A Model-Based Requirements Database Tool for Complex Embedded Systems

Matthew B. Bennett, Robert D. Rasmussen, Michel D. Ingham
Jet Propulsion Laboratory, 4800 Oak Grove Drive, Pasadena, CA 91109 USA
{matthew.b.bennett, robert.d.rasmussen, michel.d.ingham}@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 capture accurately 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 (1) how requirements for complex aerospace systems can be developed using State Analysis, (2) how these requirements inform the design of the system software, and (3) how this process has been aided through a State Analysis Database (SDB) and supporting multi-platform client. The SDB provides a productive, collaborative development environment for State Analysis that is shared by both systems and software engineers.

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 also 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 capture accurately 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*[11], and a tool to facilitate the process, called the *State Analysis Database*.

State Analysis addresses the above 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 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.
- 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 the State Analysis Database tool that captures explicit models of system behavior and software requirements structured to be consistent with the models. The model and requirements that we produce during State Analysis compiles information traditionally documented in a variety of systems engineering artifacts, including Hardware Functional Requirements, Failure Modes and Effects Analyses, Command Dictionaries, Telemetry Dictionaries and Hardware-Software Interface Control Documents. Rather than break this information up into disparate artifacts, we capture all our model information in a State Analysis Database, which has been structured to prompt the State Analysis process. The tool design promotes state discovery, and insures that the models and other requirement artifacts are consistent with the State Analysis methodology and state-based architecture. Further, the database schema has been developed to map directly into requirements on adaptations of the Mission Data System (MDS) state-based control system software frameworks[1]. In these ways, the database ensures a rigorous project development, from requirements analysis, through software design and implementation, to verification and validation.

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 State Analysis 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).

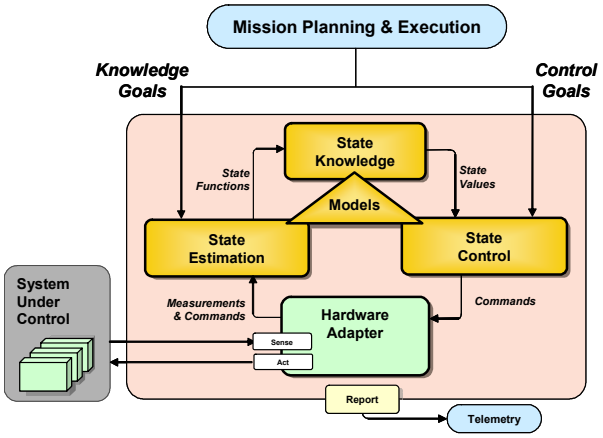

Figure 1: The state-based control architecture.

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.
- *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 hardware in 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 for both 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.

This control architecture informed the design the State Analysis Database. There are

elements in the database for state variables, separate estimators and controllers, hardware adapters, explicit models and goals. State database entries for these elements are used as specifications for the implementation of elements in the real time control architecture. There is a one-to-one mapping from these entries into an implementation built from components in a modular software architecture such as MDS.

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 states evolve. The state of a system and our knowledge of that state are not the same thing. The real state is complex. One could describe a state all the way down to the atomic level. Nevertheless, 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 the 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. Nevertheless, 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).

Definitions for all of the states in the system under control are identified as part of the state discovery process are and are entered into the State Analysis Database. In subsequent steps of the

state discovery process, the specifications for how the control system represents states as state variables are also entered into the database.

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.

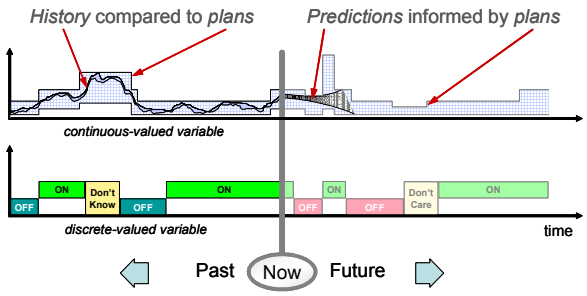

Figure 2: Timelines are used to capture state knowledge (past estimates and future predictions) and intent (past and future).

