Appears in Proceedings of
2000 IEEE Aerospace
Conference, March 2000.

Software Architecture Themes in JPL's Mission Data System[1]

Daniel Dvorak, Robert Rasmussen, Glenn Reeves, Allan Sacks
Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109-8099
818-393-4109
{dldvorak,rrasmssn,greeves,asacks}@pop.jpl.nasa.gov

Abstract—The rising frequency of NASA mission launches has highlighted the need for improvements leading to faster delivery of mission software without sacrificing reliability. In April 1998 Jet Propulsion Laboratory (JPL) initiated the Mission Data System (MDS) project to rethink the mission software lifecycle—from early mission design to mission operation—and make changes to improve software architecture and software development processes. As a result, MDS has defined a unified flight, ground, and test data system architecture for space missions based on object-oriented design, component architecture, and domain-specific frameworks. This paper describes several architectural themes shaping the MDS design and how they help meet objectives for faster, better, cheaper mission software.

TABLE OF CONTENTS

1. INTRODUCTION

JPL's deep space missions tend to be one-of-a-kind, each with distinct science objectives, instruments, and mission plans. Until recently, these missions were spaced years apart, with little attention to software reuse, given the relatively rapid pace of computer technology and computer science. Also, since radiation-hardened flight computers remain years behind their commercial counterparts in speed and memory, flight software has typically been highly customized and tuned for each mission. Thus, when JPL launched six missions in six months between October 1998 and March 1999, it wasn't surprising that there was little software reuse among them, except in the ground system.

However, despite the uniqueness of each mission, they each had to independently design and develop mechanisms for communication, commanding, attitude control, navigation, power management, fault protection, and many other standard tasks, yet there was no common architecture or frameworks for them to build upon. Clearly, in an era of monthly missions, this is an inefficient way to use software-engineering resources.

Another change affecting our approach to deep space mission software is the advent of high performance, commercially standard flight computing systems suitable for flight use. Sufficient capability now exists to justify investing a substantial part of the system resources to reusable designs and "off-the-shelf" components, which are typically not as efficient as customized code. This additional capability is also a timely boost to increased autonomy that new missions require as we move into an era of in situ exploration.

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 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 evolvability to in situ exploration and other autonomous applications. JPL's Telecommunication and Mission Operations Directorate (TMOD) 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.

[1] 0-7803-5846-5/00/$10.00 © 2000 IEEE

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.

This paper describes several architectural themes shaping the MDS design. These themes have been highlighted because they have broad impact on the design and because they differ from earlier practices. However, the themes are not novel ideas; they draw proven ideas from control systems, robotics, data networking, software engineering, and artificial intelligence.

Although most of these themes have resulted from a desire to improve flight software—and have compelling examples there—they apply equally to ground software. Also, these themes apply equally to all kinds of robots, whether spacecraft or probes or rovers.

4. Architectural Themes

Theme: Take an Architectural Approach

Construct subsystems from architectural elements, not the other way around—It has been traditional in JPL missions to divide the work along at least five dimensions: flight—ground—test, design—test—operations, engineering—science, downlink—uplink, and subsystems (navigation—power—propulsion—telecom, etc). With the work so compartmentalized, software engineers naturally applied their own customized solutions within each realm, resulting in minimal reuse and requiring many iterations at integrating the subsystems. The net result was always architecture constructed from subsystems.

In MDS there is a quest to identify common problems and create common solutions that can then be tailored to particular problems. We refer to this collection of common solutions as the MDS framework. It provides shared core elements among different systems, eliminates redundant or conflicting developments within systems, and assures uniformity across the architecture in order to improve operability and robustness. In the spirit of DARPA's Domain-Specific Software Architecture (DSSA) Project [1], the MDS project is designing a *reference architecture*, i.e., a software architecture for a family of applications in a domain.

