Introduction to MDS Architecture

Bob Rasmussen
Historical Context

• Until recently, JPL missions were one-of-a-kind, spaced many years apart
  • Each mission team developed flight software independently with minimal inheritance
    • Mars Pathfinder-to-Deep Space 1 "reuse" may be first exception
    • However, no a priori provisions for reuse were made

• Missions have been designed for human control from Earth
  • Large operations staff and budget
  • Intensive human planning & checking of spacecraft activities
  • Big gap between what operators want to say and what they have to say

• Flight software has used relatively simple time-based sequencing
  • Complicated sequence planning done on Earth, then uplinked
  • Lack of system-level reactive behaviors prevent full use of available resources

• Very little autonomy except for fault protection and a few "critical sequences"
  • Examples: Mars entry-descent-landing, Jupiter orbit insertion
  • Always a huge design effort, and typically done late in a project

• There is a big gap between systems engineering and software engineering
Pressures for Change

- New era of frequent launches
  - Low-cost missions cannot afford to start from scratch
  - Institution cannot afford costly point solutions or sub-optimal use of software engineers
  - Risks from low reuse are higher than necessary

- More *in situ* operations in uncertain environments
  - Rovers on Mars, landers on comets, aerobots in Titan’s atmosphere, hydrobots in Europa’s ocean, ...
  - Science goals depend more and more on autonomous operation

- More constrained communication with Earth demands more onboard decision-making
  - Longer round-trip light time delays (~10 hours at Pluto)
  - Lower data rates (~300 bps at Pluto with 2 m antenna, 5 W radiated power)
  - Limited viewing opportunities from landers (a few hours a day on Mars)

- Specter of mission-ending failures due to errors in software
  - Ariane 5, Clementine, Mars Polar Lander, ...
The MDS Vision

A unified architecture for flight, ground, and test systems that enables missions requiring reliable, advanced software

- Build a highly reusable core software system for a wide variety of space mission applications
- Promote modern, synergistic processes for systems and software engineering
- Establish an improved development life cycle for more reliable mission software
- Reduce development cycle time and cost
- Satisfy complex mission requirements (e.g., robust in situ exploration) and reduce operations cost with increased autonomy
This Presentation

- Summarize the key themes that have guided MDS development

- Describe two dominant architectures that shape the rest of MDS

- Establish a context for several of the detailed architectural features to be described later in the review

- Introduce the software organization that ties all of these pieces together

- Suggest how this helps us better relate software to systems engineering
MDS Architectural Themes
Themes
Organize Well

• Construct subsystems from architectural elements, not the other way around
  • Solve common problems with architectural patterns — then specialize

• A unified approach to managing interactions is essential

• Some things go together
  • Flight, ground, and test elements must be engineered as one system
  • Fault protection must be an integral part of the design, not an add-on
  • Navigation and attitude control must build from a common mathematical base

• Others do not
  • For consistency, simplicity and clarity, separate state determination logic from control logic
  • Separate data management duties and structures from those of data transport
Themes

Emphasize Operability

• Be explicit
  - System state and models form the foundation for monitoring and control
  - Express domain knowledge explicitly in models rather than implicitly in program logic
  - Operate missions via specifications of desired state rather than sequences of actions
  - State determination must be honest about the evidence; state estimates are not facts

• Close the loop
  - Design for real-time reaction to changes in state rather than for open-loop commands or earth-in-the-loop control
  - Resource usage must be authorized and monitored by a resource manager
Themes
Think Ahead

- Enable migration of capability from ground to flight, when appropriate, to simplify operations

- Design interfaces to accommodate foreseeable advances in technology
Managing Interactions

"A unified approach to managing interactions is essential"

- Interactions make software difficult
  - Elements that work separately often fail to work together
  - The combinatorics of interaction is staggering, so it's not easy to get right
  - This is a major source of unreliability
- There are two approaches to this in MDS:

**State-Based Architecture**
- Handles interactions among elements of the system under control
- Outward looking
- Addresses systems engineering issues

**Component-Based Architecture**
- Handles interactions among elements of the system software
- Forward looking
- Addresses software engineering issues
The State-Based Architecture
State-Based Architecture

[Diagram showing a state-based architecture with MDS, Report, State, and Control.]
State is Central

