# IAC-04-IAF-U.3.A.05

# GENERATING REQUIREMENTS FOR COMPLEX EMBEDDED SYSTEMS USING STATE ANALYSIS

**Michel D. Ingham, Robert D. Rasmussen,**
**Matthew B. Bennett, Alex C. Moncada**
**Jet Propulsion Laboratory, California Institute of Technology, Pasadena, USA**
**{michel.d.ingham, robert.d.rasmussen, matthew.b.bennett, alex.c.moncada}@jpl.nasa.gov**

**ABSTRACT**

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, using representative spacecraft examples.

## 1. 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 spacecraft design is reaching a threshold of complexity 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 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*.
- 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 success-

ful 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.
- 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. The State Analysis methodology is complemented by a database tool that facilitates model-based software requirements capture.

**Paper Outline**

In this paper, we discuss the state-based control architecture that provides the framework for State Analysis (Section 2), we emphasize the central notion of state, which lies at the core of the architecture (Section 3), we present the process of capturing requirements on the system under control in the form of models (Section 4), and we illustrate how these models are used in the design of a control system (Section 5). We then discuss the database tool used for documenting the models and requirements (Section 6). Finally, we describe the Mission Data System (MDS), a modular multi-mission software framework that leverages the State Analysis methodology (Section 7).

## 2. 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", see Figure 1).

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. We discuss the representation of state in more detail in Section 3.

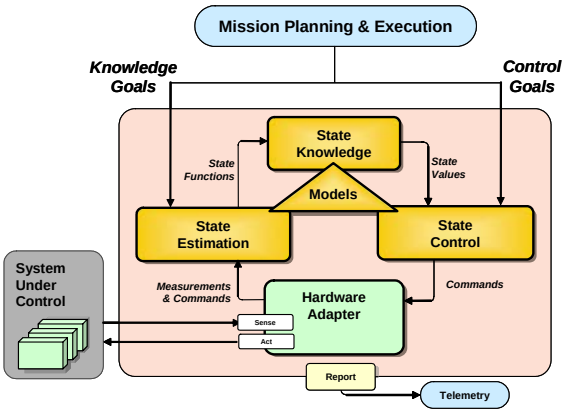


Figure 1: The state-based control architecture.

- *State estimation is separate from state control*: Estimation and control are coupled only through state variables. 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 provide the sole interface between the system under control and the control system*: They form the boundary of our state architecture, provide all the measurement and command abstractions used for control and estimation, and are responsible for translating and managing raw hardware input and output.
- *Models are ubiquitous throughout the architecture*: Models are used both for execution (estimating and controlling state) and higher-level planning (e.g., resource management). State Analysis requires that the models be documented explicitly, in whatever form is most convenient for the given application. In Section 4, 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. In Section 5, 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 7.

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.

## 3. STATE KNOWLEDGE REPRESENTATION

As discussed in the previous section, State Analysis is founded upon a state-based control architecture, where state is a representation of 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]

### 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 this knowledge of state and its behavior is complete – that no other information is required to control a system. Consequently, State Analysis adopts a broader definition of state than traditional control theory, for example: in addition to considering the position and attitude (and corresponding rates) of a spacecraft to be defined as state, we would also include 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 and health;
- resource levels (e.g., propellant; volatile and non-volatile memory);
- temperatures and pressures;
- environmental states (e.g., motions of celestial bodies and solar flux);
- static states about which we may want to refine our knowledge (e.g., dry mass of a spacecraft);
- parameters (e.g., instrument scale factors and biases, structural alignments, and sensor noise levels); and
- states of data collections, including the conditions under which the data 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. 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. 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 1).

### 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. 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, all state variables represent state as a piece-wise continuous function of 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 temporally-continuous State Functions (see Figure 1) 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 (see Figure 2), which are a complete record of a system's history ("complete" to the extent that they capture everything the control system has chosen to remember about the state, subject to storage limitations). 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.

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.

