State Knowledge Representation in the Mission Data System[1,2]

Daniel Dvorak, Robert Rasmussen, Thomas Starbird
Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109-8099
818-393-1986
{daniel.dvorak, robert.rasmussen, thomas.starbird}@jpl.nasa.gov

Abstract—The possible states of a system, be it a spacecraft, rover or ground station, are what system engineers identify and specify, what software engineers design for, and what operators monitor and control. Many activities inside mission software are directly concerned with state, whether planning it, estimating it, controlling it, reporting it or simulating it. The cause of several mission failures can be traced to inadequate or inconsistent representations of state. Consequently, the concept of 'state' and its representation occupy a prominent role in mission software architecture. The Mission Data System (MDS), presently under development by NASA to provide multi-mission flight and ground software for the next generation of deep space systems, addresses this fundamental need. This paper describes the MDS approach to state knowledge representation, covering state variables, state functions, state estimates and state constraints, emphasizing design patterns that reduce sources of human error.

TABLE OF CONTENTS

1. INTRODUCTION

Mistakes and design errors are a natural part of engineering efforts, including software engineering for space missions. Fortunately, most serious errors are caught before deployment through a series of quality assurance gates that include reviews and testing. Some bugs still slip through but their effects are usually minor and manageable. Once in a while, though, an error goes through undetected until it causes a mission-ending failure. In such a case, a team of senior engineers and managers conduct a *post mortem* analysis to identify the most probable root cause(s) and examine how such an error slipped through various quality assurance gates.

Rather than placing blame on back-end practices that allowed an error to go undetected, it's just as important to look at front-end practices that allow such errors to be born. In software architecture and design there are many different crosscutting concerns that must be properly addressed to achieve a robust system. These concerns include design patterns, error-checking strategies, synchronization policies, resource sharing, distribution, performance and others [1]. This paper focuses on one very fundamental crosscutting concern—representation of state knowledge—because errors in this area have been implicated in more than one *post mortem* analysis [2].

Briefly, state knowledge concerns the representation of quantities such as camera temperature, switch position, gimbal angle, sensor health, and vehicle position. In designing software for holding such information, a natural tendency is to declare simple variables that use a programming language's built-in data types. As this paper will describe, this and other seemingly innocent decisions introduce dangers that can occasionally lead to a mission-ending failure. Consequently, the Mission Data System (MDS) project elevates state knowledge representation to an architectural concern whose design is shaped by two main objectives: (1) a more faithful reflection in software of 'state' in the physical world, and (2) a reduction in sources of human error in dealing with state information.

2. THE MISSION DATA SYSTEM PROJECT

In order to use software-engineering resources more effectively and to sustain a quickened pace of missions, while supporting the steady advances required by new

[1] 0-7803-7231-X/01/$10.00/© 2002 IEEE
[2] IEEEAC paper #040, Updated Dec 13, 2001

missions, JPL initiated a project in April 1998 to define and develop an advanced multi-mission architecture for an end-to-end information system for deep-space missions. The system, named "Mission Data System" (MDS), addresses several institutional objectives: earlier collaboration of mission, system and software design; simpler, lower cost design, test, and operation; customer-controlled complexity; and evolution to *in situ* exploration and other autonomous applications. JPL's Inter-Planetary Network and Information Systems Directorate manages the MDS project.

3. The MDS Vision

Software development for space missions is obviously part of a much larger endeavor, but software plays a central and increasingly important system role that must be reconciled with the overall systems engineering approach adopted by a project.

Software and systems engineering are highly interdependent for two reasons. First, software needs systems engineering products. It must know how things work. It needs to understand interfaces. And it has to honor the system engineer's intentions. Second, software is essential to systems engineering. It largely determines the behavior and performance of a system. It manages the capabilities and resources of a system. And it presents one's operational view of a system.

To put it in another way, both systems engineering and software deal in the more abstract aspects of a system. These are issues that apply from the earliest conception of a mission until the last day of flight operation. They apply across all constituents of a project and to all elements of the environment affecting the system. Therefore, it is essential that systems and software share a common approach to defining, describing, developing, understanding, testing, operating, and visualizing what systems do. This is the fundamental vision and philosophy behind the MDS design: that software is part of and contributes substantially to a new systems engineering approach that seamlessly spans the entire project breadth and life cycle.

It is in this context that state knowledge representation is treated as both a systems engineering concern and a software engineering concern.

4. Motivations

To motivate the MDS design for state knowledge representation, it's instructive to look at a couple code examples that illustrate problems in conventional software. In these examples the point is not that conventional code can't be made to work, for clearly it can, as evidenced by many successful missions. Rather, the point is that the code is vulnerable to certain kinds of errors that can be greatly reduced by providing more structure, encouraging a more disciplined approach.