Object oriented analysis and design contribute to the architecture to some extent, but a fundamental driver in this approach has been the recognition that space system designs are always tightly coupled, despite best attempts. Highly constrained resources demand it. A key software role is to make this coupling manageable. Therefore, managing interactions is also a foundation of good design. For example, different activities in different subsystems issue commands that consume power, and they can potentially interfere with each other unless there is a coordination service that keeps track of available power and that has authority to control each device. Creating such a coordination service enables a cleaner simpler design

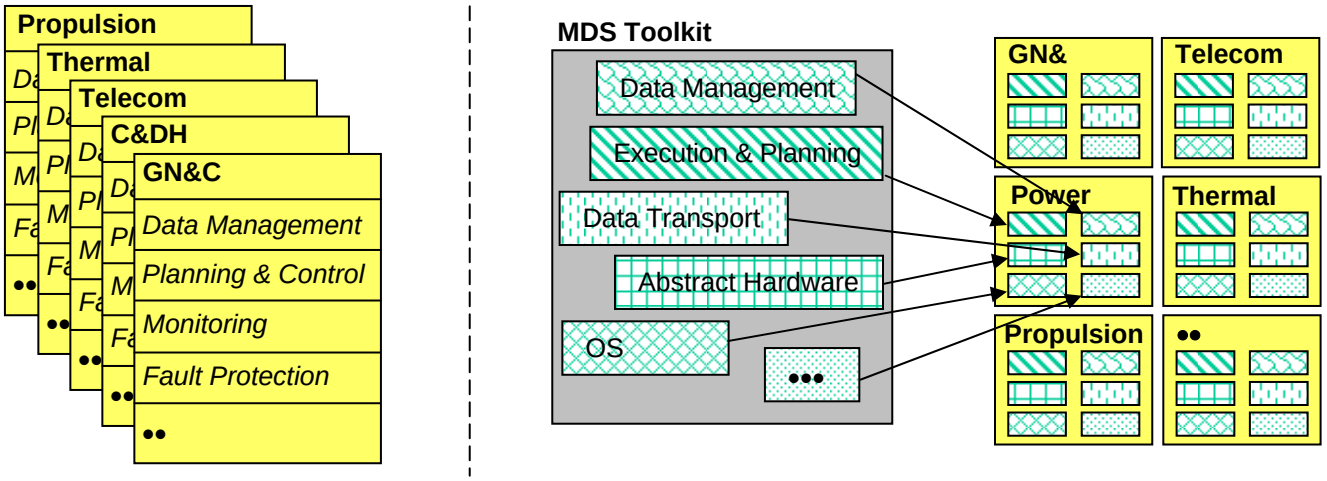

Figure 1. In the traditional approach, subsystem teams worked in isolation and created individual solutions to shared problems. In the architectural approach they apply standard MDS frameworks and services.

because it controls interactions through a common service rather than through private subsystem-to-subsystem agreements, thereby decreasing the apparent coupling between subsystems. It similarly simplifies unit testing of subsystems. The net result from applying this approach is that subsystems get constructed from architectural elements, not the other way around.

Theme: Ground-to-Flight Migration

Migrate capability from ground to flight, when appropriate, to simplify operations—MDS takes a unified view of flight and ground tasks because of opportunity and need. With increasingly powerful flight processors the *opportunity* exists to migrate to the spacecraft (or rover) some functions that have traditionally been performed on the ground, thereby reducing the need for flight-ground communication. Such migration might occur well after launch, after ground operators have gained experience with the real spacecraft and have decided that some activities can be automated, without further human-in-the-loop control. Migration can involve using the same code in flight as on the ground, but frequently flight implementations are different because they exploit the immediacy of their interaction with the spacecraft. Nevertheless, uniformity in addressing other system elements permits these migrations to take place with minimal perturbation to the rest of the system.

More importantly, the *need* for such migration exists in order to accomplish missions that must react quickly to events, without earth-in-the-loop light-time delays, such as autonomous landing on a comet, and rover explorations on Mars. By adopting a unified architecture, we assure that the wide range of possibilities offered by these missions can be accommodated with a single MDS framework. For these reasons both flight and ground capabilities must be designed for a shared architecture.

Theme: State & Models are Central

System state and models form the foundation for information processing—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.

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 complete representation of the total system ("complete" in the sense of providing adequate knowledge of state for all

Figure 2. System state is the architectural centerpiece for information processing. *State* is a representation of the momentary condition of an evolving system.

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 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.