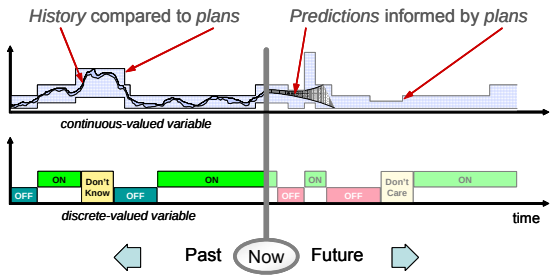


Figure 2: Timelines are used to capture state knowledge (past estimates and future predictions) and intent (past and future constraints on state).

Because of our adoption of a temporally-continuous 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).

### Representing Uncertainty

In a real system, we never really know 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. 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.

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, missing evidence, and 'old' evidence (see Figure 3). 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.

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.

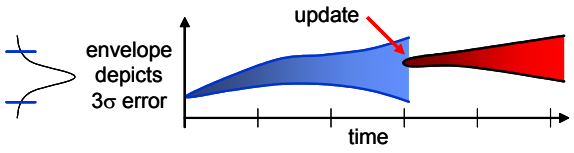


Figure 3: The level of uncertainty associated with a state estimate generally grows over time, and can decrease with the receipt of additional evidence by the estimator.

## 4. MODELING THE SYSTEM UNDER CONTROL

State Analysis provides a methodology that allows us to develop 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 (if at all), 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).

This model describes the behavior of all hardware and any software elements in the system under control, as well as the behavior of any external systems that affect the state of the system under control (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 5.

### 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, the reason should be documented.

**Example**

We now present a simple example to illustrate this iterative process. Consider the problem of preparing a rover's navigation camera to take picture (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 (step 3): the Camera Power State = 0 Watt if the Camera Power Switch Position is Open (or Tripped-Open) or if the Power Bus Voltage is less than `threshold`; otherwise, Camera Power State = 10 Watts if Camera Health = Healthy, or greater if Camera Health = Short-Circuit. 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. As far as model representation is concerned, State Analysis is flexible. We provide systems engineers with broad latitude to capture models in a form that is most convenient for their specific application.

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.

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 may choose to specify the state model for this state variable in the form of a StateChart[3] (step 3), which is a convenient representation for discrete state models that are fairly commonly used by systems engineers. This StateChart would show all nominal and off-nominal transitions between values of this state variable. The behavior in this state model would be affected by two other state variables, 'Camera Power State' (a load overcurrent condition can cause the switch to trip open) and 'Camera Power Switch Health' (nominal transitions between switch states require that the switch be healthy, i.e., not stuck).

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 the switch sensor health is Healthy, the measurement returns the true switch position; otherwise (i.e., switch sensor health is Stuck-Reading-Open, Stuck-Reading-Closed, or Stuck-Reading-Tripped-Open), the measurement returns the stuck reading, independent of the true switch position.

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 would also allow for the possibility that the sensor could exhibit other failure modes, such as intermittent random readings. Clearly, 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 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.

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): if the current switch position is Open [Closed] and the power switch health is Healthy, a Close-cmd [Open-cmd] results in the switch position transitioning to Closed [Open]; if the current switch position is Tripped-Open and the power switch health is Healthy, an Open-cmd results in the switch position transitioning to Open, whereas a Close-cmd results in no change in the switch position. Command models are used to describe *instantaneous* changes of state; we ascribe cascading effects and delayed behavior to the state model.

Figure 4 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. Measurements are depicted on the State Effects Diagram as triangles, with incoming effect arrows from all state variables that appear in the measurement model. Commands are depicted 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).

We have just stepped through two iterations of the modeling process. There are state variables in Figure 4 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. In the following section, we discuss how the models are used to design software.

## 5. 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 design of the control system. In this section, we outline how state, measurement and command models are used to inform the design of the control system. In particular, we discuss the design of the Mission Planning and Execution functions, and the Estimation and Control algorithms (recall Figure 1).