- A **system** comprises project assets in the context of some external environment that influences them
- The function of mission software is to monitor and control a system to meet operators' intents
- MDS manages all essential aspects of this function via **state**
  - Knowledge of the system, including its environment, is represented over time in **state variables**
  - The behavior of the system is represented by **models** of this state
  - Interaction with the system is achieved via modeled relationships between state and interface data (**measurements** and **commands**), as mediated by **hardware proxies**
  - Information is reported, stored, and transported as **histories** of state, measurements, and commands
  - Operators' intent, including flight rules and constraints, are expressed as **goals** on system states
A High Level View
State Knowledge

Everything You Need to Know

- Dynamics
  - Vehicle position & attitude, gimbal angles, wheel rotation, ...
- Environment
  - Ephemeris, light level, atmospheric profiles, terrain, ...
- Device status
  - Configuration, temperature, operating modes, failure modes, ...
- Parameters
  - Mass properties, scale factors, biases, alignments, noise levels, ...
- Resources
  - Power & energy, propellant, data storage, bandwidth,
- Data product collections
  - Science data, measurement sets, ...
- DM/DT Policies
  - Compression/deletion, transport priority, ...
- Externally controlled factors
  - Space link schedule & configuration, ...

... and so on
State Determination
Making Sense of the World

• One can act only on one’s knowledge of the system
  • Knowledge is what you know, not how you know it
  • Observations (e.g., measurements) are not knowledge

• Estimators find “good” explanations for observations and other evidence, given a model of how things work
  • Knowledge may be propagated into the future, given models and plans

• All knowledge is uncertain
  • Judgment must be based both on what is known, and on how well it is known

• However, one can achieve local consistency of knowledge
State Control
Closing the Loop

- Operators express their intent in the form of **goals**
  - Goals declare *what* should happen, **not how**
  - Goals may be expressed at any level
- High level goals are elaborated recursively into lower level goals
  - **Elaboration** may be conditional, in order to react to present circumstances
  - **Coordination** of activities is accomplished by **scheduling**
  - Conflicts are resolved, with priority as final arbiter
- Knowledge of all states is maintained, as required to achieve goals
  - Knowledge is compared to goal constraints to test for compliance
- Corrective action is applied, as required to achieve goals
  - Alternate methods of **achievement**
    - may be applied at any level
  - Unachievable goals (and their elaborations) are dropped individually without sacrificing others
- **Supports fault tolerance, critical activities, *in situ* autonomy, opportunistic science, and more**
Models
Tying It All Together

- Relationships among states
  - Power varies with solar incidence angle, temperature, and occultation
- Relationships between measurement values and states
  - Temperature data depends on temperature, but also on calibration parameters and transducer health
- Relationships between command values and states
  - It can take up to half a second from commanding a switch to full on
- Sequential state machines
  - Some sequences of valve operations are okay; others are not
- Dynamical state models
  - Accelerating to a turn rate takes time
- Inference rules
  - If there has been no communication from the ground in a week, assume something in the uplink has failed
- Conditional behaviors
  - Pointing performance can’t be maintained until rates are low
- Compatibility rules
  - Reaction wheel momentum cannot be dumped while being used for control
  ... and so on
Hardware Proxies

Connecting With the World

• Provide local software representatives of system hardware
  • Delineating the abstract model of the system \textit{(including time!)}
  • Translating raw input/output data into abstract declarations about state
    • \textbf{Measurement models} relate incoming data to state
    • \textbf{Command models} do the same for outgoing data

• Augments system hardware with supplemental behaviors
  • Sampling
  • Time and metadata tagging
  • Data format translation
  • Local tight control loops
  • Data compression
  • I/O sequencing and synchronization
  • Data buffering and routing
  • Error checking
  • Data preprocessing
  • Etc.

• Isolates state frameworks from platform specific interfaces
  • Built on ACE middleware
  • Real, simulated, or abstract hardware
  • Real or virtual time
State Timelines

- **State timelines** maintain the value or set of possible values (e.g., a range) of a state variable as a function of time.
- They capture both knowledge and intent about state.
State Knowledge

- Knowledge of the system is expressed by generating **state functions**
  - Each spans an **interval of time**
    - Intervals spanning past times express experience
    - Intervals spanning future times express expectations
  - Each bounds the possible values of a state variable over its time interval
    - The state's **value** is assumed to be somewhere within this **uncertainty**
    - Expressed values can vary over the interval

- Each state timeline is covered by a contiguous series of state functions for all time

- Newly created state functions overlay or replace older ones
  - Estimators produce new state functions which improve old knowledge
  - Other mechanisms produce newer state functions which compress or summarize older knowledge
State Intent

- Control is exercised over the system by imposing ...
  - **Constraints on states**, which limit the range of a state variable
    - State is allowed flexibility within these bounds
  - **Constraints on time**, which limit the duration between two **time points**
    - Time points are variable points in time
    - These times are allowed flexibility, but again, with constraints

