non-conflicting and achievable);
2. merging all the goals into x-goals on the state timelines;
3. propagating state effects from affecting states to affected states, and projecting each state variable over time according to state constraints and initial state values; and
4. checking the consistency of the resulting x-goal timelines, including checking to make sure that the projected states are compatible with the scheduled x-goals.

This is an automated process performed by a scheduling engine, which must be informed by the models of the system under control provided by systems engineers. The means by which the models inform the scheduling is through a handful of logic functions specified as part of the State Analysis process. For instance, we must specify the logic associated with merging multiple goals on a given state variable (step #2). This corresponds to an intersection operation performed on the goals' state constraints, as shown in Fig. 17.

State Analysis also specifies the logic used in step #3 to propagate state effects across the system (Fig. 18) and project state into the future (Fig. 19). This logic is derived directly from the state models described in Section IV. This projection logic provides a mechanism for generalized resource management for the system under control.

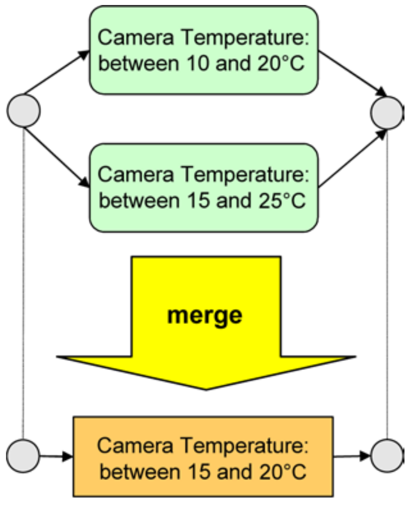

Camera Temperature: between 10 and 20°C
Camera Temperature: between 15 and 25°C
merge
Camera Temperature: between 15 and 20°C

Fig. 17 Goal merging function.

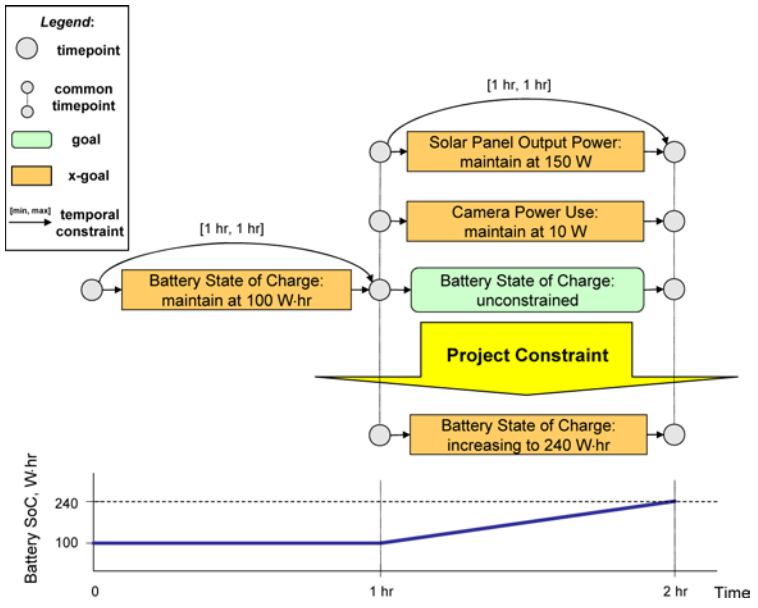

Legend:
timepoint
common timepoint
goal
x-goal
[min, max]
temporal constraint
[1 hr, 1 hr]
Solar Panel Output Power: maintain at 150 W
Camera Power Use: maintain at 10 W
[1 hr, 1 hr]
Battery State of Charge: maintain at 100 W·hr
Battery State of Charge: unconstrained
Project Constraint
Battery State of Charge: increasing to 240 W·hr
Battery SoC, W·hr
240
100
0
1 hr
2 hr
Time

Fig. 18 **Projecting effects across the state variables of the system under control. Lower plot depicts the resulting constraint on Battery State of Charge vs. time.**

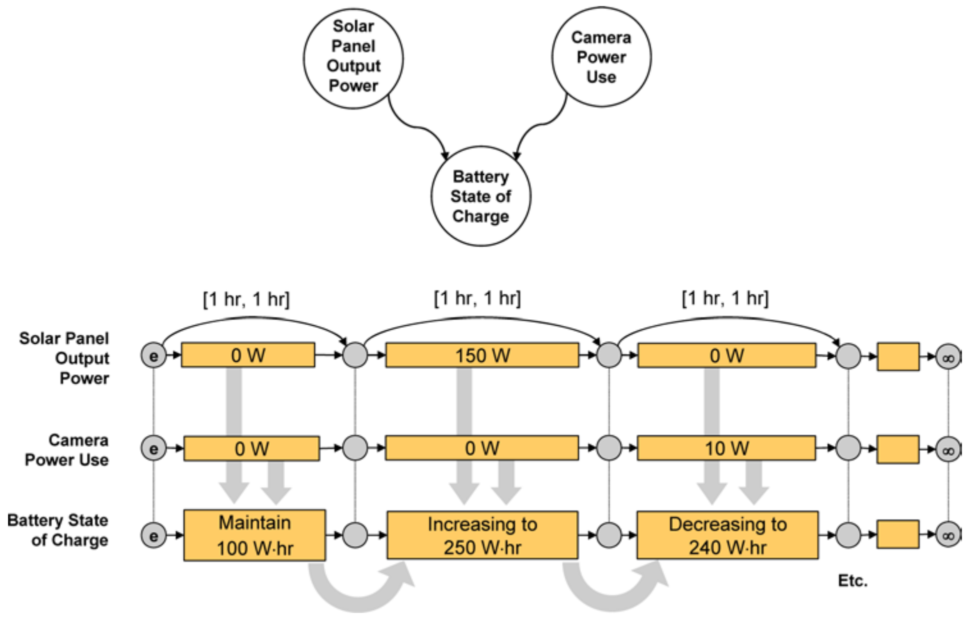


Fig. 19 **Projecting state effects across state over time. This simple example assumes that the Camera is the sole power user. The time point labeled “e” corresponds to an “epoch” reference time, and the time point labeled “∞” corresponds to infinity, denoting the end time point of the network. Though the temporal constraints on these x-goals are fixed 1-hour intervals, flexible temporal constraints are also supported.**