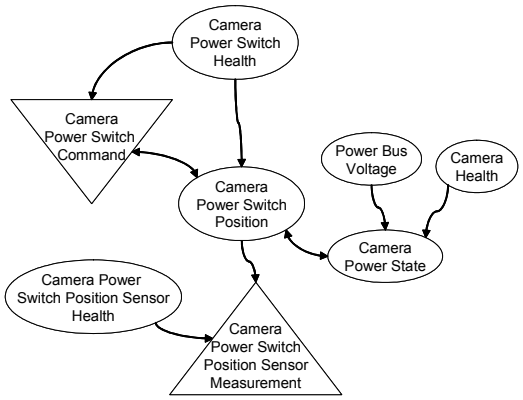


Figure 4: State Effects Diagram after two iterations of the modeling process.

### Mission Planning and Execution:

As mentioned in Section 2, one of the key features of State Analysis is that it emphasizes *goal-directed closed-loop operation*. The control architecture in Figure 1 includes a Mission Planning and Execution 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-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. It empowers the spacecraft to accommodate most surprises without the need for ground intervention and demonstrates reliability, independent of our knowledge of the environment. Recent space missions, including the Cassini and Mars Exploration Rover spacecraft, have demonstrated a fair amount of goal-directed behavior. 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.

Goals:

In State Analysis, a goal is defined as a *constraint on the value history of a state variable over a time interval*. 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 (see Figure 5). Spacecraft operators specify instantiations of the goal types in the *goal dictionary* to construct activity plans for accomplishing mission objectives.

A goal is expressed as an assertion whose success/failure can be evaluated with respect to its state variable's value history (state timeline). It is important to distinguish between goals and commands. For example, "At 2:00pm, 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? Clearly, we would not have achieved our underlying objective of initiating the heating of the camera, even though we did issue the close-switch command as specified. 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.

Here are some examples of valid goals:

1. "Camera Temperature is between 10 and 20 degrees Celsius from 2:00pm to 3:00pm" (*control goal* that specifies a constraint on state value, to be maintained by controller).
2. "Camera Temperature is transitioning to be between 10 and 20 degrees Celsius by 2:00 pm" (*transitional* control goal that achieves the appropriate precondition for goal #1).
3. "Camera Temperature standard deviation is less than 0.5 degree Celsius from 1:00 pm until 5:00 pm" (*knowledge goal* that specifies a constraint on quality of state knowledge, to be maintained by estimator).
4. "Camera Temperature mean value, plus or minus 3-sigma, is in the range 10-20 degrees Celsius [$10 \leq \text{mean} - 3\sigma \leq \text{mean} + 3\sigma \leq 20$], from 2:00 pm to 3:00 pm" (*inseparably-combined control and knowledge goal*, specifying constraints on *both* state value and quality of knowledge).
5. "Camera Temperature measurement data collection state contains at least one measurement less than 10 seconds old, from 1:00 pm until 5:00 pm" (*data goal*, specifying a constraint on the state of a data collection).

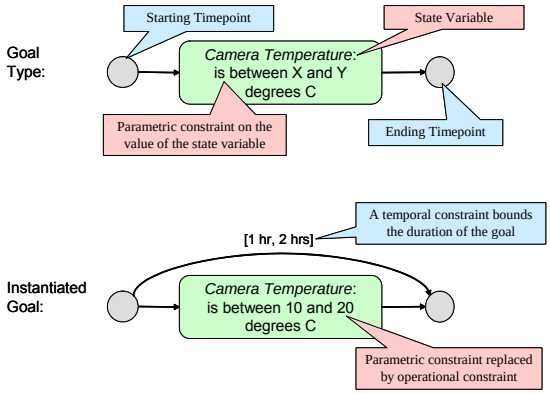


Figure 5: The anatomy of a goal type and an instantiated goal. Every goal has a starting timepoint and an ending timepoint. A goal can be instantiated with a flexible temporal constraint on its duration, indicated by an arrow from its starting to its ending timepoint, labeled with a [min, max] duration window.