The first example, shown in Figure 1, contains code to take some action when a pressure becomes too high. This code illustrates a common problem: lack of explicit representation of the state being estimated and controlled. A pressure is being monitored and controlled, but nowhere in

Conventional

```
get pressure sensor data
if data is not credible
then
   if persistence count is too high
   then
      set sensor failure indication
   else
      increment persistence count
else
   if data is below a threshold
   then
      take action
   reset persistence count
```

State Based

```
Get and publish pressure sensor data
```

```
// State Determination
if subscribed sensor data is credible
then
   decrease suspicion of sensor health
else
   increase suspicion of sensor health

if sensor health is good enough
then
   convert subscribed sensor data
   set pressure to converted data
else if sensor health is bad enough
then
   set pressure to unknown
```

```
//  State Control
if pressure is known and too high
then
   take action
```

Figure 1. This figure compares two approaches to implementing a control system that performs an action when a pressure is too high. The conventional approach, though shorter, lacks an explicit representation of the states being estimated (sensor health and pressure) and the state being controlled (pressure). The state-based approach is easier to validate because it converts evidence to states and separates state determination from state control.

```
...
// camera temperature state var
double cam_temp;
...
// update temperature value
cam_temp = function1();
...
// use temperature value
function2(cam_temp);
...
```

Figure 2. The use of a built-in primitive data type is a common approach for state variables, but it lacks representation of units, uncertainty and age, and lacks mechanisms for extrapolation in time, telemetry, naming and check-pointing.

the code is there a state variable representing pressure in Pascals (or any other suitable unit). Instead, a sensor's raw measurement in 'data number' form is used, provided that the data number passes some credibility test, and action is taken if the number is *below* a threshold. Why *below*? Because this sensor produces lower numbers for higher pressures. Other pitfalls exist: sensor data must be reinterpreted on the ground; ground must reset the failure indicator; and the action threshold is likely to be wrong if the sensor is recalibrated. In contrast, a state-based approach explicitly represents pressure in a state variable and includes 'unknown' as a possible value. It also treats sensor health as a separate state, as indeed it is. Finally, it keeps state determination logic separated from control logic, making the code easier to understand and reuse.

The second example, shown in Figure 2, contains code to estimate and control camera temperature. The simple representation of temperature state in a floating-point variable is a common approach, with direct unregulated access by all clients. This approach holds several shortcomings and dangers: (1) units are not explicitly represented, so we can't tell if the designer intended Kelvin or Celsius or Fahrenheit; (2) such variables are often declared as global, later requiring addition of some form of thread-safe access; (3) there is no representation of uncertainty, so a controller may be unknowingly making control decisions using highly uncertain information; (4) there is no associated timestamp indicating how long ago the variable was updated; (5) there is no provision for extrapolation in time, except for the *de facto* notion of constant-until-updated, which is often wrong; (6) such variables' values are often meaningful only under certain circumstances, which users must deduce from context; (7) predictions of future values and archives of past values must be handled elsewhere; (8) there is no general discipline to avoid "delta" representations (e.g., delta-V), which must be used with caution and which often create issues with when and how to reset their values; (9) telemetry and its controls must be provided elsewhere; (10) check-pointing and reboot initialization from persistent storage must be provided; and (11) a naming mechanism must be provided for supporting operational queries.

All of these shortcomings and dangers beg for uniform mechanisms for all state variables. Without an architecture for state knowledge representation, two undesirable things can happen. First, and most importantly, inadequate representations will appear and will make control logic and estimation logic error prone and hard to understand. Second, different subsystem teams will create their own solutions for the needed capabilities, leading to the all-too-common difficulties of subsystem integration.

5. State Architecture

MDS is founded upon a state-based architecture, where *state* is a representation of the momentary condition of an evolving system and *models* describe how state evolves. Together, state and models supply what is needed to operate a system, predict future state, control toward a desired state, and assess performance (see Figure 3).

System states include device operating modes, device health, resource levels, attitude and trajectory, temperatures, pressures, etc, as well as environmental states such as the motions of celestial bodies and solar flux. Some aspects of system state are best described as functions of other states; e.g., pointing can be derived from attitude and trajectory.

The totality of state representations, largely organized hierarchically within control systems, should provide a

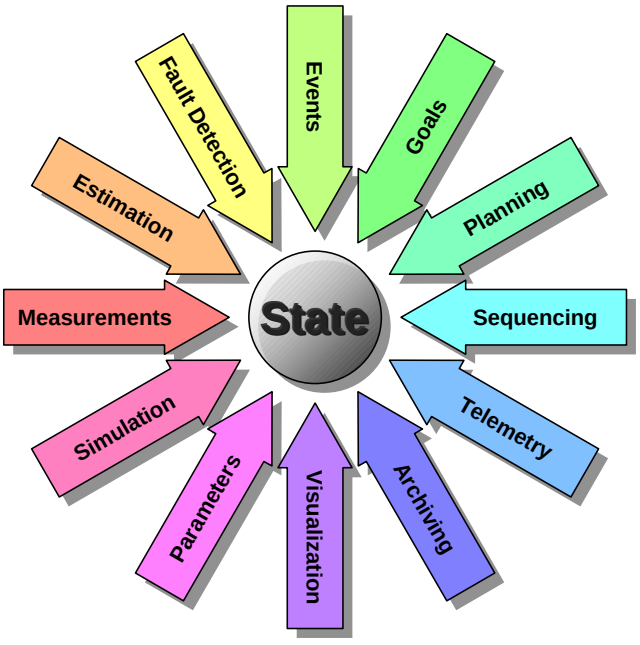

Figure 3. *State* is a representation of the momentary condition of an evolving system. System state is the architectural centerpiece for information processing because many activities are involved with state.

complete representation of the total system ("complete" in the sense of providing adequate knowledge of state for all control purposes). While there may be elements of a project outside the MDS purview, the external elements are described at least by their visible behavior. In all cases, state is accessible in a uniform way through *state variables*, as opposed to a program's global and local variables.