Finally, we must also specify the logic associated with checking the consistency of the resulting x-goal timelines (step #4). This involves checking each x-goal for achievability, checking that each consecutive pair of x-goals is compatible (i.e., that the transition from one x-goal to the next is achievable, as in Fig. 20), and finally, checking that the computed state projections are consistent with the x-goal constraints. These logic functions take the form of Boolean queries that are posed to the appropriate estimator (in the case of a knowledge x-goal) or controller (in the case of a control x-goal), which uses the specified logic and the state information available at schedule time to answer the query. Aside from checking the projections, we do not check affecting states, because elaboration has already imposed any conditions on them that we would need to check. Likewise, if there are timing considerations, they have already been imposed via temporal constraints in the elaboration.

Scheduling is finished when all the goals in all the timelines have been scheduled, all the effects of all the x-goals have been combined and merged with the affected timeline, and all the x-goals are consistent and their transitions are consistent. Figure 21 shows the results of scheduling a goal network for a simple example with two state variables, ‘Camera Temperature’ and ‘Heater Switch Position and Health’. Prior to scheduling, the timelines for both state variables are unconstrained (top, Fig. 21). In this example, we wish to schedule three goals: “Camera Temperature is between 10 and 20°C for 1 hour, starting sometime in the next 3 hours,” “Heater Switch Position and Health is Healthy and Off, for up to 2 hours,” and “Camera Temperature is always between 15 and 25°C.” The first of these goals elaborates into a preceding transitional goal on Camera Temperature (which itself has an elaboration) and a concurrent goal on Heater Switch Position and Health, while the other two goals are defined to be terminal goals, with no elaborations. As shown in the figure, the scheduling process produces x-goals that result from merging compatible constraints on Camera Temperature, and the insertion of a precedence constraint (labeled with a “0” time interval) between two incompatible constraints on Heater Switch Position and Health.

We have described elaboration and scheduling as sequential steps, which accurately reflects the current implementation of the MDS Mission Planning and Execution system. However, by interleaving elaboration and scheduling we could improve the performance by potentially reducing the amount of backtracking search we need to do to find a consistent schedule. The design and implementation of an algorithm for interleaved elaboration and scheduling is an area of current work.

4. *Goal Network Execution*

Once the goal network has been fully elaborated and scheduled, it is ready to be executed.[5] The time points between x-goals on each state timeline can be thought of as events; these events can be shared across multiple state timelines. Executing a goal network is paced by “firing the time points” on its x-goal timelines, at times consistent with the temporal constraints imposed on these time points. Before a time point can fire, all the goals that have this time point as their starting time point must be “ready” to start executing; that is, the post-conditions and pre-conditions associated with the transition from the current x-goal to the next x-goal on the timeline must be satisfied. As each time point fires, it becomes “grounded” in time (assuming it had flexibility in its temporal constraints) and the next x-goal is dispatched to the appropriate state variable’s controller (in the case of a control goal) and estimator (in the case of a knowledge goal). The estimators and controllers achieve these goals by updating state knowledge and issuing appropriate commands to the system under control, respectively; this is the topic of the next subsection of the paper on “Estimation and Control.”

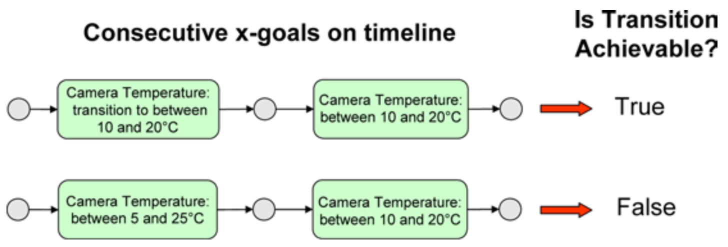
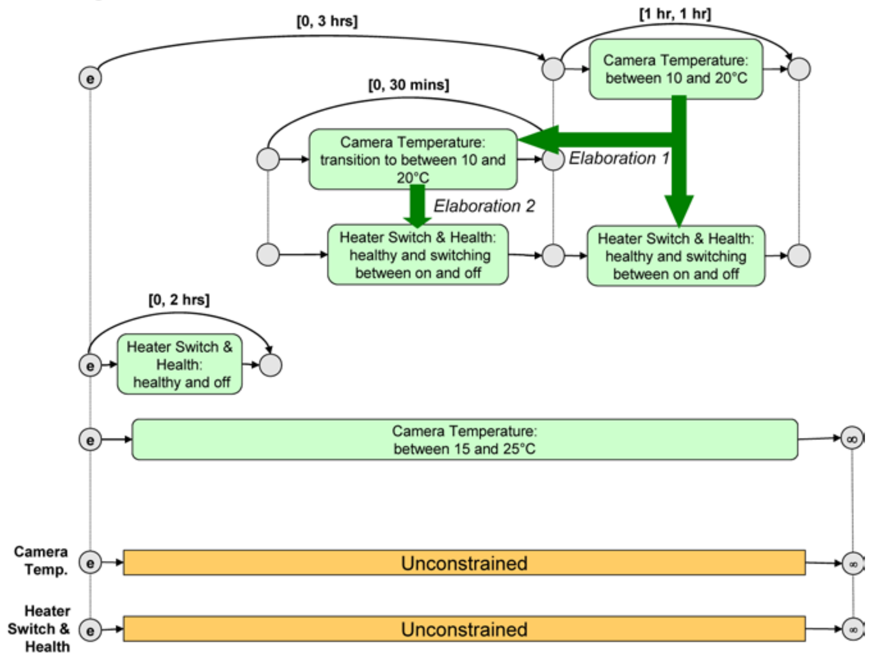

Consecutive x-goals on timeline
Is Transition Achievable?
Camera Temperature: transition to between 10 and 20°C
Camera Temperature: between 10 and 20°C
True
Camera Temperature: between 5 and 25°C
Camera Temperature: between 10 and 20°C
False

Fig. 20 Transition achievability function.

