Outline

• Position Statement

• Fundamentals

• Past Experience and Lessons Learned

• Current State of Practice

• Future Evolution

• Summary
POSITION STATEMENT
At JPL and across the spacecraft engineering community, the development of robust FDIR* capabilities has been more of an “art” than a “science”.

We posit that there is significant benefit to be gleaned from applying greater rigor and a more systematic approach to FDIR system development, and that the burgeoning field of Model-Based Systems Engineering can provide useful techniques and tools to help us in this endeavour.

* Note: In this package, we use the somewhat more general terms “Fault Management” and “Fault Protection”. There are subtle distinctions between each of these terms, which we can discuss offline if there is interest.
What is Fault Protection?

• As used and applied at JPL, Fault Protection is both:
  – A specific SE discipline (similar to EEIS or mission planning), whose activities are separately scheduled and tracked, and
  – The elements of a system that address off-nominal behavior

• “Fault Management” is becoming the preferred term within NASA
  – Fault Protection is functionally equivalent to “Fault Management*”, but suggests a flight system bias

• Focused on the flight system, Fault Protection includes
  – Flight system fault detection and response
  – Ground-based failure diagnosis and recovery
  – Ground-based contingency planning and action

* Per current draft of NASA-HDBK-1002, “Fault Management Handbook”; however, the definition remains in work…
Flight System Fault Tolerance Design Strategies include:
1) Graceful Degradation
2) Application of Redundancy
   - physical
   - functional
3) Fault Containment Regions
   - limit propagation
4) FDIR

In-Flight Examples include:
1) Robust Design Features
   - Simple Design (e.g. fixed solar array or HGA)
   - conservative design practices and performance margins
2) On-Board Autonomy
   - (e.g. attitude constraint checking)
3) Post-Launch Operations Processes and Procedures
   - These include the use of Flight Rules and executing commands on a spacecraft simulation test-bed.

For Example:
1) EDAC
2) FPGA TMR
3) Active Redundancy

Fault Protection Scope

Fault Protection

Fault Avoidance

Fault Tolerance

Fault Masking

FDIR

Flight HW FDIR

Flight SW FDIR

Ground-based FDIR

For Example:
1) Telemetry Alarm Checking
2) Telemetry Trending & Analysis
3) Contingency Plans & Procedures

For Example:
1) OV/UV Detect Circuits
2) Watchdog Timers
3) On-Chip Built-in-Tests
4) Memory scrubbing

For Example:
1) Sensor Threshold / Persistence
2) Diagnosis / Response
3) Re-plan & Continue Activity
PAST EXPERIENCE
Missions and Capabilities

- The set of missions historically flown by JPL has led to the development of robust autonomous FP capabilities
  - JPL FP designs and processes formed by experience and lessons learned (some painfully)

- FP capability fielded on Viking and Voyager, gradually increasing in scale to significant levels of complexity and autonomy
  - Cassini SOI is a good example of autonomous FP capability

- MSL represents the most complex FP system JPL has built, with 1097 system-level monitors and 38 system-level responses (plus on the order of 800 local responses)
Typical Constraints and Driving Requirements

• Operate with Limited Ground contact
  – Extended periods with no planned contact (1 to 4 weeks)
  – Planned contact periods may be short (1 to 2 hours)
  – Ground may not show for planned contacts (5% to 10%)
  – Large one-way light times (minutes to hours)
  – Low downlink data rates (10 to 40 bps)

• Protect fragile elements of systems

• Leverage existing flight system components

• Protect/complete critical activities
  – Orbit insertion, entry/descent/landing, irreversible deployments

• Long mission life
  – Survive without maintenance for primary missions lasting 5-11 years

• Harsh environments
  – TID of 100 krad to 4 mrad
Fault Protection “Family Tree”

*See reference [16], "OCE FSW Complexity Study, Appendix F."
FP “Family Tree” – Detail
JPL missions have suffered relatively few permanent faults
- Flight hardware for deep space missions has to be (and has been) very reliable

Fault protection activity during our missions has been most commonly caused by:
- Operator errors
- Fundamental design flaws, including software design flaws
- False alarms due to unnecessarily tight thresholds
- Unforeseen transient behavior due to interactions and/or variations in the operating environment, SEUs, etc.