State evolution is described on *state timelines*, which are a complete record of a system's history ("complete" to the extent that the state representations are adequate, and subject to storage limitations). State timelines capture current and past estimates, future predictions and plans, and past experience. State timelines provide the fundamental coordinating mechanism since they describe both knowledge and intent. This information, together with models of state behavior, provides everything needed to predict and plan, and it is available in an internally consistent form, via state variables.

State timelines are also the objects of a uniform mechanism of information exchange between Flight and Ground, largely supplanting conventional engineering data traffic in both directions. 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 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.

6. State Knowledge Overview

All of these needs have shaped the architecture of state knowledge representation in MDS. The next three sections will describe that architecture in terms of three simple concepts: state values, state functions, and state variables. The concepts are illustrated with concrete examples of classes described in the Unified Modeling Language (UML) [3].

7. State Value

Timestamp

A "state value" is the value of a state variable at an instant in time. A state value contains a timestamp and a value. The timestamp is represented in a uniform way using a framework class RTEpoch (run-time epoch). Timestamps (objects of type RTEpoch) have value semantics in that they can appear in equations of time and can be compared to other timestamps. "Run-time" refers to the fact that a timestamp's time frame is part of the object and is used in time calculations and comparisons. Time frames include International Atomic Time (TAI), Ephemeris Time (ET), Coordinated Universal Time (UTC) and others.

Value and Uncertainty

In deployments other than simulation, where state knowledge is always uncertain, a state value is termed an *estimate*. The "value" part of an estimate has no standard form since there are many ways to represent state knowledge, including its uncertainty. In an example of camera temperature estimates (see Figure 4) detailed temperature state is represented as a normal distribution in degrees Kelvin (the base unit for temperature in SI), and a compressed temperature state is represented as a uniform distribution in degrees Kelvin. These are just two of many possible representations. Another possible representation is the qualitative values 'cold', 'nominal' and 'hot' paired with qualitative certainties such as 'possibly', 'probably' and 'certainly'. The choice here, as in other variation points, should be driven by need; choose a representation that is adequate for the task without introducing unnecessary complexity.

In designing a value representation it's helpful to think of values as objects that can answer questions about themselves, particularly questions that a controller might ask, rather than as publicly visible data attributes. For example, regardless of the particular representation chosen for camera temperature, a camera temperature controller will need to make decisions about turning a heater on or off, so it will need to ask questions of the form "is the

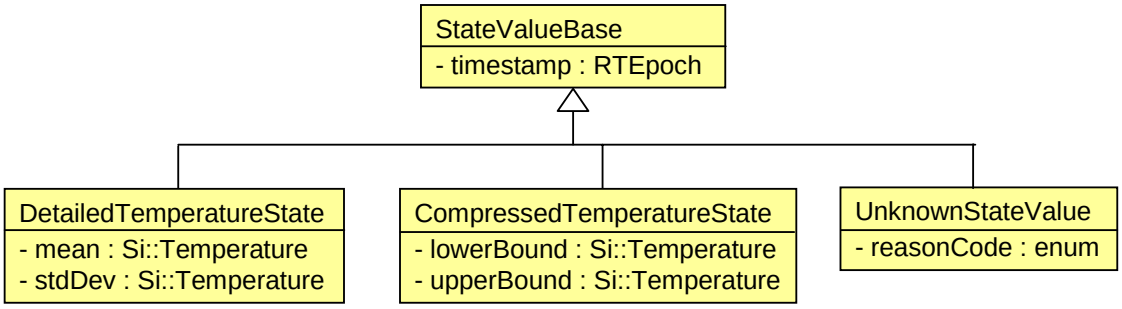

Figure 4. A state value is a time-stamped object that represents the value of a state variable at an instant of time. State values in deployments *other* than simulation deployments are called "estimates" because they represent uncertainty in some form. The three leaf classes above describe three kinds of state values for a hypothetical temperature state variable, reflecting different levels of detail that may exist in different parts of its state timeline.

temperature within range *r* with certainty ≥ *c*?" and "is the temperature below value *v* with certainty ≥ *c*?" Neither question dictates a particular internal representation.

Unknown Estimate

The most extreme form of uncertainty in a state estimate is "unknown". This value can arise in an estimate for several reasons: complete lack of evidence (such as due to sensor failure), deeply conflicting evidence, query at a time in distant past or future, query during system startup, and query for a time in the past after that portion of history has been deleted. As such, MDS requires that the state value design for every state variable include a way to represent "unknown".