Before scheduling:


[0, 3 hrs]
[1 hr, 1 hr]
Camera Temperature: between 10 and 20°C
[0, 30 mins]
Camera Temperature: transition to between 10 and 20°C
Elaboration 1
Elaboration 2
Heater Switch & Health: healthy and switching between on and off
Heater Switch & Health: healthy and switching between on and off
[0, 2 hrs]
Heater Switch & Health: healthy and off
Camera Temperature: between 15 and 25°C
Camera Temp.
Unconstrained
Heater Switch & Health
Unconstrained
e
∞

After scheduling:

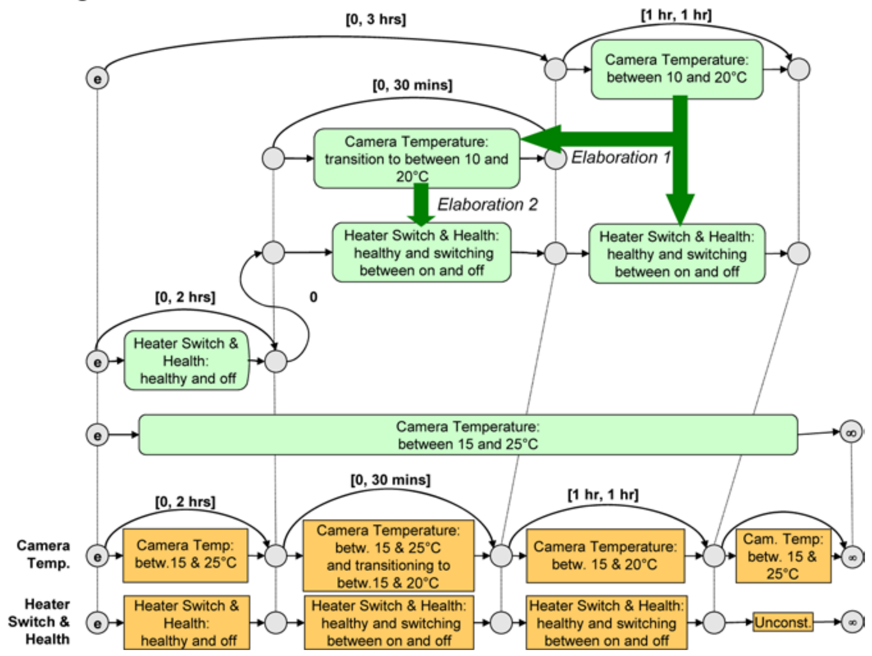

[0, 3 hrs]
[1 hr, 1 hr]
Camera Temperature: between 10 and 20°C
[0, 30 mins]
Camera Temperature: transition to between 10 and 20°C
Elaboration 1
Elaboration 2
Heater Switch & Health: healthy and switching between on and off
Heater Switch & Health: healthy and switching between on and off
0
[0, 2 hrs]
Heater Switch & Health: healthy and off
Camera Temperature: between 15 and 25°C
[0, 2 hrs]
[0, 30 mins]
[1 hr, 1 hr]
Camera Temp.
Camera Temp: betw.15 & 25°C
Camera Temperature: betw. 15 & 25°C and transitioning to betw.15 & 20°C
Camera Temperature: betw. 15 & 20°C
Cam. Temp: betw. 15 & 25°C
Heater Switch & Health
Heater Switch & Health: healthy and off
Heater Switch & Health: healthy and switching between on and off
Heater Switch & Health: healthy and switching between on and off
Unconst.
e
∞

Fig. 21 Scheduling example (top, before scheduling; bottom, after scheduling).

As execution proceeds, the status of each currently-executing goal in the goal network is monitored to see whether its state constraint is being satisfied. A goal "fails" if its state constraint is determined to be violated, based on the state history stored in the state variable. Note that we monitor the status of each individual goal in the goal network, rather than just monitoring the status of the x-goals that result from merging goals onto the state timelines. As a result, we can specifically determine which of the original specifications of intent are not being met, and respond appropriately; for instance, we would probably choose to respond differently to the failure of a goal on a state variable that was associated with some routine science observation than to the failure of a goal on the same state variable that asserted a mission health and safety constraint. A number of different responses to goal failure are allowed, including:

- simply removing the goal (and all of its supporting goals) and continuing execution of the remaining goals in the network; failed goals may be placed into a holding bin for later re-elaboration and rescheduling;
- propagating the failure up the network by failing the goal that the failed goal is supporting (assuming the failed goal was instantiated as part of an elaboration);
- triggering the re-elaboration and rescheduling of the onboard activities; or
- safing the spacecraft, in the case of failure of a goal that jeopardizes mission success or safety.

In addition, the failure of a goal is normally reported via telemetry. The desired response to goal failure must be specified as part of the State Analysis.