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.

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. This avoids the possibility of returning inconsistent values for a state and its derivative.

Definitions for state variables and their state functions are captured in the State Analysis Database.

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.

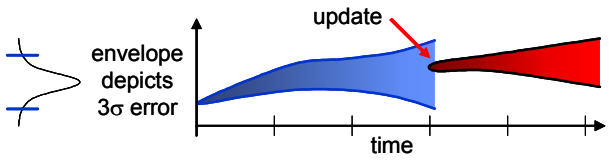

Figure 3: The level of uncertainty associated with a state estimate generally grows over time, and can decrease with the receipt of additional evidence by the estimator.

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.

The State Analysis Database captures the representation of uncertainty within the definition of a state function. A state function is defined to produce a state value as a function of time, where each state value contains a representation of the state's 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.

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.

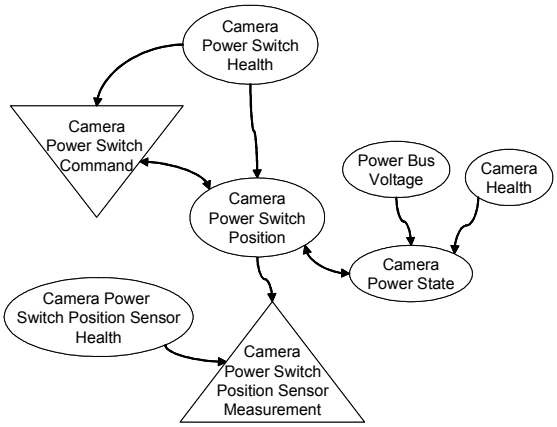

Figure 4: State Effects Diagram after two iterations of the modeling process.

Our model of the system under control is composed of:

- *State Models* describing how each physical 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 physical states in the system under control; and
- *Command Models* describing how

physical 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 overall physical state of the system under control (e.g., environmental effects). Figure 4 shows a graphical representation of states and effects, which we call a State Effects Diagram. This provides a convenient view of the physical state in the system under control, and the effects between the physical states. Measurements are depicted on the State Effects Diagram as triangles, with incoming effect arrows from all physical states 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 example), and incoming arrows from the physical states that have an impact on the effects of the command (Camera Power Switch Position and Camera Power Switch Health). It is important to note that these models are expressed in terms of physical 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 is 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.

Each of the elements discovered above by the modeling process is captured in the State Analysis Database: state variables and their representation, state models, measurements and their representation, measurement models, commands and their representation, and command models (captured within state models).

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.

Detailed examples of the application of this methodology can be found in reference 11.

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.

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).

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 *goals*, each with parametric state constraints and unspecified temporal constraints. Spacecraft operators specify instantiations of the goals from the *goal dictionary* and temporal constraints 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.

An example of a valid goal is "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).

A systems engineer defines goals in the State Analysis Database. Each goal in the database is defined to constrain an associated state variable. The State Analysis Database tool produces a goal dictionary from the database that can be used by operators to construct activity plans.

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 using 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.

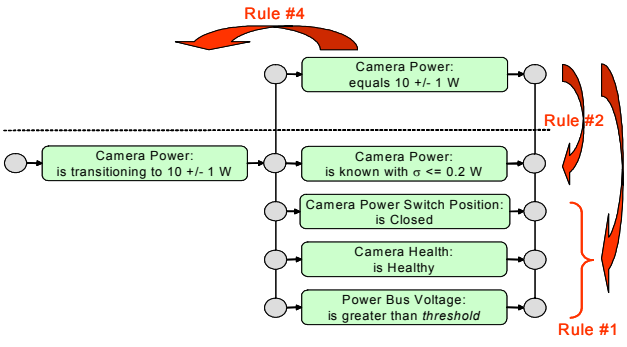

Figure 5: The elaboration for the "Camera Power State equals 10 ± 1 Watts" goal.