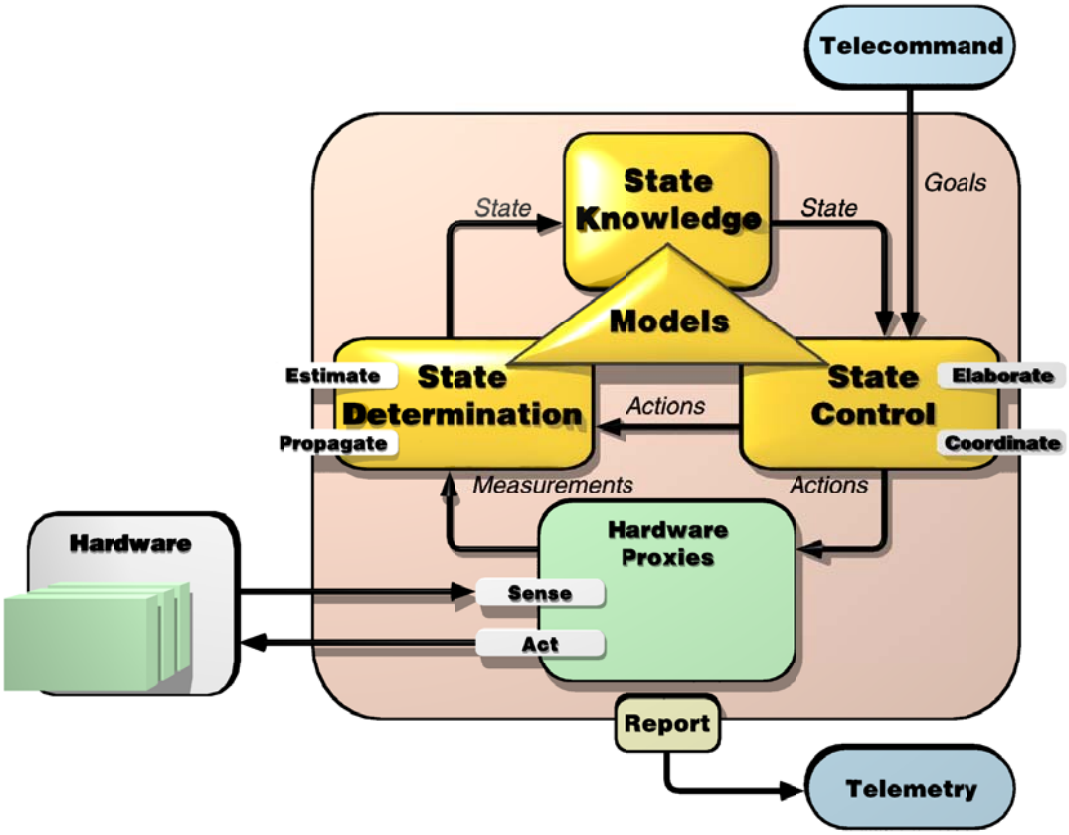

Figure 3. This diagram emphasizes several architectural themes: the central role of state knowledge and models, goal-directed operation, separation of state determination from control, and closed-loop control.

Theme: Explicit Use of Models

Express domain knowledge explicitly in models rather than implicitly in program logic—Much of what makes software different from mission to mission is domain knowledge about instruments and actuators and sensors and plumbing and wiring and many other things. This knowledge includes relationships such as how power varies with solar incidence angle, conditions such as the fact that gyros saturate above a certain rate, state machines that prescribe safe sequences for valve operation, and dynamic models that predict how long a turn will take. Conventional practice has been to develop programs whose logic implicitly contains such domain knowledge, but this expresses the knowledge in a "hidden" form that is hard to validate and hard to reuse. In fact, it is often hard to discern even that some assumed domain knowledge has been applied. One might see in the code, for instance, that an important command is issued twice and gather nothing further from it. Behind this innocent act, however, is a presumed attempt to be sure the command takes effect, which implies further that commands are assumed unreliable, but not so much so that the likelihood of a second failed command is acceptably small. This leads one immediately into questioning the nature and validity of this assumed model, which is nowhere to be found.

In contrast, MDS advocates that domain knowledge be represented more explicitly in *inspectable models*. Such models can be tables or functions or rules or state machines or any of several forms, as long as they separate the application-specific knowledge from the reusable logic for applying that knowledge to solve a problem. The task of customizing MDS for a mission, then, becomes largely a task of defining and validating models.