Just as in goal elaboration and scheduling, the execution of a goal network is informed by the models of the system under control provided by systems engineers. We must specify the logic functions that dictate execution as part of the State Analysis process. The two primary execution-related functions that need to be specified are the logic associated with checking that the conditions on transition between x-goals are satisfied (post-conditions and pre-conditions), and the logic associated with checking that a currently-executing goal is being satisfied. The latter function can be a straightforward check for violation of the goal's state constraint, but is more generally defined as a check of whether the goal *is still satisfiable*, given our model of the system under control and our controller's or estimator's capabilities. As an example of the former, consider the pre-condition associated with an x-goal "Camera Temperature is between 10 and 20°C." An obvious choice would be to condition the start of this x-goal on the satisfaction of its state constraint. However, it is important to note that there is not necessarily a simple correspondence between an x-goal's pre-condition and its state constraint; we have the freedom to express the pre-condition in the form of any appropriate constraint on the state variable. In this case, we might choose to specify a slightly tighter range for our pre-condition, say, between 12 and 18°C. This might help improve our chances of satisfying the goals that were merged to produce the x-goal.

For "transition"-type goals, such as "Camera Temperature is transitioning to between 10 and 20°C," we frequently specify no pre-condition on the start of the goal, meaning that our goal should be achievable no matter what value our state variable has when the goal starts executing. An appropriate post-condition on this type of goal would be that the transition objective has been met, e.g., Camera Temperature is between 10 and 20°C. Of course, if our schedule allows insufficient time for the transition to occur, the goal will fail. Therefore, elaboration must assure the necessary transition time by adding temporal constraints, and any assumptions involved in computing this time (such as worst-case initial conditions) may need to be added as pre-conditions.

Off-nominal execution can result in goal failure (when a goal's state constraint is not satisfied) or temporal constraint violation (when a time point's time window expires before all outgoing goals are ready to start executing). The violation of a temporal constraint on a time point results in the automatic firing of that time point (even though not all of the post- and pre-conditions on the corresponding x-goal transitions were satisfied), which will likely result in the failure of one of the ensuing goals via state constraint violation. Thus, *all* execution failures manifest themselves as goal failures.

Firing a time point (i.e., satisfaction of its post- and pre-conditions) is an event. Therefore, as we have described thus far, goal network execution is event-driven (within temporal constraints), and all of the functions performed by estimators and controllers are paced by these events. We expect these functions to establish appropriate conditions for the subsequent goal, so execution proceeds in a controlled manner from one goal to the next. However, event-driven execution need not be confined to controllable state variables. For instance, consider the simple goal network in Fig. 22, which represents the assertion "External Temperature $\geq$10°C sometime in the next 12 hours; once this temperature is reached, the Heater Power State shall begin following a duty cycle of 30% On, 70% Off." Assuming we specify the pre-condition on the External Temperature goal as "External Temperature $\geq$10°C" and no pre-condition

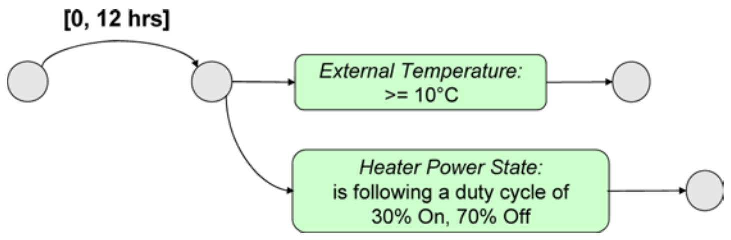

[0, 12 hrs]
External Temperature: >= 10°C
Heater Power State: is following a duty cycle of 30% On, 70% Off

Fig. 22 Goal network that illustrates event-driven execution using a goal on an uncontrollable state variable.

on the Heater Power State goal, the execution of this simple network will result in the system waiting for the External Temperature goal's pre-condition to be satisfied before firing the second time point. When the second time point fires, the heater begins operating at the duty cycle prescribed in the goal. If the expected level of External Temperature is not reached by the 12-hour deadline, the time point will fire, and the goal will fail, allowing us to take an appropriate response.

In summary, the products of State Analysis are used to inform the Mission Planning and Execution functions of the control system. This results in sequences that are verifiably executable, self-monitoring, robust during nominal operations, and reactive during off-nominal circumstances.

B. Estimation and Control

In the description of the State Analysis control architecture (Section II), we emphasized the importance of making a clear distinction between estimation and control, and we introduced estimators and controllers as the achievers of desired state. In this section we will briefly discuss how the model of the system under control is used to inform the algorithm development of the estimators and controllers, and how our paradigm of goal-driven execution impacts their design.

The use of models for estimation and control is not new—estimation and control theory is founded on the notion of using models of the system's state dynamics, measurements, and commands to compute estimates of current state and decide on appropriate control actions. This principle is commonly applied to the estimation and control of spacecraft position and attitude, structural dynamics, and temperature states, to name a few examples. In State Analysis, we simply demand that state models for all state variables of interest be documented, extending this paradigm across the whole system under control.

As discussed previously, state estimation is a process of interpreting information to achieve a requested quality of state knowledge, expressed in the form of a knowledge goal (see Fig. 23). This process involves not only collecting measurement data, but interpreting the measurements, filtering noisy data, resolving potentially conflicting information, using models to inform its determinations, and forming a single coherent notion of state knowledge for use across the system. Estimators update a state variable's value as well as its level of certainty. State control is a process of reacting to state information to generate commands that affect the state of the system under control in such a way as to satisfy a specified control goal (see Fig. 24). Controllers may react to the value of a state variable, or its level of certainty. Estimators and controllers may be invoked periodically, or in an event-driven fashion (e.g., conditioned on the arrival of new data or a change of estimated state), depending on the specific application.

State Analysis adopts the following architectural rules relating to estimators and controllers:

– Estimators are the only architectural components that can update state variables.
– Every state variable is updated by one (and only one) estimator, and controlled by at most one controller (some state variables are not controllable).
– An estimator can update multiple state variables.
– Estimators are the only components that can process measurements.
– Controllers are the only components that can issue commands to hardware adapters.
– A controller can control multiple state variables.
– A controller can issue commands to one or more hardware adapters.
– A hardware adapter can receive commands from at most one controller.