One possible way to represent 'unknown' is to include a flag in an otherwise ordinary estimate class. However, this approach is vulnerable to a simple programming error where some code fails to check the flag, and therefore uses the estimate as a known quantity. Another way to represent 'unknown' is through a reserved value that should never occur in normal operation, such as a variance of infinity. Again, this approach is vulnerable to programming error where the code fails to check for the reserved value and then blindly uses the estimate in the ordinary way. To avoid these errors MDS requires that 'unknown' be represented using a distinct data type, ensuring that client code cannot accidentally treat an unknown estimate as a known estimate.

Unit Safety

Many states in the physical world are described as scalars or composites of scalars. A scalar is a quantity such as mass, length, time or temperature, completely specified by a number on an appropriate scale. Unfortunately, mainstream programming languages offer no built-in support for scalars, so programmers typically use "naked" numeric types such as 'float' and 'double'. The problem with this approach is that there is no protection against three kinds of errors: *interface errors* (e.g. force passed to an interface where mass was expected, or voltage and current arguments transposed in an interface), *scale errors* (e.g. length given in feet where meters was expected, or length given in kilometers where meters was expected), and *expression errors* (e.g. a formula that adds velocity and acceleration, or a conditional that compares power level to energy level).

Through suitable class design it is possible to protect against such errors with detection at compile time (preferred) or run time. By taking advantage of templates as specified in the international standard for C++ [5], it is possible to support scalars such that all errors are detected at compile time. Such a package for supporting the SI system of units (*Le Système International d'Unités*) is available from the Fermilab Physics Class Library Task Force [6]. Interestingly, good optimizing compilers can eliminate all runtime overhead associated with this design such that unit-safe expressions can be evaluated with the same performance as unsafe expressions that use built-in numeric data types.

Ideally, unit safety should be practiced in all software, but in reality there is a lot of legacy software (including math packages) that is widely used and well tested, though unsafe in the sense of using 'naked' numeric data types. A pragmatic approach in such cases is to protect critical interfaces—particularly subsystem interfaces where different teams work on different sides of the interface—and still capitalize on the internal legacy code.

Similar arguments can be made for coordinate frame safety; units are just one-dimensional coordinate frames and physical type rolled into one concept. Frame-tagged vectors (and other quantities) are in design.

Simulation States

State values that appear inside simulators differ from state estimates that appear inside flight/ground deployments in one aspect: they contain no uncertainty. Physical quantities in a simulated world are true, just as they are in the physical world. Aside from this difference, the same state-based architecture applies equally to flight, ground, and simulation deployments in MDS. This commonality contributes to system verification in a significant way because it facilitates direct comparison of simulated state and estimated state.

8. TIMELINE & STATE FUNCTION

State knowledge is represented on timelines that span past, present and future (Figure 5). A timeline expresses two aspects of state knowledge: knowledge of what has been *estimated* from observations and models, and knowledge of what has been *planned* from operational goals and models.

Estimated States

As a matter of architectural principle, MDS specifies that estimated states be defined for all instants of time, from the beginning of a timeline (termed "distant past") to its end (termed "distant future"). Naturally, for some state variables, there will be regions on the timeline where estimated states are unknown, such as camera temperature a year before mission start. That's OK since 'unknown' is always a possible state estimate, as described earlier.

Estimated states on a timeline are represented through a series of "state functions" that, collectively, cover all time continuously from distant past to distant future. The term "state function" indicates that these are functions of time that can return a state estimate given a time. There are many ways of expressing a function of time, and MDS does not limit a user's choices. For example, constant functions, linear functions, and higher-order polynomial functions are all candidates. Likewise, discontinuous functions that describe abrupt state transitions (such as closing a switch) are candidates. In all cases the choice of function should be