Theme: Goal-Directed Operation

Operate missions via specifications of desired state rather than sequences of actions—Traditionally, spacecraft have been controlled through linear (non-branching) command sequences that have been carefully designed on the ground. Moreover, most commands only specify actions to take — usually in an open loop manner, and often under assumptions of a particular prior state. Such design is difficult for two reasons. First, ground must predict spacecraft state for the time at which the sequence is scheduled to start, and that's difficult to know with confidence because of flight/ground communication limitations (data rate and light-time delay). Second, in the event that the actual spacecraft state is significantly different than the predicted state at any time during execution, the sequence should be designed to fail rather than chance doing something harmful. This is usually accomplished outside the sequence in a separate concurrent fault monitoring system, which then steps in after the sequence is terminated to impose a substantially different model of control on the system — one generally incompatible with sequencing.

MDS, in contrast, controls state—both flight and ground state—via "goals". A *goal* is defined as a prioritized constraint on the value of a state variable during a time interval. The time interval is allowed to float, subject to temporal constraints. A goal differs from a command in that it specifies *intent* in the form of desired state. Such goal-directed operation is simpler than traditional sequencing because a goal is easier to specify than the actions needed to accomplish it. Importantly, goals specify only success criteria; they leave options open about the means of accomplishing the goal and the possible use of alternate actions to recover from problems. A goal is a unifying concept that encompasses daily operations, maintenance and calibration, resource allocation, flight rules, and fault responses. Of course, all of this begs the question of who or what elaborates a goal into a program of actions, which brings us to goal-achieving modules and closed-loop control.

Theme: Closed Loop Control

Design for real-time reaction to changes in state rather than for open-loop commands or earth-in-the-loop control—Goal-directed operation implies closed-loop control because a goal, like a "set point" for a conventional controller, only specifies desired state, but not the actions needed to accomplish it. In MDS goals are issued to *goal-achieving modules* (GAMs). A GAM controls state by comparing estimated state to desired state, then deciding how to change the state if necessary, then issuing either sub-goals (with appropriate temporal constraints) to lower-level GAMs or issuing direct low-level actions (i.e., primitive actions). When a GAM accepts a goal it takes on the responsibility to either achieve the goal or report that it cannot. A GAM's logic can be arbitrarily simple or sophisticated, but it must always keep the goal issuer informed about the goal's status.

Many GAMs achieve their goals by issuing sub-goals, creating a hierarchy of GAMs based on delegation of control. Naturally, the bottom layer of GAMs issues only primitive actions. Importantly, GAMs can report why they acted as they did in terms of differences between estimated state and desired state (both available on state timelines), and what sub-goals or commands were issued in response.

A GAM is inherently self-checking, by definition, since it must monitor whether it is achieving each goal that it has accepted, keeping goal status up-to-date. During system testing this self-checking nature of GAMs considerably simplifies the job of analyzing test results; unexpected goal failures and unexpected goal successes (as when running fault scenarios) highlight areas for human inspection. Similarly, during mission operation, system behavior can be largely understood through the status of goals in a control hierarchy.

Theme: Real-Time Resource Management

Resource usage must be authorized and monitored by a resource manager—"Resources" are things like available battery energy, power, fuel, memory, thermal margin, etc. They are any state, in fact, that is affected by other states in a potentially conflicting way. Overuse of spacecraft resources can be disastrous, such as accidentally using too much power near the time of a critical orbit insertion maneuver, causing the spacecraft power bus to trip. For reasons like this ground operators have tended to be very conservative about resource usage, especially given their time-delayed knowledge of it. However, such conservative operation limits the amount of science data acquisition and return, especially during periods of great opportunity, such as during a fly-by or a short-lived science event.

MDS avoids this kind of operational dilemma through a resource management mechanism that prevents overuse, even if a resource is accidentally oversubscribed. Specifically, resource-using activities are forced to obtain a "ticket" in order to use a given resource, much as one obtains a file descriptor in order to access a file. An activity must state to a resource manager the amount of resource and the time interval when it is needed, and the ticket is issued only if the usage does not conflict with any other higher-priority usage. Further, if measurements show that more of a resource is being used than was ticketed (such as might occur from an unexplained power drain), the manager can disable one or more tickets until an adequate margin is recovered. Because a resource manager always knows the available amount, other activities can be triggered to opportunistically use the resource, thereby increasing science data return.

A resource manager is just another GAM, except that it deals in constraints on allowable states instead of constraints on the state itself. Issuing a ticket is the means by which it exercises this control. By treating resource management in this way it becomes a participant in the larger state coordination process, rather than a separate additional mechanism.