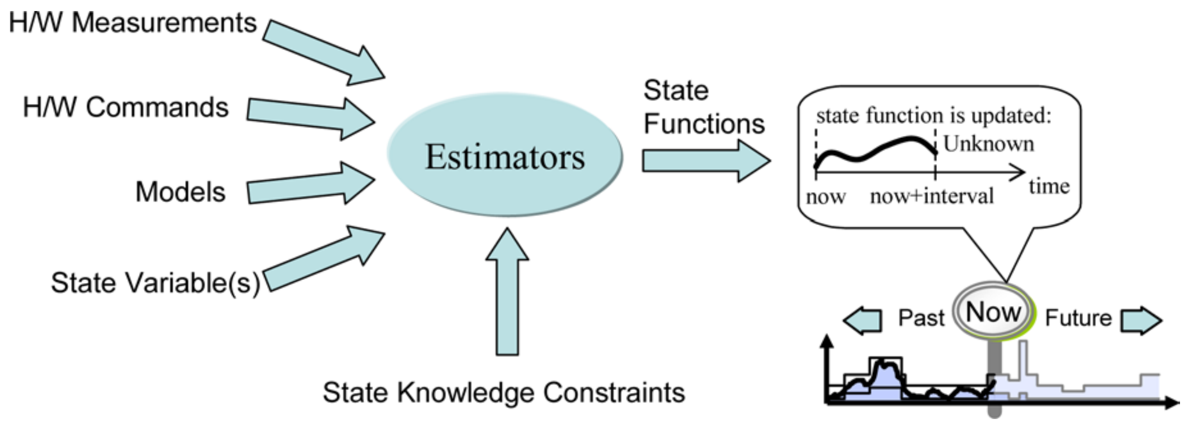


Fig. 23 Estimators use measurement, command and state information, along with models of the system under control, to satisfy state knowledge constraints (goals) by updating state in the form of state functions.

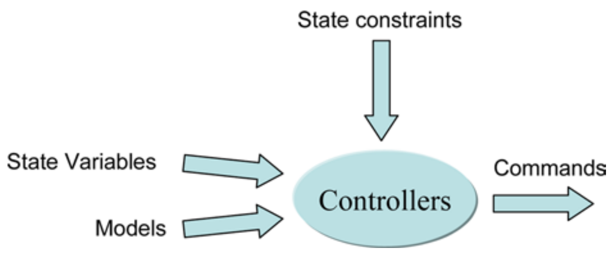


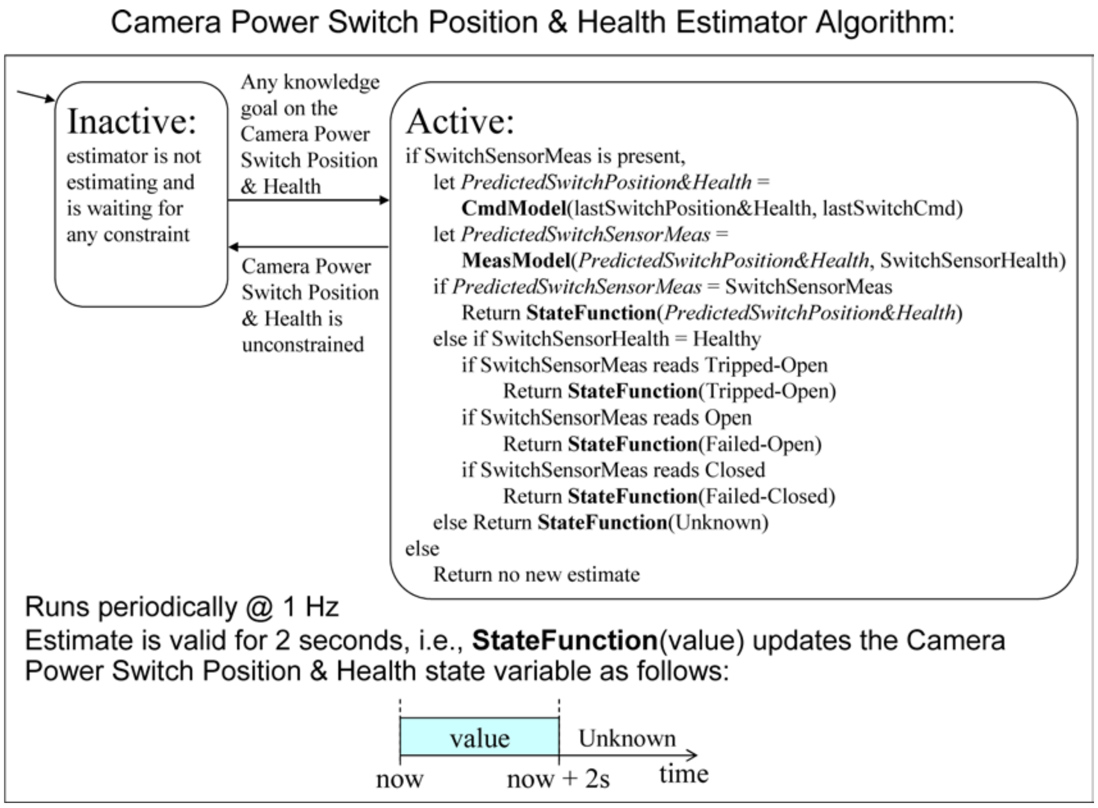
Fig. 24 Controllers use state information, along with models of the system under control, to satisfy state constraints (goals) by issuing commands to the hardware adapters.

- An estimator can receive measurements from zero or more hardware adapters (sometimes only indirect evidence is available).
- A hardware adapter can provide measurements to any number of estimators.
- An estimator or a controller can issue state constraints to one or more controllers (of other state variables) that have been delegated to it.
- Estimators and controllers can retrieve state information from state variables.