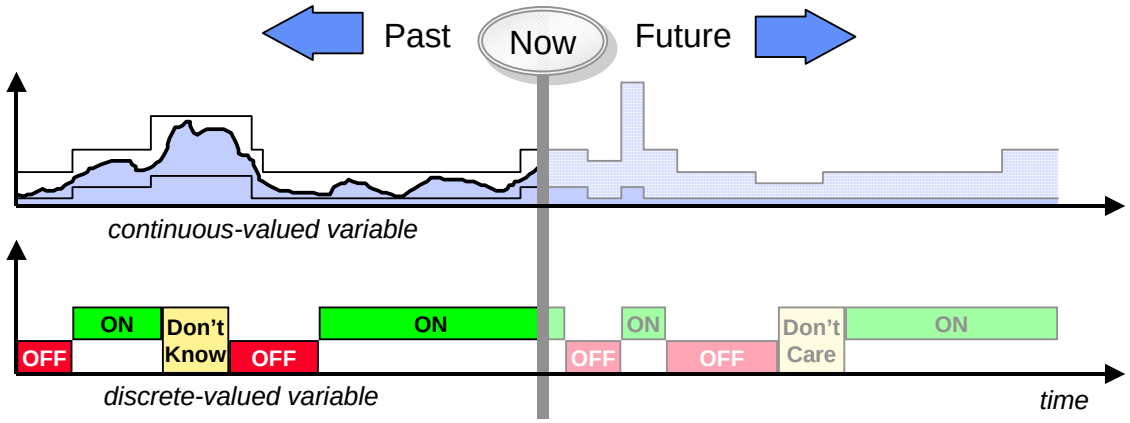

Figure 5. A timeline represents a state variable's value as a function of time spanning the past, present, and future. A timeline holds two kinds of state knowledge: estimated states based on interpretation of observations, and planned states based on operational goals. As shown, the future part of a timeline contains planned states while the past part contains both planned and estimated states. Estimated state is represented in a series of state functions and planned state is represented in a series of goals.

driven by need, as influenced by system dynamics and estimation rate.

The principle of a continuously defined timeline may seem unusual in the context of conventional practice where histories consist of time-stamped samples, but there are three important motivations for it. First, MDS strives to reflect the underlying physics. States in the physical world are defined at all instants, and the role of state knowledge representation in MDS is to represent that reality. Second, cyclic real-time applications become less sensitive to jitter and cycle-slip since they can obtain estimates for synchronous instants in time, as opposed to whenever the data happened to be sampled. For example, the Cassini attitude and articulation control system (AACS) uses interpolation functions for exactly this purpose [13]. Third, functions of time can be compressed in a variety of memory-saving ways while preserving much of the information. For example, a series of piecewise linear functions of time can be replaced by a curve fit to a polynomial function.

Time Derivatives

In physics the concept of "state" refers to physical quantities whose values, collectively, provide an instantaneous description of a system. Position and velocity are considered separate states, but not all time derivatives are states in the physics sense.

MDS, in contrast, keeps all time derivatives of x in the state variable for x. This difference is a natural consequence of the fact that state knowledge is represented in functions of time (state functions). A state variable for spacecraft position implicitly contains knowledge about spacecraft velocity since its state functions contain position versus time. If there were separate state variables for position and velocity then not only would there be redundant information but also their respective values would have to be kept consistent at all times. As a matter of simplicity and safety, then, MDS keeps all time derivatives of a quantity together in the same state variable.

Planned States

The MDS architecture is designed for goal-driven operation. By definition, an MDS goal is a constraint on the value of a state variable over a time interval. A constraint defines a set of state histories that satisfy a goal. Accordingly, a state timeline contains a series of goals that represent the current plan. The plan part of a timeline, then, differs from the estimated part in two ways. First, the value of a plan at an instant of time is a *set* of states rather than a single state, reflecting the fact that any state in that set is compatible with the goal. Second, a goal's time interval is bounded by two "time points" whereas, on the estimated part of the timeline, state functions are bounded by two absolute times. A time point is different in that it represents a time range resulting from temporal constraints among goals.

This paper focuses primarily on the estimated part of a timeline. For a description of the interfaces and operations on the plan part of a timeline, see [9].

9. State Variable

The role of state variables is to provide access to state knowledge. State variables are like "Grand Central Station" in the MDS architecture since every component that needs to obtain or update state information goes to the appropriate state variables. State variables serve not only real-time clients such as estimators and controllers but also deliberative clients for goal elaboration and scheduling.