Goal Elaborations:

As we discussed in Section 4, 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 control 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 states 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 four rules:

1. A goal on a state variable may elaborate into concurrent control goals on directly affecting state variables.
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 can elaborate into preceding goals (typically on the same state variable). For example, a "maintenance" goal on a state variable may elaborate to a preceding transitional goal on the same state variable.

We note that goal elaborations are defined locally for each goal by considering only direct effects, that is, effects of state variables that are only a single step away in the State Effects Diagram.

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 Figure 4. Consider the following goal on the Camera Power State: "Camera Power State is equal to 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 Figure 6.

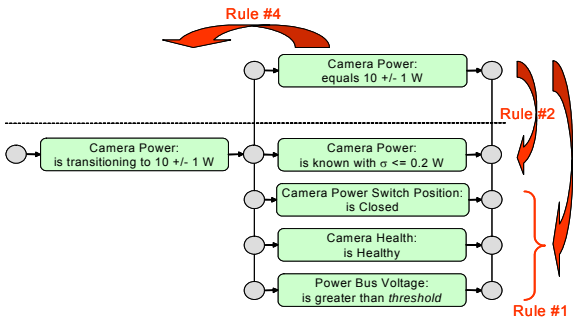


Figure 6: The elaboration for the "Camera Power State equals 10 ± 1 Watts" goal. Per Rule #1, our control goal elaborates into concurrent supporting control goals on the affecting state variables. Per Rule #2, it elaborates into a knowledge goal that asserts that the uncertainty must be limited to $\sigma \leq 0.2$ Watts. Per Rule #4, our 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.

Goal elaboration is an iterative process, so supporting goals that appear in an elaboration are, in turn, elaborated. The elaborations chain together to encompass the full set of relevant state variable interactions. 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. Loops in the elaboration chain are 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 individual elaborations and schedule them; this is the subject of the next subsection.

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[4] as a potential solution to this problem.

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. In State Analysis, the mechanism we use 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: an affecting state variable (e.g., wheel rotation) can delegate to an affected state variable (e.g., rover position and heading). Run-time delegation is enabled via elaboration, where the affecting state variable authorizes the affected state variable to send it reactive goals "on-the-fly," within allocations established at elaboration time.

Goal Network Scheduling & Execution:

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;
- merge these elaborated goals into the current goal network, which includes all background goals (capturing flight rules and constraints) and previously-scheduled activities;
- 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 automated, iterative search process that may require backtracking, and heuristics are used for efficiency to guide the search (details on this process have been previously published[5]). This process 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 during State Analysis. For instance, we must specify the logic associated with merging multiple concurrent goals on a given state variable. This corresponds to an intersection operation performed on the goals' state constraints. The result of this merging of constraints is called an *executable goal*, or *x-goal*. X-goals reside on state timelines, and capture intent on state (recall Figure 2).

State Analysis also specifies the logic used to propagate state effects across the system and project state into the future. This logic is derived directly from the state models described in Section 4. This projection logic provides a mechanism for generalized resource management for the system under control.

Finally, we must also specify the logic associated with checking the consistency of the resulting x-goal timelines. This involves checking each x-goal for achievability, and checking that each consecutive pair of x-goals is compatible (i.e., that the transition between x-goals is achievable).

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.

Once the goal network has been fully elaborated and scheduled, it is ready to be executed.[5] 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 it is appropriate to transition from executing one x-goal to the next x-goal on the timeline, and the logic associated with checking for violation of a goal's state constraint ("goal failure").

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

**Estimation and Control:**

In the description of the State Analysis control architecture (Section 2), 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.

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 command effects 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 spacecraft system.

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. 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. 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 hardware 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.
- 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), continuous (e.g., Kalman filter estimators, linear controllers), or any other 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.