An important part of the State Analysis process is the specification of estimator and controller algorithms. These algorithms may be modal (e.g., state machines) or not; they may handle continuous values (e.g., Kalman filter estimators, linear controllers) or not; they may be of any design that is consistent with the model-based nature of State Analysis. We encourage, but do not require, that estimators and controllers make explicit use of the models we introduced in Section 4, but we presume that their translation into software will be as direct as possible (recall the basic principle from Section 1). State Analysis imposes no additional estimation or control issues beyond those driven by the problem itself, though it demands that estimators and controllers consider both nominal and off-nominal behavior of the system under control, and support degraded operations where possible.

1. Example

As an example of how models can be used in the design of estimators and controllers, Fig. 25 presents a simple estimator for our Camera Power Switch example from Section IV (this example assumes the combined state variable for Camera Power Switch Position and Health, as depicted in Fig. 11). In this case, we adopted a "pseudo-code" representation for our algorithm specification. This estimator is modal in nature; the estimator is initialized in 'inactive' mode, where it performs no updating of its state variable, and transitions into 'active' mode upon receipt of any knowledge goal on its state variable. We note that, in this case, the mode switching is entirely a function of the


Camera Power Switch Position & Health Estimator Algorithm:
Inactive:
estimator is not estimating and is waiting for any constraint
Any knowledge goal on the Camera Power Switch Position & Health
Camera Power Switch Position & Health is unconstrained
Active:
Runs periodically @ 1 Hz
Estimate is valid for 2 seconds, i.e., **StateFunction**(value) updates the Camera Power Switch Position & Health state variable as follows:
value
Unknown
now
now + 2s
time

```
if SwitchSensorMeas is present,
    let PredictedSwitchPosition&Health =
        CmdModel(lastSwitchPosition&Health, lastSwitchCmd)
    let PredictedSwitchSensorMeas =
        MeasModel(PredictedSwitchPosition&Health, SwitchSensorHealth)
    if PredictedSwitchSensorMeas = SwitchSensorMeas
        Return StateFunction(PredictedSwitchPosition&Health)
    else if SwitchSensorHealth = Healthy
        if SwitchSensorMeas reads Tripped-Open
            Return StateFunction(Tripped-Open)
        if SwitchSensorMeas reads Open
            Return StateFunction(Failed-Open)
        if SwitchSensorMeas reads Closed
            Return StateFunction(Failed-Closed)
    else Return StateFunction(Unknown)
else
    Return no new estimate
```

Fig. 25 Simple estimator for Camera Power Switch Position & Health. It uses the Camera Power Switch Sensor Measurement Model and Camera Power Switch Command Model, and it updates the Camera Power Switch Position & Health state variable using a State Function. [SV: state variable; HA: hardware adapter]

current goal that the estimator is working on. This is in keeping with another general principle in State Analysis, that estimators and controllers be as stateless as possible, such that the motive for all control system activity is explicitly captured in the state timelines. In particular, we only allow modal estimator and controller behavior to be conditioned on issued goals or state information of the system under control. Our estimator makes explicit use of the command and measurement models we introduced in Section IV, along with the previous estimate of the state, to produce a prediction of the expected measurement, which it compares to the actual measurement it received. This is analogous to a traditional "residual" computation in estimation theory, and illustrates a straightforward way that our models can be used in the control system.

Figure 26 shows a UML (Unified Modeling Language[7]) collaboration diagram excerpt for our example. The term collaboration diagram reflects the fact that a control system is a collection of software components "collaborating" to achieve a common purpose. Collaboration diagrams provide a map of the software component interconnections and information flow, which can be formally checked against the architecture rules described above, as a way to spot unusual features of the implementation that deserve scrutiny. These diagrams clearly show how State Analysis produces requirements on the software, which can be mapped directly into software components of a modular state-based architecture, such as MDS (see Section VII).

The construction of collaboration diagrams is informed by our state, measurement, and command models, and can be checked against them as another verification step. For example, our Camera Power Switch Position and Health estimator can access information on:

– measurements that are affected by the Camera Power Switch Position and Health (in this case, the Camera Power Switch Sensor measurement produced by the Camera Power Switch hardware adapter);

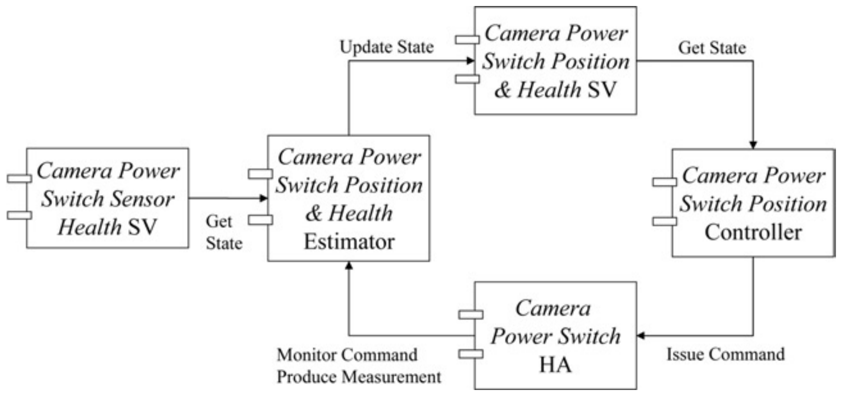

Camera Power Switch Position & Health SV
Update State
Get State
Camera Power Switch Sensor Health SV
Get State
Camera Power Switch Position & Health Estimator
Camera Power Switch Position Controller
Camera Power Switch HA
Monitor Command
Produce Measurement
Issue Command

Fig. 26 Collaboration diagram showing the estimation and control pattern for the Camera Power Switch Position and Health state variable.

– other state variables that affect the Camera Power Switch Sensor measurement (i.e., inputs to the measurement model; in this case, Camera Power Switch Sensor Health); and
– other state variables that affect the Camera Power Switch Position and Health (though the estimator design in this example does not use any state information from the Camera Power State, which is an affecting state variable, per Fig. 11).