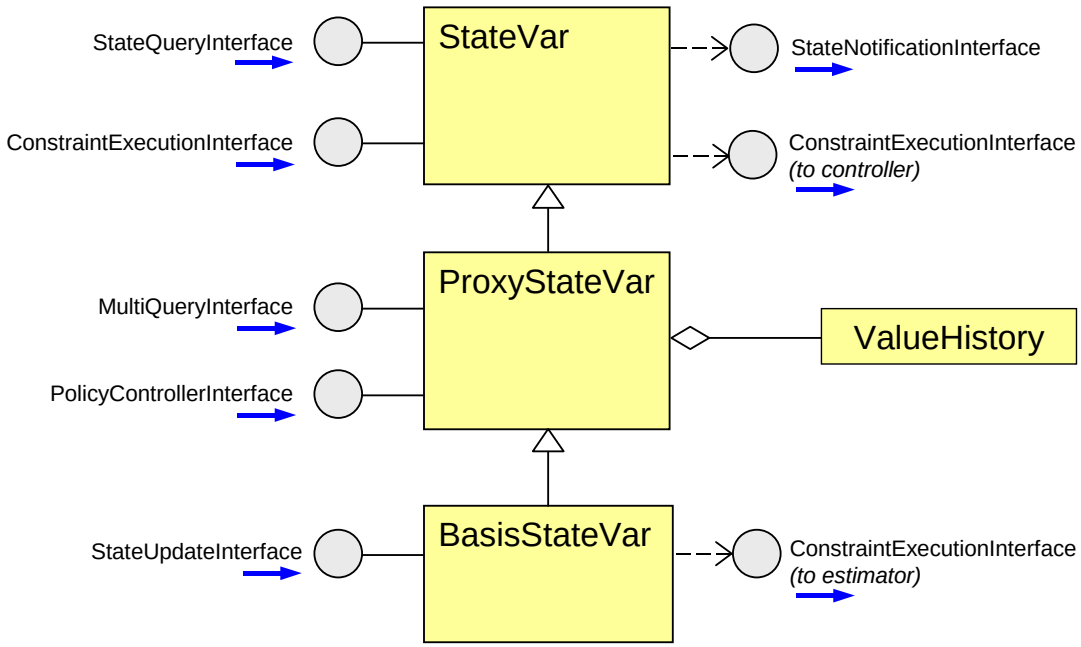

Figure 6. The state architecture defines what interfaces—and thus operations—are valid for a state variable, depending on its type. At the top of the hierarchy a "derived" state variable has no value history of its own but can derive a state value by combining values from other state variables. A "proxy" state variable has a value history and therefore supports primitive query operations and policy control of that history, but provides no update operation since a proxy provides read-only access. A "basis" state variable is locally estimated so it adds a state update interface.

Kinds of State Variables

There are three kinds of state variables, as shown in Figure 6. A "basis state variable" has a local estimator that updates its state timeline as needed. Such variables are typically located near sources of evidence and the ability to interpret that evidence. For example, a planetary lander having a battery temperature sensor would have a basis state variable for battery temperature. A "proxy state variable" provides remote, read-only, time-delayed access to the state timeline of a corresponding basis state variable. For example, since battery temperature would be of interest to earth-based engineers, there would be a proxy state variable for battery temperature in a ground deployment. The locations of basis state variable and its proxy can occur in the opposite order as well. For example, sensor calibration is often estimated on the ground by humans or ground-based software and then up-linked. In such a case the ground deployment would have the basis state variable and the flight deployment would have the proxy state variable.

The third kind of state variable is a "derived state variable", so named because any value that it returns is derived from two or more other state variables. For example, the total power consumed by three instruments could be made available in a derived state variable that has access to the three individual instrument power state variables.

In addition to ordinary state variables whose values are implicitly defined with respect to some standard reference, there is another category of state variable whose values always represent a relation between two entities. One example of the latter is spacecraft position relative to celestial bodies. In a space mission the most 'interesting' view of spacecraft position changes, depending on planned activities. In a mission such as Cassini, with multiple gravity assists, the view has changed over time among Earth and Venus and Jupiter and Saturn and the Sun. This category of state variables that represents relational states is termed "graph state variables" because state knowledge is represented in the edges of a graph that are traversed from one node to another when answering a query about the value of one node relative to another. For more details, see Bennett [4].

State Variable Interfaces

State variables in MDS reflect the union of its state-based architecture and its component-based architecture. The state-based architecture covers the concepts described earlier, namely, state estimates, timelines, and state functions. The component-based architecture covers software engineering issues such as interfaces, ports, components, connections, and synchronization [8]. This section focuses on interface descriptions for state variables.

As the name suggests, an interface represents an agreement between two parties regarding how they will interact with each other. In order for two components to interact, they must do so through a connection between a port on one

component and a port on the other component. Both ports must be defined in terms of the same interface. One component "provides" the interface while the other component "requires" it. Very simply, this is the distinction between the called party (the provider of a service) and the calling party (the user who requires the service). For example, a state variable provides a state query interface for use by other clients, such as controllers, that need to query a state variable.

A state variable has several interfaces, depending on the kind of state variable, as shown in Figure 5. Broadly, there are five kinds of interfaces for state update, state query, notification upon change, data management control, and constraint execution. The following subsections describe the purpose of each interface and the operations that it supports. Interface declarations are detailed in Figure 7.

State Query Interface—The purpose of the state query interface is to provide an operation for obtaining a state value at an instant in time. As such, the 'getState' operation takes a single time argument and returns a smart pointer to a state value object. The data type of that object can vary depending on whether the data has been compressed and whether the value is unknown. This interface currently defines only a single operation that returns a smart pointer to a state value object. Such an operation is general, safe, and efficient for large objects, but involves a fair amount of overhead for small objects. Other query operations involving return-by-value may be added to support different tradeoffs among speed, memory, and safety.

During state variable initialization the state query interface is locked in such a way that all queries return 'unknown' (by returning the UnknownStateValue object shown in Figure 4). This prevents controllers and other clients from taking inappropriate actions based on garbage values that could be present during startup.

State Update Interface—The purpose of the state update interface is to provide operations for timeline updates: routine updates as well as startup initialization. This interface exists for the exclusive use of a single state estimator / generator, an architectural rule that is enforced by the component manager. As noted earlier, a state variable begins life in a locked state, and any queries to the state query interface are rebuffed with a returned value of 'unknown'. The first duty of a state estimator / generator upon startup is to initialize the timeline and then unlock the state variable. Initialization can include selective recovery of state from persistent storage (if desired), repairing erroneous areas of history, and obtaining new evidence from sensors. The fact that a system reset has occurred is itself evidence that may influence the updated value of state.