Figure 7 shows a UML (Unified Modeling Language[6]) *collaboration diagram* excerpt for our Camera Power Switch example from Section 4 (the term collaboration diagram reflects the fact that a control system is a collection of software components "collaborating" to achieve a common purpose). These diagrams provide a map of the software component interconnections and information flow. They 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 7).

The construction of collaboration diagrams is informed by our state, measurement and command models. For example, our Camera Power Switch Position estimator can access:

- measurements that are affected by the Camera Power Switch Position (in this case, the Camera Power Switch Sensor measurement);
- 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 (in this case, our estimator uses knowledge of Camera Power Switch Health, but not Camera Power State).

Similarly, our Camera Power Switch Position controller needs information on other state variables that affect the results of the switch command (i.e., inputs to the command model; in this case, Camera Power Switch Health).

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. This powerful idea, commonly referred to as "executable models", is being leveraged in the field of *model-based autonomy*. Model-based executives, like Livingstone[7] (which was flight-validated on the Deep Space 1 spacecraft), Livingstone2[8] and Titan[9], have been developed and demonstrated on a variety of mission scenarios and spacecraft designs.

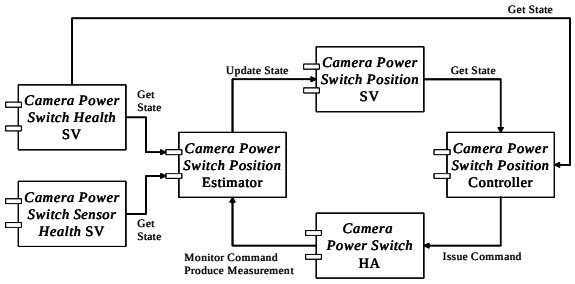


Figure 7: Collaboration diagram showing the estimation and control pattern for the Camera Power Switch Position state variable. [SV: state variable; HA: hardware adapter]

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 MDS.

## 6. 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).

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, including a text-based record editor and a diagram editor. It 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.

## 7. 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. 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. This mapping 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, 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 supports modular reuse, and helps to assure that the system is constructed in accordance with the State Analysis requirements.

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 currently runs in a workstation environment, against a simulation of the EDL scenario. A simpler Java implementation of the MDS architecture, called GoldenGate,[10] has also been demonstrated on the Rocky7 rover.

## 8. 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.

## REFERENCES

1. D. Dvorak, R. Rasmussen, G. Reeves, and A. Sacks, “Software architecture themes in JPL's Mission Data System,” *Proceedings of the AIAA Guidance, Navigation, and Control Conference*, number AIAA-99-4553, 1999.
2. D. Dvorak, R. Rasmussen, and T. Starbird, “State Knowledge Representation in the Mission Data System,” *Proceedings of the IEEE Aerospace Conference*, 2002.
3. D. Harel, “Statecharts: A visual formulation for complex systems,” *Science of Computer Programming*, 8(3):231-274, 1987.
4. S. Chung, *Decomposed symbolic approach to reactive planning*, S.M. Thesis, MIT Dept. of Aeronautics and Astronautics, Cambridge, MA, 2003.
5. A. Barrett, R. Knight, R. Morris, and R. Rasmussen, “Mission Planning and Execution Within the Mission Data System,” *Proceedings of the International Workshop on Planning and Scheduling for Space*, 2004.
6. G. Booch, J. Rumbaugh, and I. Jacobsen, *The Unified Modeling Language User Guide*, Addison Wesley Longman, Inc., 1999.
7. B.C. Williams and P. Nayak, “A model-based approach to reactive self-configuring systems,” *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI-96)*, volume 2, pages 971-978, 1996.
8. J. Kurien and P. Nayak, “Back to the future for consistency-based trajectory tracking,” *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI-02)*, pages 370-377, 2000.
9. B.C. Williams, M. Ingham, S. Chung, and P. Elliott, “Model-based programming of intelligent embedded systems and robotic space explorers,” *Proceedings of the IEEE*, 91(1):212-237, 2003.
10. D. Dvorak, 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.