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.

Goal elaboration is an iterative process, so supporting goals that appear in an elaboration are, in turn, elaborated. The elaborations are chained 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. Elaborations produced by a systems engineer are captured in the State Analysis database as well. Each goal on a state variable may elaborate to one or more supporting goals, and that relationship is documented in the database.

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.

A systems engineer can specify a goal as a delegation goal in the State Analysis Database. The delegate and delegating roles are determined by elaborations in the state database as specified by the systems engineer and the following rule. If a delegation goal is a supporting goal in an elaboration, then the supporting goal's controller is delegated to the parent goal's estimator or controller.

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 uses heuristics 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.

The State Analysis database contains all goal scheduling logic specified by the systems engineers. Goal merge logic is captured on relationships between goal types, which are generic for all state variables of a certain type. The logic for propagating state effects and projecting state into the future is captured in specifications of projection methods for each goal type on a particular state variable and its achiever types. The database also stores achievability and transition achievability logic for x-goals in relationships between state variables, achievers and goals.

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").

The State Analysis Database provides a place to hold this logic. The first is specified as a method description stored in the relationship between a goal type and a controller or estimator type and within the goal type itself if it has no achievers. The second is stored in the definition of a goal type, because a goal represents a state constraint. The database holds other kinds of scheduling logic as well.

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 at most one estimator (some state variables are not "estimated" from evidence but are simply functions of other state variables).
- Every state variable is 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.

Each architectural rule is enforced by the State Analysis Database in one of two ways. In the first way, the structure of the database schema enforces the rule. For example, a state variable only has a single slot for an estimator, thus the rule for each state variable to have at most one estimator can never be broken. The second way is a consistency check run by a script to verify that the rule holds true. This latter method gives the systems engineer some flexibility in doing the state analysis, until he or she commits the analysis artifacts for checking.

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).

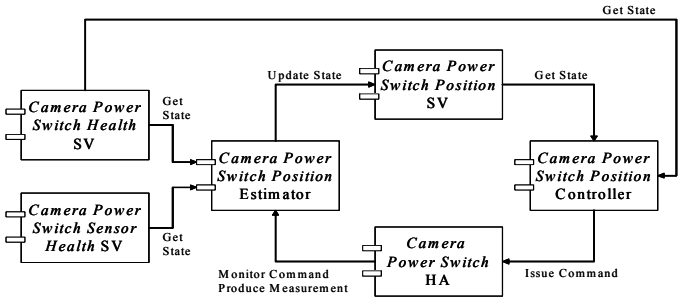

Figure 6: Collaboration diagram showing the estimation and control pattern for a Camera Power Switch Position state variable. [SV: state variable; HA: hardware adapter]

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.

Systems engineers use the database to capture requirements on estimators and controller algorithms. Each estimator or controller is represented as a record in the state database, along with other tables that describe the relationships with other architectural elements. For example, if an estimator consults a state variable as part of its estimation algorithm, there is a relationship stored in the database that indicates this.

Figure 6 shows a UML (Unified Modeling Language[6]) *collaboration diagram* example (the term collaboration diagram reflects the fact that a control system is a collection of software components "collaborating" to achieve a common purpose). Such a diagram provides a map of the software component interconnections and information flow. It shows 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 by a systems engineer can be informed by the use of state, measurement, and command models by estimators and controllers.

The State Analysis Database tool will be able to produce collaboration diagrams that are graphical representations of the interconnections stored as relationships in the database.

6. DOCUMENTING 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 Hardware Functional Requirements, Failure Modes and Effects Analyses, Command Dictionaries, Telemetry Dictionaries and Hardware-Software Interface Control Documents. Rather than break this information up into disparate artifacts, we capture all our model information in the *State Analysis Database*, which has been structured to prompt engineers to follow the State Analysis process.