State Notification Interface—The purpose of the state notification interface is to notify interested listeners when a state variable's timeline has been updated. This interface supports the Observer design pattern for data-driven reactions [7]. Note that this is an interface that a state variable *requires*, rather than one it provides, since this is an interface that it *calls*. The trigger for notification depends on the type of state variable. A basis state variable calls the 'changed' operation when its own 'updateState' operation has been called. However, a proxy state variable calls 'changed' upon receipt of new data from the data transport service. Notification always includes the identity of the notifying state variable plus a vector of changed items in the timeline.

Policy Control Interface—Internally, a state variable's estimated state timeline is managed by a data management service. This service is managed by policies that specify when to checkpoint, what to transport, when to compress, how much to recover upon startup, and other management functions. The policy control interface exists for the purpose

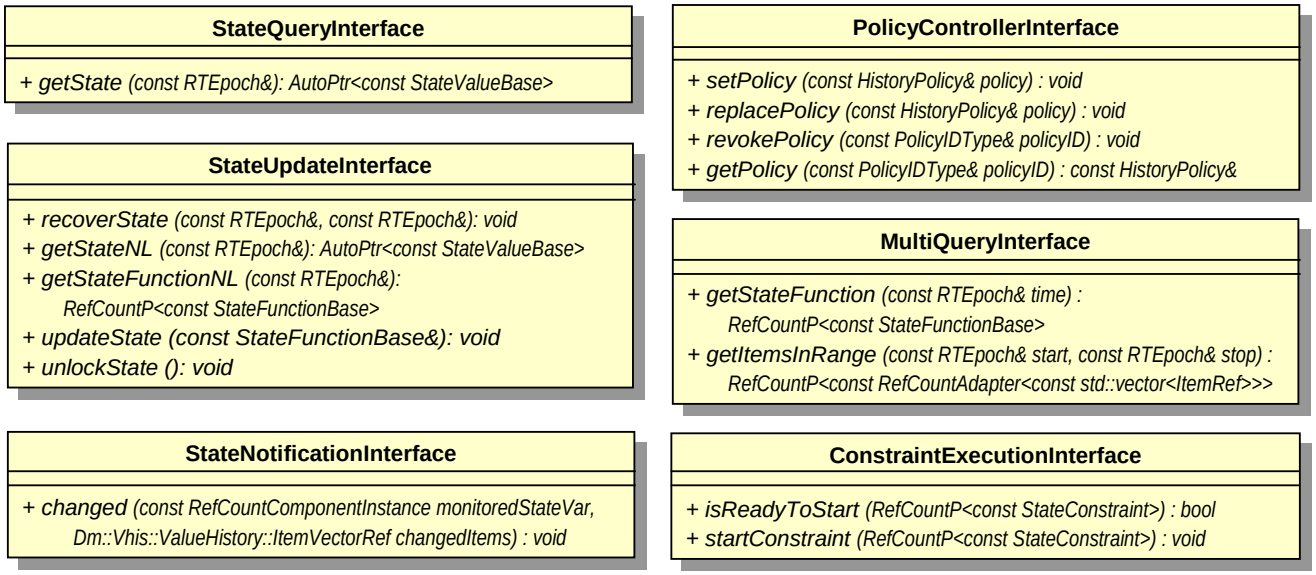

Figure 7. State variable interface declarations.

of adjusting these data management policies.

Multi Query Interface—Ordinarily a state variable exposes only its state values to clients, not its state functions. This deliberately hides the form of its state functions as an implementation detail that can be changed without affecting clients. However, there are some situations where a client needs access to state functions in order to compute a value that depends on the shape of one or more state functions (such as to compute the area under a curve). To minimize unnecessary dependencies upon implementation details, use of this interface is restricted to special cases and is treated as an automatic inspection item.

Constraint Execution Interface—In addition to its role as *the* access point for state knowledge, state variables also act as intermediary for clients that need to talk to each other concerning their mutual interest in a state variable (see Figure 8). In this respect the component for dispatching executable goals for state variable *x* needs to talk to the goal achievers for *x* (its controller and/or estimator). The constraint execution interface makes this possible. The interface is both provided and required by the state variable; it is provided so that the goal dispatcher can call it, and it is required so that the state variable can relay the calls to associated goal achievers. Note that using the state variable as an intermediary has the desirable property of decoupling the estimator, controller, and goal dispatcher that would otherwise have to know of each other's existence.

10. Related Work

Software architectures vary significantly in the stature given to state knowledge versus other architectural elements. Some architectures that elevate state knowledge to an important role, though not precisely in the same way as MDS, include PRS-CL[3] [10], Altairis MCS [11], and the Deep Space One Remote Agent [12].

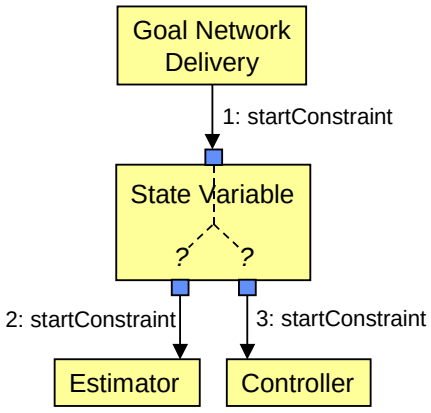

Figure 8. In addition to its role as access point for state knowledge, a state variable helps reduce coupling among other components by acting as an intermediary for communication among components that would otherwise need references to each other.