Similarly, our Camera Power Switch Position controller needs information on other state variables that affect the results of the Camera Power Switch command (i.e., inputs to the command model). In this case, the only state information it requires is the Camera Power Switch Position and Health.

2. *Executable Models*

State Analysis makes models available for all state variables in the system under control. This opens up the possibility of using the state models explicitly during estimation and control. Endowing a spacecraft with the ability to automatically perform on-line reasoning about its modeled behavior has a number of potential benefits:

1. It relieves the software engineer of the responsibility of a priori encoding into estimators and controllers the complex set of low-level system interactions under a range of possible nominal and off-nominal situations. This error-prone task is instead delegated to a model-based reasoning engine that automatically diagnoses and plans courses of action at reactive time scales, based on models of the system under control, including its environment.
2. It facilitates software reuse by moving from the current practice of designing and implementing specialized estimators and controllers from scratch, to a paradigm of providing application-specific engineering models to a generic re-usable reasoning engine that can correctly synthesize state estimates and control actions.
3. It makes significant progress toward the Holy Grail of provably-correct behavior, by decomposing the challenging problem of validating the estimation and control software into two simpler problems: validating the systems engineering models and validating the reasoning engine.
4. It enhances robustness by transparently reasoning about all nominal and off-nominal behavior we have modeled. Thus, the reasoning engine is able to detect and respond to failures on-the-fly, within the control diamond loop.

This powerful idea, commonly referred to as "executable models," is being leveraged in the field of model-based autonomy. Model-based executives, like Livingstone[8] (which was flight-validated on the Deep Space 1 spacecraft), Livingstone2[9] and Titan[10], have been developed and demonstrated on a variety of mission scenarios and spacecraft designs.

The potential to exploit executable models has been in our sights from the beginning. The process described in this paper works directly toward this end, and we have begun to investigate how to leverage the principles of model-based autonomy in the context of the State Analysis. Our work to date in this area has shown significant promise, and we are pursuing ongoing work in integrating model-based execution capability into the MDS software architecture.

VI. Documenting the Models and Software Requirements

The model of the system under control that we produce during State Analysis compiles information traditionally documented in a variety of systems engineering artifacts, including the Hardware Functional Requirements, the Failure Modes and Effects Analysis, the Command Dictionary, the Telemetry Dictionary and the Hardware-Software Interface Control Document. Rather than break this information up into disparate artifacts, we capture all our model information in a State Database, which has been structured to prompt the State Analysis process. We use the same State Database to document the requirements on the control system that are produced by State Analysis, including goal specifications and elaborations, estimator and controller algorithms, and software component connectivity information (as depicted in collaboration diagrams). Figure 27 shows the main contents of the State Database, and a sampling of the documents and products that can be produced from the database.

The State Database is shared, central, and globally accessible to promote consistency. It is accessible by a variety of tools, including a graphical client tool that provides multiple interfaces for access to State Analysis data. This tool provides multiple convenient user interfaces to the State Database data, including a text-based record editor (which allows direct access to the models and requirements) and a diagram editor (which currently provides views on the database via State Effects Diagrams, but will be augmented in the near future to include collaboration diagrams and graphical modeling representations like StateCharts). Our client tool is designed to be capable of generating a variety of reports from the information it contains, including the set of documents described above. The State Database thus provides systems engineers with a tool that consolidates their system and software requirements in a single place, and allows them to inspect and review this information in whatever form is most appropriate.

VII. The Mission Data System Software Architecture

MDS is an embedded software architecture, currently under development at the Jet Propulsion Laboratory (JPL). Its overarching goal is to provide a multi-mission information and control architecture for robotic exploration spacecraft, that will be used in all aspects of a mission: from development and testing to flight and ground operations. In the process of achieving this ambitious goal, the MDS team has rethought the traditional mission software lifecycle. MDS acknowledges the intimate coupling between software and systems engineering by leveraging the State Analysis

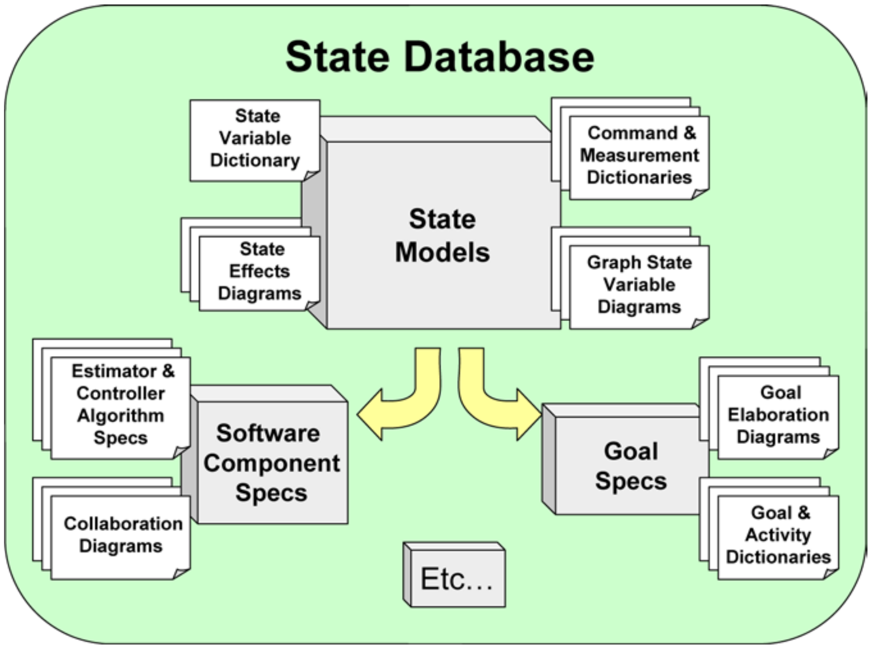

State Database
State Variable Dictionary
State Effects Diagrams
State Models
Command & Measurement Dictionaries
Graph State Variable Diagrams
Estimator & Controller Algorithm Specs
Collaboration Diagrams
Software Component Specs
Goal Specs
Goal Elaboration Diagrams
Goal & Activity Dictionaries
Etc…

Fig. 27 State Analysis products captured in the State Database.

methodology. The regular structure of State Analysis is replicated in the MDS architecture, with every State Analysis product having a direct counterpart in the software implementation.