As discussed in Sections 4 and 5, we use the State Analysis Database to not only document behavior requirements on the system under control, but to also document requirements on the control system: goal specifications and elaborations, estimator and controller algorithms, and software component connectivity information (as depicted in collaboration diagrams). This allows us to more easily validate and check the requirements on the control system for consistency with state effects models of the environment and hardware in the system under control. In fact, the structure of the State Analysis Database schema has been developed to enable enforcement of the correct relationships between models, software specifications, and each other. Thus, the schema prevents a class of engineering errors, and provides a guide for

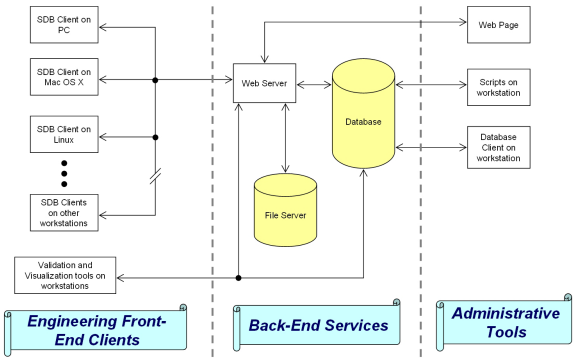

Figure 7: State Analysis Database distributed system design.

doing a complete and consistent engineering analysis.

In addition, the tool has the capability to produce from the database artifacts and documents like those enumerated above. These are inherently consistent with one another because the tool derives them from the same modeling and software specification information in the database.

Figure 7 shows that the State Analysis Database has a client-server distributed system design. A single central database is the repository for requirements and models. The database is hosted using a commercial product. Communication with clients and other tools is through SQL queries and other industry standard mechanisms so that the database can be easily re-hosted on a variety of systems. The database can be accessed through the internet using the HTTP protocol – a database web server services HTTP database requests from clients and tools.

The clients and tools allow for:

1. Capturing and inspecting State Analysis data using graphical interfaces;
2. Visualizing and validating the database contents;
3. Uploading, archiving, and viewing ancillary files that are part of the analysis; and
4. Configuration managing and administrating the database.

Database Schema. We have designed the database schema to reflect the formal state analysis process. Each of the kinds of architectural elements that must be modeled or specified in state analysis has a corresponding table in the database. Relationships between architectural elements are captured as references to related elements in tables, or through the use of linking tables between tables. The design was guided by the nature of each kind of relationship and the desire for the schema to enforce architectural rules where possible and reasonable, rather than having to rely completely on external consistency checking scripts.

The elements of the state analysis that pertain to specifications on the control system (estimators, controllers, etc) are in fact specifications for adaptations of state-based software frameworks such as MDS (described in Section 7) that embody the architecture behind the state analysis process. In theory, the tool could directly translate these specifications in the database into a control system implementation using currently available code generation technology.

The State Analysis process is an incremental spiral methodology, so we have built versioning into the database schema. We can clone an "increment" to be the starting point for the next. In this way, we manage configuration at the increment level and promote an incremental methodology of discovery, without putting a previous increment's analysis at risk.

The database content can be simultaneously organized by system structure, work breakdown, and other criteria. The general organizing mechanism within the database is called an *Item*. An item may contain any number of state variables or other items. A state variable may belong to

more than one item. In this way, items can organize states into groups required for the analysis of overlapping domains. For example an inertial measurement unit (IMU) operating state may be included in:

1. the guidance, navigation, and control item, because the operating state must be controlled to insure that the IMU produces rate measurements needed for estimating the position and orientation of the spacecraft;
2. the power item, because the operating state affects power usage;
3. the temperature domain, because the operating state affect the IMU's temperature; and
4. the avionics domain because the IMU hardware is the responsibility of the Avionics work element.