11. Summary

"State knowledge" is what you know and how well you know it. State knowledge encompasses the many states that a system must know about, such as vehicle position, device temperature, sensor calibration values, power usage, terrain topology, and many others. Inadequate or inconsistent state representations can and have caused mission-ending failures. Common representational deficiencies include indirect or hidden representations, multiple private and potentially inconsistent estimations of the same state, lack of units of measurement, lack of or hidden expressions of uncertainty, lack of timestamps, and "algorithm state" as a substitute for published readable state information.

As a state-based architecture, MDS elevates state knowledge representation to an architectural concern that begins with systems engineering in identifying important states and continues through to software engineering, verification & validation, and operation. The MDS state architecture and associated framework software design support state knowledge representation through four architectural elements: state variables, timelines and state functions, state estimates, and state constraints. This architecture has been shaped by two main objectives: a closer reflection in software of the physical states that are being monitored and controlled, and a desire to reduce sources of human error in using and updating state information in mission software.

References

[1] Xerox Corporation, *Aspect-Oriented Programming*, http://www.parc.xerox.com/csl/projects/aop/

[2] Thomas Young (chair), "Mars Program Independent Assessment Team Summary Report", March 14, 2000, http://www.nasa.gov/newsinfo/mpiat_summary.pdf.

[3] Grady Booch, James Rumbaugh, Ivar Jacobsen, "The Unified Modeling Language User Guide," 1999, Addison Wesley Longman, Inc.

[4] Matthew Bennett, "Modeling Relationships Using Graph State Variables", *2002 IEEE Aerospace Conference*.

[5] International Standard ISO/IEC 14882, Joint Technical Committee ISO/IEC JTC 1, *Information technology*, subcommittee SC 22, *Programming languages, their environments and system software interfaces*, Sep. 1, 1998.

[6] Walter E. Brown, "Introduction to the SI Library of Unit-Based Computation", *International Conference on*

[3] PRS-CL is a trademark of SRI International.

Computing in High Energy Physics (CHEP '98), August 31–September 4, 1998.

[7] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, 1995.

[8] Clemens Szuperski, "Component Software: Beyond Object-Oriented Programming", Addison Wesley, 1999.

[9] Russell Knight, Steve Chien, and Gregg Rabideau. "Extending the Representational Power of Model-based Systems using Generalized Timelines." *The 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space* (i-SAIRAS 2001), June 18-22, 2001, Montreal, Canada.

[10] Karen Myers, "PRS-CL Architecture," http://www.ai.sri.com/~prs/prs-arch.html.

[11] Aeroflex Altair Cybernetics Corporation, http://www.altaira.com/

[12] B. Pell, D. Bernard, S. Chien, E. Gat, N Muscettola, P. Nayak, M. Wagner, B. Williams, "An Autonomous Spacecraft Agent Prototype," *Proceedings of the First Annual Workshop on Intelligent Agents*, Marina Del Rey, CA, 1997.

[13] G. M. Brown, D. Bernard, R. Rasmussen, "Attitude and Articulation Control for the Cassini Spacecraft: A Fault Tolerant Overview," *Proceedings of the 14th AIAA/IEEE Digital Avionics System Conference*, November 1995

Daniel Dvorak *is a principal engineer in the Exploration Systems Autonomy section at the Jet Propulsion Laboratory, California Institute of Technology, where his interests have focused on state estimation, fault detection and diagnosis, and verification of autonomous systems. Prior to 1996 he worked at Bell Laboratories on the monitoring of telephone switching systems and on the design and development of R++, a rule-based extension to C++. Dan holds a BS in electrical engineering from Rose-Hulman Institute of Technology, an MS in computer engineering from Stanford University, and a Ph.D. in computer science from The University of Texas at Austin.*

Robert Rasmussen *is a principal engineer in the Information Technologies and Software Systems division at the Jet Propulsion Laboratory, California Institute of Technology, where he is the Mission Data System architect. He holds a BS, MS, and Ph.D. in Electrical Engineering from Iowa State University. He has extensive experience in spacecraft attitude control and computer systems, test and flight operations, and automation and autonomy — particularly in the area of spacecraft fault tolerance. Most recently, he was cognizant engineer for the Attitude and Articulation Control Subsystem on the Cassini mission to Saturn.*

Thomas Starbird *is a principal in system design in the Mission Systems Engineering section of the Jet Propulsion Laboratory (JPL), California Institute of Technology. He received a B.A. from Pomona College and a Ph.D. from the University of California at Berkeley, both in Mathematics. He has participated in software development and Mission Operations System development for several space projects at JPL. He led the implementation of SEQ_GEN, multi-mission software used for constructing and checking sequences of commands to be sent to a spacecraft. He is currently leading the implementation of the Planning & Execution portions of the Mission Data System.*