Theme: Separate State Determination from State Control

For consistency, simplicity and clarity, separate state determination logic from control logic—It's not unusual to see software that co-mingles control logic with state determination logic, but this practice is usually a bad idea for three reasons. First, if two or more controllers each make their own private determination for the same state variable, their estimates may differ, potentially leading to conflicting control actions. Second, mixing two distinct tasks in the same module makes the code harder to understand and less reusable. Third, these two tasks are an ill fit in the same module because control has a hierarchical structure based on delegation whereas state determination has a network structure based on pathways of interaction mechanisms (electrical, thermal, etc.).

Architecturally, MDS separates state determination from state control, coupled only through state variables. State determination is a process of interpreting measurements to generate state knowledge, and the process may combine multiple sources of evidence into a determination of state, supplied to a state variable as an estimate. Control, in contrast, attempts to achieve goals by issuing commands and sub-goals that should drive estimated state toward desired state. Keeping these two tasks separate simplifies design, programming, and testing, and also allows for independent improvements.

An added benefit is to avoid the temptation often encountered in designs to warp an estimate to meet the objectives of control. For instance, in order to attenuate a controller's superfluous reactions to noise, an estimate might be smoothed by lowering gains in the estimator. Not only does this link competing performance criteria in a single parameter, but now the system is deprived of accurate information about this state. Keeping state determination separate does not prevent this distortion, but it does express state knowledge in a public and uniform manner that permits a consistent pattern of testing designed to identify such breaches.

Theme: Integral Fault Protection

Fault protection must be an integral part of the design, not an add-on—Fault protection, which includes fault detection, localization, and recovery, has generally been treated as an add-on to a basic command & control system. As such, it was designed as an adjunct to the control system and usually arrived later in the project cycle. Such was the case for the Cassini attitude and articulation control system, and an interesting thing happened when fault protection was first enabled: numerous faults were detected in a control system that had already undergone a fair amount of testing. The Cassini AACS team learned more in that month than they had in the previous six months because they finally had independent detailed monitoring of system behavior.

Table 1. Fault protection is an integral part of design, not an add-on; its elements appear throughout the architectural elements.

General:	**State Determination:**
event logging	estimate uncertainty
assertion violations	fault detection
out-of-memory	diagnosis
Models:	health states
error states	**State Control:**
anomalous transitions	safety goals
failure probabilities	reactions to abnormal states
failure modes	reporting goal failures
	replanning

In MDS fault protection will be an integral part of the design—not an add-on—because it is an essential part of robust control and because it is extremely valuable during system testing. Goal-achieving modules in MDS need at least some minimum level of fault detection since they *must* report when an active goal is not being achieved. GAMs may also provide recovery strategies ranging from very simple to very sophisticated. In any case, this is always accomplished entirely within the same context and framework as normal operations, and it permits fault recovery that restores disrupted operations. Re-establishing a sequence after a fault is no longer a heroic effort. It is simply the way the system works all the time.

Theme: Acknowledge State Uncertainty

State determination must be honest about the evidence; state estimates are not facts—State values are rarely known with certainty, but a lot of software effectively pretends that they are by treating state estimates as facts. However, disastrous errors can result when control decisions are based on highly uncertain state. For example, it is probably unwise to perform a main-engine burn when the estimated position of the engine gimbals is below some minimum certainty. Uncertainty can arise in several ways, sometimes as conflicting evidence, sometimes through characteristic degradation of sensors, and sometimes during periods of rapid dynamic change.

MDS takes the position that a level of certainty should accompany every state estimate. State determination must be honest about what the evidence is telling it. If there are two credible pieces of evidence that conflict, and there's no timely way to reconcile the conflict, then the resulting state estimate must have an appropriately reduced level of certainty. Similarly, control must take into account the certainty level of the state estimates upon which it is basing a decision. If certainty drops below some context-specific minimum, then control must react appropriately, perhaps by attempting an alternate approach or by abandoning a goal entirely.