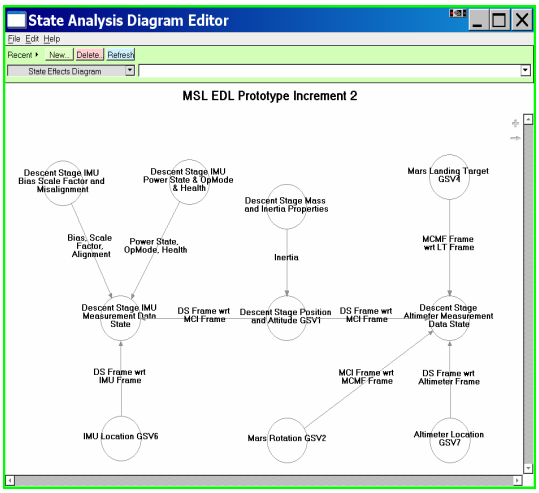

Figure 8: State Effects Diagram Editor.

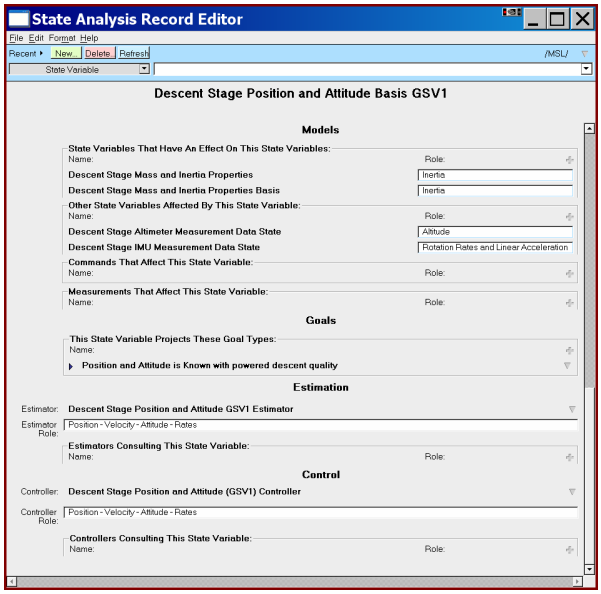

Figure 9: State Analysis Database record editor.

A standard spacecraft subsystem hierarchy can also be represented in terms of items. Items may also be used to group together states that correspond to a specific function or hardware element or groups of states in the environment with common effects on the spacecraft, etc…

Client User Interface. The graphical client tool includes a state effects diagram editor, a text-based record editor, and a report generator. This multiple-platform front-end client provides engineers with access to the database through HTTP requests. The client has been designed to work on a variety of platforms under Windows, MacOS 9, MacOS X, and most common Linux and Unix operating systems. This internet-based multi-platform approach enables large collaborative system engineering efforts for teams that may be geographically dispersed.

The state effects diagram editor (Figure 8) provides an environment for engineers that encourages them to take a state discovery approach to develop state-based models of the system to be controlled. New states can be easily added (and deleted), as well as effects between states. Each of the states on the diagram is hyperlinked to more detailed information accessed through a form in the record editor. This form contains fields for detailed descriptions of the state effects model.

The record editor (Figure 9) contains a record for each architectural element of State Analysis. The relationships between elements, such as an estimator that updates a

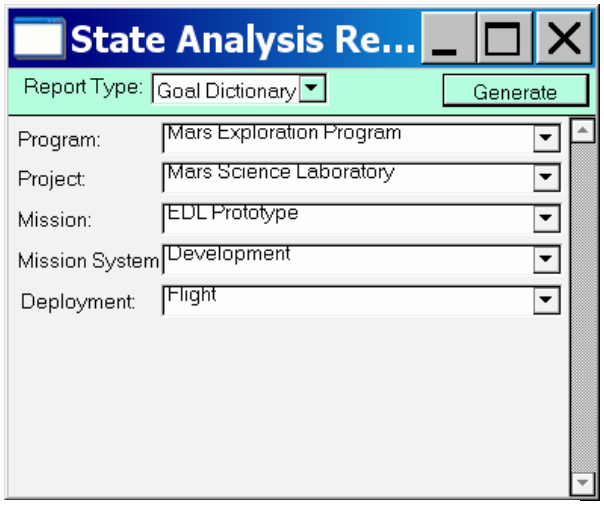

Figure 10: State Analysis Database report generator.

state variable, are laid out in the form. A related element is hyperlinked, so that clicking on its name opens its form. In this way, an engineer can easily follow a train of thought in the analysis by clicking though links. In addition, text fields may contain hyperlinks to files containing supporting analysis located in the web. The record editor can upload files to the file server and automatically insert hyperlink references to them in text fields. A user can apply HTML formatting to text descriptions and can embed diagrams and pictures in a variety of image formats.