- A state constraint between two time points is called a **goal**
- A time constraint between two time points is called a **temporal constraint**

- Goals and temporal constraints are expressions of **intent**

- Success in constraint achievement is an objective matter
  - Criteria are explicitly expressed in constraint evaluation code
  - Directly verifiable during test, since constraints are explicitly evaluated
Constraint Networks

- Goals and temporal constraints each connect a pair of time points
- Time points are often shared (e.g., one beginning as another ends)
- A collection of connected goals and temporal constraints form a constraint network
Resolving Conflicts

- Example: three goals on the same state

The constraint

The time interval

Goal 1

Goal 2

Goal 3

Goals 1 and 2 overlap, so they're compatible, as is

Goal 3 is incompatible with Goal 2, but it can wait

Crosshatched areas are outside goal constraints
Timeline Execution

- Goals are accepted if successfully placed on the timeline for the goal state variable.
- Goals are frozen and acted upon when they appear on the timeline in the immediate future.
- Goals are acted upon by **achievers** assigned to each state variable.
- Elaborators monitor execution and adapt plans, as necessary.

... given the present goals ...
... and given the present state, ...
... achieve the goals.

Intent

Knowledge

Time
Putting It Together

- Elaborators, scheduling, ...
  - Goal/event-driven
  - Planning and constraint solving
  - Analogous to sequencing, mode and configuration control, fault responses

- Achievers, DM/DT, ...
  - Provide system behaviors
  - Managed via goals and temporal constraints
  - Fairly conventional real-time monitoring and control processes
Allocation Goals

- "Normal" constraint goals confine a state within some range
  - They limit states whose effects on others must be controlled
  - They merge via their intersection
- Allocation goals set limits on how tight normal constraints can be
  - They indicate which effects on states by others must be allowed
  - They merge via their union
- Both types of goals have been rigorously captured within a common theory

- The complementary declarations of intent and indulgence that constraint and allocation goals provide are highly expressive
- Allocation goals provide the means to...
  - Manage resources and coordinate their use
  - Express and accommodate changing error budgets
  - Make allowances for uncertainty
  - Address conflicting side effects of otherwise disparate activities
  - Delegate control authority
Delegation

• Delegation temporarily moves the locus of goal achievement from nominally assigned achievers to others

• Addresses various problematic situations
  • Accommodation of real time limitations in goal elaboration and scheduling
  • Reflexive response to emergency situations
  • Consolidation of authority, when coordinated control is required
  • Direct, but protected, access by test operators to low level control functions
  • Additional implementation flexibility
    • Not always necessary to express complex or precise details in the constraint network

• Yet does so completely under the cognizance of the constraint network with its associated rules

• Enabled by the existence of allocation goals as equal participants in the constraint network
Events

- For controllable states...
  - Choices made by achievers may be arbitrary within the confines of goals and temporal constraints
  - Particular states and times within constraints are selected by the system

- For uncontrollable states...
  - Constraint satisfaction occurs at the impetus of external forces
  - Particular states and times at which constraints are met cannot be selected by the system

- However, goals constraining uncontrollable states may appear in a constraint network

- In this case, they act as event definitions
  - By becoming true (e.g., altitude is less than 10 km) and triggering time points
  - Or becoming false (e.g., a device is no longer healthy) and triggering goal failures

- System behaviors can be tied to these events
Value Histories

- A container mechanism supporting functions that produce values over time (state variable timelines, measurements, commands, ...)
- Encapsulate the interface to data management persistent storage and data transport
  - Stored and transported as data products
  - Selected data products are preserved across resets
- Leverage the use of models to preserve continuous information using less storage space
- Can also simply store a set of discrete value instances
- Controlled by storage and transport policies

Entries are combined and compressed as they age and are eventually deleted
Component-Based Architecture
Components are Fundamental

- The Component Architecture establishes the elements of software design and their coherent integration
- Components and their connections embody...
  - The elements of functionality
  - Their types and registered instances within a deployment
  - Their interfaces and distribution across platforms
  - Their coordinated execution and synchronization
- These issues are raised to the level of symbolic realization
  - Software organization is established independently and systematically
  - It can be manipulated directly — including at run time, if necessary
  - Complexity becomes a manageable entity

- The State Architecture establishes the elements of functionality and their functional relationships
  - E.g., state variables, achievers, hardware proxies, and so on
- It does not establish the software design
Connection Rules