The mapping of State Analysis products to software is accomplished via a component architecture. Each state variable, estimator, controller, and hardware adapter is embodied as a component. State Analysis defines the interconnection topology among these components according to the canonical patterns and standard interfaces described in this paper; it provides the required interface details through the definition of state functions, measurements, commands, and goals; it provides the methods needed for planning, scheduling and execution; and it defines the functionality of each component to accomplish the desired intent. The component architecture helps to assure that the system is constructed in accordance with the State Analysis requirements, it aids modular reuse, and it provides support for a variety of software engineering and analysis issues.

The component architecture is part of a much larger framework of MDS software, wherein each of the concepts in State Analysis is endowed with a core implementation that provides common features, interfaces, and so on. These are further constructed upon an in-depth support structure of additional layered, modular frameworks supporting low-level services, extensive libraries in math and physics (e.g., units), data management and transport functions, high-level automated planning and scheduling engines, and so on. There are nearly thirty distinct core framework packages. Moreover, as various adaptations of these frameworks for particular projects occur, it is our intent to factor common elements into an additional set of engineering and science discipline framework packages for even greater reuse among projects.

The formally coherent structure shared by State Analysis and the MDS architecture has enabled an unprecedented level of coordination and control of the development process. Requirements are cleanly partitioned and traceable directly to implementation, making it easy to track and manage each step in the development process. Verification and validation exploits the same explicit structure, as well as the objective specification of each system element and the overt declaration of success criteria at all levels of operation.

We have also taken advantage of this structure in an iterative incremental development process with workflow and configuration management tools specifically built around State Analysis elements and spanning the entire development life cycle. Metrics gathered from this process are detailed and directly attributable to particular design elements, enabling far better feed-forward to future development efforts.

A C++ implementation of MDS has been demonstrated on multiple hardware platforms, including the Rocky7 and Rocky8 rovers at JPL. In addition, an MDS adaptation is currently being developed for the Entry, Descent and Landing (EDL) stage of the Mars Science Laboratory spacecraft, scheduled for launch in 2009. This flight software prototype runs in a workstation environment, against a simulation of the EDL scenario. A simpler Java implementation of the MDS architecture, called GoldenGate,[11] has also been demonstrated on the Rocky7 rover.

VIII. Conclusion

State Analysis is a Systems Engineering methodology that improves on the current state-of-the-practice. It does so by leveraging a state-based control architecture to produce requirements on system and software design in the form of explicit models of system behavior. This provides a common language for systems and software engineers to communicate, and thus bridges the usual gap between software requirements and software implementation. This provides a powerful framework for engineering robust embedded systems, and also promotes the infusion of advanced model-based autonomy technologies. Therefore, we believe State Analysis is a systems engineering methodology for today's complex systems that can carry us well into the future.

Acknowledgments

The work described in this paper was performed at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. We wish to thank the rest of the Mission Data System development team, and the Mars Science Laboratory mission personnel who have participated in the maturation of the State Analysis methodology and tools. In particular, we would like to acknowledge Daniel Dvorak, Russell Knight, Stephen Peters, George Rinker and Marcel Schoppers, the other co-instructors of the State Analysis training class, from which the material in this paper was inspired.

References

[1]Dvorak, D., Rasmussen, R., Reeves, G., and Sacks, A., "Software architecture themes in JPL's Mission Data System," *Proceedings of the AIAA Guidance, Navigation, and Control Conference*, number AIAA-99-4553, 1999.

[2]Dvorak, D., Rasmussen, R., and Starbird, T., "State Knowledge Representation in the Mission Data System," *Proceedings of the IEEE Aerospace Conference*, 2002.

[3]Bennett, M., and Rasmussen, R., "Modeling Relationships Using Graph State Variables," *Proceedings of the IEEE Aerospace Conference*, 2002.

[4]Harel, D., "Statecharts: A visual formulation for complex systems," *Science of Computer Programming*, Vol. 8, No. 3, 1987, pp. 231–274.

[5]Barrett, A., Knight, R., Morris, R., and Rasmussen, R., "Mission Planning and Execution Within the Mission Data System," *Proceedings of the International Workshop on Planning and Scheduling for Space*, 2004.

[6]Chung, S., *Decomposed symbolic approach to reactive planning*, S.M. Thesis, Massachusetts Institute of Technology, Department of Aeronautics and Astronautics, Cambridge, MA, 2003.

[7]Booch, G., Rumbaugh, J., and Jacobson, I., *The Unified Modeling Language User Guide*, Addison Wesley Longman, Inc., 1999.

[8]Williams, B. C., and Nayak, P., "A model-based approach to reactive self-configuring systems," *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI-96)*, Vol. 2, 1996, pp. 971–978.

[9]Kurien, J. and Nayak, P. "Back to the future for consistency-based trajectory tracking," *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI-02)*, 2000, pp. 370–377.

[10]Williams, B. C., Ingham, M., Chung, S., and Elliott, P., "Model-based programming of intelligent embedded systems and robotic space explorers," *Proceedings of the IEEE*, Vol. 91, No. 1, 2003, pp. 212–337.

[11]Dvorak, D. et al., "Project Golden Gate: Towards Real-Time Java in Space Missions," *Proceedings of the 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2004)*, 2004.