The report generator (Figure 10) in the client is designed to be capable of generating a variety of reports from the information contained in the database, including the set of documents described earlier that typically appear in a project's document tree.

Configuration Management. The State Analysis Database has features for performing configuration management of state analysis artifacts. As described previously, the state analysis artifacts within an iteration in the spiral development process are organized into an increment, one increment for each spiral. When a user logs into the database client, the user picks an increment to view or modify. For the rest of that session only those artifacts within that increment are visible and new artifacts the engineer creates are automatically added to that increment. In addition, each user has an *identity* with an associated user name and password. Each identity is given a set of change authority paths that give the user permission to change or delete sets of records. *Change authorities* are organized into hierarchies of groups, so that collaborating users may be given change authorities for groups of related state analysis artifacts. These hierarchies can be organized according to a project's work breakdown structure to reflect team responsibilities for doing state analysis on their domains of expertise. Change authorities allow a project to manage the collaborative effort by assigning change authorities to engineers participating in their areas of responsibility, and by preventing them from changing artifacts in other areas. When a user logs into the client, the user selects a change authority path that has been granted to him. This gives him change access to the set of artifacts that are assigned to that path.

A web-based database administration tool has been developed that allows management of the identities, change authorities, and increments. The administration tool has the following functions:

1. Add User – creates a new identity with the following information: Name, Login Name, Title, Picture URL.
2. Update User – changes an identity's information.
3. Remove User – deletes an identity.
4. List Users – lists all the identities alphabetically by their Name, showing their Login Names,

and granted change authority paths.

5. User Privileges – lists an identity's granted change authority paths, and provides the ability to grant and revoke change authority paths.
6. Add Change Authority – adds a new change authority.
7. Delete Change Authority – deletes an existing change authority.
8. Create Authority Paths – places a change authority within the change authority hierarchy.
9. Delete Change Authority Path – removes a change authority from a location in the change authority hierarchy.
10. Change Authority Path Privileges – provides the ability to grant or revoke a change authority path for a selected identity.
11. Add Increment – creates a new increment by cloning an existing increment or by creating an empty increment.
12. List Increments – lists the set of increments.

Consistency and Data Integrity. The structure of database schema enforces architectural rules. For example, an estimator is not allowed to issue commands (only a controller may). This rule is enforced the linking table between an estimator and a hardware adapter that only allows an estimator to receive measurements from a hardware adapter, and not issue commands. In addition, data integrity is checked by the database application. For example, deletion of artifacts referenced elsewhere in the database can be flagged to the user. Scripts will also check architectural rules on relationships between the state effects model and the control system software specifications. Engineers may run consistency-checking scripts once they have completed the software specifications. This gives the engineer the flexibility of trying different control strategies without being restricted by the client. Scripts may be built into the tool or by running them on the database server backend.

"Shall" Statements and Future Work. In the future, we would like to link elements in State Analysis to traditional high-level requirements expressed as "shall" statements. The database organizes the state analysis artifacts into high-level elements called deployments, mission systems (for identifying the common types used by a set of deployments), missions (sets of mission systems, each corresponding to different groups of control systems for planning, development, and operations), projects (sets of missions), and programs (sets of projects). Requirements on these organizing elements may be specified as shall statements. Similarly shall statements on items may be used as requirements on subsystems. At lower levels of the analysis, we treat models as requirements. For example, models of hardware behavior and interfaces are captured in the state effects model through the identification of states, measurements, and commands and the effects between them. Such models are considered requirements. For high-level elements and items, we may add linkages to records containing shall statements or may link to requirement management tools though the use of IDs or other mechanisms for identifying external requirements. Sub-allocation of shall statements may be handled by linkages in external requirement management tools, or by providing allocation links between shall statements captured in the database.