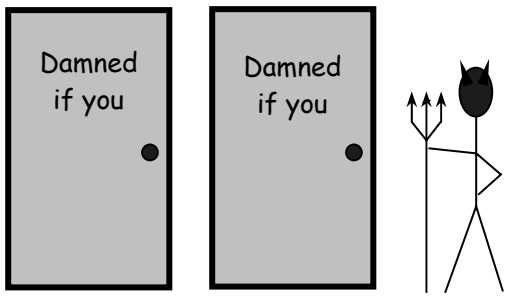

Figure 4. An architecture that doesn't express the amount of uncertainty in state estimates prevents control systems from exercising caution during periods of higher-than-acceptable uncertainty.

Uncertainty can be expressed in a number of ways, ranging from complete probability distributions to a simple enumeration of possible values. It isn't essential that the representation be rigorously statistical; often a heuristic criterion will do. The only rule is to represent the uncertainty in state knowledge effectively.

Theme: Separate Data Management from Data Transport

Data management duties and structures should be separated from those of data transport—Flight/ground data management has long been tightly coupled with data transport issues, largely because such capabilities evolved from a time when flight processors were extremely limited. This resulted in an architecturally flattened implementation approach where, for example, application code was built around the CCSDS packet format. While such designs had some justification in the speed and memory constraints of earlier missions, the time has come to adopt a cleaner layered separation and prepare for the day when spacecraft are in fact nodes in an inter-planetary network.

MDS clearly distinguishes between *data management* and *data transport*. The former elevates data products as entities in their own right, as objects and files that can be updated and summarized and aged, and that may or may not be destined for the ground or some other recipient. In fact, data management is a service that transcends the flight-ground divide so that data products are treated consistently in all places. Data transport, in contrast, can access any data product and serialize it for transport between flight and ground. Packet formats and link protocols are completely hidden from the level of data management. Decoupling these two capabilities keeps the design and testing simpler for each and allows for independent improvements. In this area MDS applies a layered organization [3].

The separation of data management and data transport also allows more intuitive management of the spacecraft to ground link. For example, in Mars Pathfinder the majority of the telemetry data was images of the surface of Mars. These images were identified uniquely in the planning process, the commands to create them sent to the spacecraft (the commands included the image identifier), and the onboard software took the image and created a data product. The image data product consisted of the image (raw or compressed), the image identifier, the time the image was taken and other ancillary information such as the exposure time, filter wheel position, etc. The onboard software then broke the image data product into CCSDS packets that were stored until sent as telemetry. The data transport system, both on the spacecraft and on the ground, coordinated the retransmission of lost or missing elements of the packet stream. When a missing packet was detected a request (command) could be sent to the spacecraft to effect the retransmission. Unfortunately, this mechanism proved very hard to manage because the mapping from packet sequence number to image identifier was difficult to determine and because of the spacecraft/Mars/Earth alignment dictated a single telemetry session per operation. Furthermore, the importance of a specific image to the overall science objectives would change based on the importance of the subsequent days' activities.

A further value in decoupling occurs due to the fundamental difference in nature of these two domains at the level of system coordination. In its fullest realization, space data transport must take its place as a peer in the larger data transport network covering everything from local hardwire communication within a system, to proximity links between sister vehicles, to lander-orbiter links at another planet, to links between planets. Spanning this vast range with its attendant physical exigencies makes quality of service a vital component of any data transport framework that is attempting to pull such a network together. Quality of service, in turn, must be a visible participant in the coordination of system activities, whether it be configuring radio equipment, pointing an antenna, or evaluating link characteristics. As prescribed, this is accomplished through goals. That is, quality of service is a state.

Data management, too, resides in a physical realm that links its actions to the rest of the system and demands that it participate in coordination processes. In this case the status of data products (existence, content, size, importance, location, and so on) collectively comprise a set of states. Since the goal of most JPL missions is the collection of data, goals on these data product states become the key driver to most other mission activities. Interactions and trades between them can be coordinated entirely within the goal-directed architecture.

Bringing the whole of system functionality within the fold would not be possible without the clean separation of data management from data transport.

Theme: Join Navigation with Attitude Control

Navigation and attitude control must build from a common mathematical base—Navigation and attitude control have been weakly coupled on most JPL missions because, in empty space, they operate on vastly different time scales and their dynamics usually don't greatly affect each other. As such, navigation software and attitude control software have been largely independent development efforts and the interfaces between them have been ad hoc. In upcoming missions, however, the coupling becomes much tighter. For example, escape velocity near an asteroid is so small that firing thrusters for attitude control can significantly affect the trajectory of an orbiting spacecraft. Likewise, docking with another vehicle, as in a sample-return mission, requires navigation and attitude corrections on similar time scales.