- Functional elements of the State Architecture are structural elements in the Component Architecture
  - State variables, achievers, hardware proxies, and so on, are Components

- State Architecture elements all interrelate in a few formally established patterns
  - E.g., measurements are used only by estimators, goals are directed to state variables, only controllers issue commands, only estimators update state knowledge, and so on
  - These are rules on connections within the Component Architecture of the design

- The Component Architecture implements and enforces these patterns
  - Compliance is inspectable
  - Exceptions must be overtly managed — nothing is hidden
Deployments

- A deployment is an executable product
  - Each project will have several deployments
  - E.g., the flight software, the simulation software during test, parts of the ground software, and so on

- Each deployment is constructed from components, connected as appropriate for that application
  - Not every component belongs in every deployment
  - E.g., attitude is usually estimated only on board, while trajectory is usually estimated only on the ground

- Deployments may be interconnected

- For remote links, deployments communicate via component proxies
  - Exchanges between a component and its proxy are managed by data transport services
For example...

Ground↔Flight Knowledge Exchange

• State knowledge is needed in both places
  • Common representation
  • Coordinated, consolidated & maintained, as appropriate

• Information is exchanged via state variable proxies
  • Original source in one deployment
  • Copied (at some level) to a proxy in the other

• Ground-based state determination is...
  • Typically for things like orbit determination, calibration, ...
  • Up-linked as necessary (trajectories, parameters, ...)

• Flight-based state determination is...
  • Typically for things like attitude determination, device states, faults, ...
  • Down-linked as available (part of telemetry)
Functional Partitioning Across Deployments

- There are similar stories of data exchange for goals, measurements, science data, and so on

- The architecture will support…
  - Knowledge sharing across multiple deployments
  - Coordinated agents exchanging goals as peers

- Where latency is not an issue…
  - Measurements in one deployment may be sent to estimators in another
  - Controllers in one deployment may send commands to hardware proxies in another
  - Goals may be elaborated part way in one deployment, and completed in another
The State and Component Architectures are defined within a set of classes called the **MDS Framework**

- Frameworks are the elements of a partially complete application
- The MDS framework is organized in a hierarchy of dozens of packages
- Each project *adapts* the framework by extending it in mission-specific ways
The MDS Common Model

- The MDS Framework is the collection of most core classes within the MDS architecture
  - Developed and maintained exclusively by MDS
  - Uniform (except for versioning) across MDS adaptations

- Each project does an Adaptation of the framework
  - Captures project requirements and scenarios
  - Extends framework classes to address functions and configurations specific to the project
  - Reusable extensions are generalized (if necessary) and moved to the framework

- Several Deployments of the adaptation are defined
  - These are the executable configurations to be used in various settings (test beds, flight, ground, etc.)
Reuse Among Projects

- Each project uses the same framework, except that later projects will adapt later versions
  - Can continue to track framework evolution up to some freeze point
  - Updates to frozen version are confined to that project
    - Though mainline framework development may decide to make some of the same updates
- Projects can adapt from one another
  - A similar track-then-freeze configuration management process would be necessary
The Framework

- Disciplines extend and customize the core infrastructure*
  - Partitioned as peers for modularity, acknowledgement of discipline vagaries, and the ability to aggregate functionality across disciplines as necessary

* Infrastructure
  - All of the classes embodying core, generic features, concepts, and services
  - Includes essential features of the State and Component architectures
  - Internally layered (hierarchical) to maximize reuse and uniformity, and to build more complex structure in manageable steps
Systems Engineering

- Systems and software engineering need to complement one another
  - Systems engineering must define the system and behavior
  - Software must understand the system and guide its behavior

- **State Analysis** is a model-based process defined by MDS to aid systems and software engineering
  - State analysis prompts comparatively methodical and rigorous analyses of systems
  - MDS permits the uniform expression of systems engineering concepts in software architectural terms
  - Due to the alignment of State and Component architectures, both functionality and software design are considered simultaneously
  - Resulting products map directly onto the MDS architectural elements
  - Most MDS adaptation requirements can be defined by state analysis

- State and Component architecture specifications are supported by tools, which will ultimately evolve into a unified code generation system for MDS
Summary

- MDS addresses...
  - Architectures for both functional and software design interactions
  - Unification and reuse across deployments and projects
  - A wide range of technical issues from autonomy to data management
  - The collaboration of systems and software engineering
  - Processes, tools, and design rigor up to the challenge of a flight program

- State and Component Architectures are the bedrock of our approach
  - Each exploits a relatively small but powerful set of ideas
  - The two architectures complement one another in a natural but far-reaching manner