Additional opportunities exist for adding capabilities to the State Analysis Database. These include code generation; elaboration and collaboration diagrams; other kinds of documents such as state, command, and measurement dictionaries; real-time execution requirements on collaborating elements such as estimators, controllers, hardware adapters, and state variables; simulation element requirements; and goal-based test cases and scenario definitions.

Today, the State Database provides systems engineers with a tool that can consolidate their

system and software requirements in a single place, and allows them to inspect, review, and validate 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 experimental rovers at JPL. In addition, an MDS adaptation has been developed for an Entry, Descent and Landing (EDL) stage based upon the preliminary design of a spacecraft scheduled for launch to Mars in 2009. This flight software prototype runs in a workstation environment, against a simulation of the EDL scenario. Currently, we are doing State Analysis for the Deep Space Network Array project, which will control an array of antennas to meet high communication bandwidth needs of future missions. The State Analysis for these rover, EDL, and DSN Array adaptations has been developed using the State Analysis Database. It should also be noted that a simpler Java implementation of the MDS architecture, called GoldenGate,[10] has 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. The State Analysis Database tool implements this language using a relational database and a set of user interfaces for developing, managing, inspecting, and validating the requirements. Together the methodology and tool 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 and the State Analysis Database is a systems engineering methodology and tool for today's complex systems that can carry us well into the future.

ACKNOWLEDGMENTS

This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government or the Jet Propulsion Laboratory, California Institute of Technology. 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 the contributions of Kenneth Rabe, Paul Ramirez, Sandy Krasner, Michelle McCullar, Alex Moncada, Erann Gat, Yu-Wen Tung, and Tom Starbird towards the development of the State Analysis Database.

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. 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.
11. Michel D. Ingham, Robert D. Rasmussen, Matthew B. Bennett, and Alex C. Moncada, "Engineering Complex Embedded Systems with State Analysis and the Mission Data System", *Proceedings of the 1st IEEE Intelligent Systems Technical Conference*, 2004.
12. M. Bennett and R. Rasmussen, "Modeling Relationships Using Graph State Variables", *Proceedings of the IEEE Aerospace Conference*, 2002.

BIOGRAPHIES

Matthew B. Bennett is a Senior Software Systems Engineer for JPL's Flight Software & Data Systems Section. He has contributed to the development of the State Analysis methodology, State Analysis Database, and Mission Data System software architecture. He has experience in spacecraft development, test, and operations, including the Galileo Jupiter and Cassini Saturn missions. He has developed software for fault protection, guidance and control, science data collection, performance analysis, and simulation. He received a M.Sc. from the University of Washington in Computer Science (1987), and a B.Sc. from the University of California at San Diego in Computer Engineering (1984).

Robert D. Rasmussen is Chief Engineer for the Systems and Software Division of JPL. He received his Ph.D in Electrical Engineering from Iowa State University in 1975. State analysis reflects lessons learned since then from his experience in systems engineering over several flight projects, starting with Voyager and continuing through Cassini, where he was development cognizant engineer for the attitude control subsystem. Project contributions have involved spacecraft guidance and control, avionics, test and flight operations, and autonomy - particularly for fault protection. Until recently, he was chief architect of the Mission Data System project under which state analysis was developed.

Michel D. Ingham is a Software Systems Engineer and member of the Senior Technical Staff at the Caltech Jet Propulsion Laboratory. He received his Sc.D. (2003) and S.M. (1998) degrees from MIT's Department of Aeronautics and Astronautics, and his B.Eng. (1995) in Honors Mechanical Engineering from McGill University. At JPL, Dr. Ingham is a member of the software architecture team for the Mission Data System, where his roles include the development of the State Analysis model-based systems engineering methodology, and the infusion of model-based programming and execution technology into NASA space exploration missions.