The approach that MDS is taking here, as in other areas, is to design common architectural mechanisms for common problems. Since the same forces influence navigation and attitude control, the architecture must allow for a common

model; since both are solving geometry problems, the architecture must provide for common solvers.

Theme: Instrument the Software

Instrument the software to gain visibility into its operation, not just during testing but also during operation—Perhaps the most vexing problem that operators face during a mission emergency is in not having enough information about what's happening inside the spacecraft to explain some anomalous behavior. Software developers face the same problem during system testing (albeit in less stressful circumstances) and they traditionally address the problem by adding temporary, ad hoc "instrumentation", i.e., software instructions that output some internal state. Such instrumentation is often removed later, either because it generates too much output or because it reduces performance or because it bypasses the downlink subsystem, outputting directly to a testbed console. Such adding and removing of temporary code is messy at best and error-prone at worst.

To address the need in a standard way—as Mars Pathfinder and Deep Space One did—MDS defines an "event logging framework" (ELF) that provides a standard mechanism for logging noteworthy events, whether generated on the flight side or ground side. Importantly, ELF allows operators to control the nature and amount of logging by controlling "entry policy" parameters such as event severity level, event IDs, and event reporting frequency. ELF reporting functions are designed for speed in discarding events disabled by the entry policy, so instrumentation (i.e., ELF calls) can generally be left in place, even in flight code. The net result is that software instrumentation is encouraged because it can be controlled at run time by ground operators and therefore can remain as a permanent part of the software, providing value not just during system testing but also during mission operations.

Logically, ELF provides a specialized interface to Data Management and therefore capitalizes on its capabilities for accumulating value histories, for summarizing them, and for discarding old and/or less important data products in order to make room for new data products.

Theme: Upward Compatibility

Design interfaces to accommodate foreseeable advances in technology—MDS is intended to serve missions for many years to come, and during that time there will be numerous advances in software technology for control systems, fault detection & diagnosis, planning & scheduling, databases, communication protocols, etc. MDS must be prepared to exploit such technologies else it will become an obstacle rather than an enabler for increasingly challenging missions, but MDS also needs to maintain some architectural stability to amortize its cost over its missions. The strategy for achieving this centers around careful design of architectural interfaces, behind which a variety of technical approaches can be used. Specifically, MDS designers consult with researchers to understand how software interfaces may need to evolve, and then implement a restricted subset of an interface using current mission-ready technology. When the more advanced technology becomes mission-ready, they implement the fuller interface in an upward compatible manner, namely, in a manner that still works for interface clients that use the restricted subset. Thus, interface client software is not forced to change on the same schedule as interface provider software.

The value of upward compatibility is powerfully illustrated in the history of IBM. In 1964 when IBM introduced the System/360 architecture, they transformed the computer industry with the first "compatible" family of computers. Software and peripherals worked virtually interchangeably on any of the five original processors, so customer investments were preserved when they upgraded to a more powerful processor. IBM continued to improve the technology over the years, but always within the System/360 architecture and its extensions. Although the MDS architecture applies to a much smaller marketplace, the benefits of upward compatibility make sense for MDS customers *and* providers.

5. Customer Benefits

The main value of MDS is that it should enable customer missions to focus on mission-specific design and development without having to create and test a supporting infrastructure. Customers will receive a set of pre-integrated and pre-tested frameworks, complete with executable example uses of those frameworks running on a simulated spacecraft and mission. These frameworks will be based on an object-oriented design described in Unified Modeling Language (UML) [5], the *lingua franca* of MDS software design and scenario description.

As a project, MDS is balancing a long-term architectural vision against a near-term commitment to its first customer mission, Europa Orbiter, scheduled to launch in 2003. Such commitments help focus MDS design efforts on pragmatic, well-understood mechanisms for supporting the architectural themes.

6. Related Work

Software Architecture

As Shaw and Garlan have wisely observed, "good architectural design has always been a major factor in determining the success of a software system" [3]. Of four major categories of activities is software architecture, MDS is squarely in the category of "frameworks for specific domains", in the same vein as DARPA's domain-specific software architecture (DSSA) program [1,2]. The MDS frameworks are being described in Unified Modeling Language (UML) [4], while recognizing that it has limitations as an architecture description language, most notably for its lack of descriptions of connectors and

interfaces as first-class entities and descriptions of hierarchical organization [5].

Shaw and Garlan also note that most systems typically involve a combination of architectural styles [3, chapter 2], and that's certainly true of MDS, given its scope as a unified flight-ground control and data system. Perhaps the most conspicuous architectural styles in MDS are closed-loop process control (in support of conventional feedback control systems as well as goal-achieving modules), state transition systems (in support of "reactive" discrete-state control systems), and hierarchical combination (reflecting delegation of sub-goals by goal-achieving modules). Also, the style of data abstraction and object-oriented organization pervades the MDS architecture.

Remote Agent Project

In a 1995 joint study between NASA Ames and JPL known as the New Millennium Autonomy Architecture Prototype (NewMAAP) a number of existing concepts for improving flight software were brought together in a prototype form. These concepts included goal-based commanding, closed-loop control, model-based diagnosis, onboard resource management, and onboard planning. When the Deep Space One (DS-1) mission was subsequently announced as a technology validation mission, the NewMAAP project rapidly segued into the Remote Agent project [6]. In May 1999 the Remote Agent eXperiment (RAX) flew on DS-1 and provided the first in-flight demonstration of the concepts. The MDS project, which is populated with many people who worked on or with RAX, was established in April 1998 to define and develop an advanced multi-mission data system that unifies the flight, ground, and test elements in a common architecture. That architecture is shaped with the themes described in this paper, some of which were explored and refined by the RAX experience.

Altairis Mission Control System

The Altairis Mission Control System (*Altairis MCS*) is a distributed, object-oriented commercial-off-the-shelf (COTS) satellite command and control system [7]. *Altairis MCS* is positioned as a ground-based mission control system built on a fully distributed CORBA-compliant architecture that can be distributed across networks of mixed UNIX and Windows NT computers. At a conceptual level, *Altairis MCS* shares some of the themes described in this paper. In particular, it emphasizes the central role of state and models in organizing a control system, with mission-specific extensions clearly separated from core software. Extensions primarily consist of finite state models—organized hierarchically—along with procedural scripts and data definition files. State transitions are composed of a required entry state, target state, executor, and latency. Operation is based on commanding of user-defined state transitions, where a transition is immediately executed (transitions to the target state) only if the associated state machine is already in the required entry state. In comparison, MDS operation is based on issuance of goals, which are specifications of desired state for a time interval in the future, regardless of current state. In addition, temporally overlapping goals on the same state variable are allowed, as long as they are mutually compatible.

7. Acknowledgements

The research and design described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

References

[1] "Collected Overview Reports from the DSSA Project", Will Tracz, editor. Lockheed Martin Federal Systems, Owego, NY, September 1995.

[2] *Proceedings of the Workshop on Domain-Specific Software Architectures*, Software Engineering Institute, Hidden Valley, Pennsylvania, July 1990.

[3] "Software Architecture: Perspectives on an Emerging Discipline", Mary Shaw and David Garlan, Prentice Hall, 1996.

[4] The Unified Modeling Language User Guide, Grady Booch, James Rumbaugh, and Ivar Jacobsen, Addison Wesley, 1999.

[5] "Is UML an Architecture Description Language?" in *OOPSLA 99 Companion, Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Association for Computing Machinery, November 1999.

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

[7] Altairis Mission Control System, Altair Aerospace Corporation, http://www.altaira.com.

Daniel Dvorak is a researcher 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, as well as 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 Avionic Systems Engineering section 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.

Glenn Reeves is a senior engineer in the Autonomy and Control section at the Jet Propulsion Laboratory, California Institute of Technology, where he is a member of the Mission Data System project team. Prior to MDS he was the lead engineer for the Mars Pathfinder lander flight software development. Glenn holds a BS in computer science from California State Polytechnic University, Pomona.

Allan Sacks is manager of the Mission Data System Office in the Telecommunications and Mission Operations Directorate at the Jet Propulsion Laboratory, California Institute of Technology. Prior to MDS he managed the Mars Pathfinder Ground Data System project, 1992–1997. This work led to concurrent engineering activities at JPL among ground system, flight system, and mission operations that have been critical in reducing project risks and cost. Al also served as deputy manager of the Space Flight Operations Center Project, 1988–1992.