Many examples where fault protection responded appropriately to transient behavior that was unexpected
- Magellan (1990 - 1992): Software flaw that caused heartbeat termination
- Cassini (1993): Attitude estimator transient during backup Star Tracker checkout
- MER Spirit Rover (2005): Potato-sized rock jammed in right rear wheel
- Dawn (2008): Cosmic ray upset of attitude control electronics
- Kepler (2009): Undervoltage due to unexpected power interactions at launch
PAST EXPERIENCE IN DEEP SPACE FAULT PROTECTION

PRESENT
On JPL flight projects, Fault Protection is a broad-based systems engineering task, and includes components of:

- **Mission Engineering**
  - Timeline, Nominal, Critical and Time-Critical Activities

- **Project System Engineering**
  - Systems Architecture

- **Flight System Engineering**
  - Failure Analysis
  - Requirement/Design Flow-down to FSW, Subsystem SE, Reliability
  - Design, Test, and Operation of On-Board autonomous Fault Detection, Isolation, and Response logic responsible for maintaining vehicle health and safety.
  - Hardware Redundancy is often included

- **Mission Operations**
  - Contingency Planning and Anomaly Resolution
  - Flight System Data Analysis and trending, state tracking, simulation

- **Mission Assurance**
  - Reliability Analysis, Parts Qualification, Environments etc.

The FP effort is often managed like a ‘spacecraft subsystem’.

- Reviews, budget/schedule (WBS), specific work products
- Keeps effort from being lost or or mismanaged
Characteristics of JPL FP Approach

• **Single-failure tolerance (SFT)**
  – No single point of failure will result in loss of mission
  – For some missions, waived in part or whole (e.g., single-string)

• **Limited use of reliability data**
  – JPL does not use reliability estimates as a basis for meeting single-failure tolerance requirements
  – Reliability estimates used for lifetime calculations
  – Reliability estimates used as supporting rationale in SFT waivers

• **Maintain failure tolerance after first failure**
  – Clear temporary failures
  – Maintain failure tolerance in safing modes
  – Robustness to multiple orthogonal failures
FP Across the Project Lifecycle

**Architecture**
- Pre-PHASE A
  - Formulation/Proposal
  - Initial Flight System Architecture

**Requirements**
- PHASES A/B
  - Concept Development Preliminary Design
  - PMSR
  - Develop Top-level Requirements
  - Develop Functional Requirements

**Analysis**
- PHASES C/D
  - Detailed Design Build, Test and Launch
  - PDR
  - Develop Detailed Req'ts
  - System-level Risk Analyses
  - Apply Systems Analysis Results to Design Req'ts

**Verify and Validate**
- PHASE E
  - Operations
  - CDR
  - Support Req't Verification
  - FP Ops activities

**Conduct Reviews**
- Pre-PHASE A
  - Formulation/Proposal
  - MCR/TMC

- PHASES A/B
  - Preliminary Design
  - PMSR

- PHASES C/D
  - Detailed Design Build, Test and Launch
  - PDR

- PHASE E
  - Operations
  - MRR
  - CERR

*See reference [9], “Fault Protection System Engineering: Tasks and Products Across the Project Lifecycle” for more detail.*
FM Completeness: Requires Top-Down and Bottom-up Analyses

**Top-down assessment**
- Develop necessary FDIR
- FDIR necessary to maintain acceptable functionality for each identified failure scenario

**Bottom-up assessment**
- list of local fault responses
- for each failure mode, identify failure effects
- FMEA, FTA
- for each failure effect, assess relevant mission phases/activities; add identified hazards

**Top-down assessment**
- analyze set of failure scenarios
- determine set of failure scenarios
- for each failure scenario, assess acceptability (FDIR vs. FEPT)

**Bottom-up assessment**
- determine fault set
- FMEA, FTA
- for each failure effect, assess relevant mission phases/activities; add identified hazards

**Top-down assessment**
- determine set of failure effects
- for each failure mode, identify failure effects

**Bottom-up assessment**
- develop necessary FDIR
- list of local fault responses
- for each failure mode, identify failure effects
- FMEA, FTA

**Top-down assessment**
- determine system functions
- functional analysis, FTA, HA, IHA
- identify state(s) associated with each function

**Bottom-up assessment**
- analyze set of success scenarios
- for each mission phase/activity, determine FDIR necessary to maintain acceptable function

**Top-down assessment**
- determine acceptable ranges
- determine the acceptable values of each state for relevant mission phases/activities (goals); acceptable values may change over course of mission

**Bottom-up assessment**
- determine states associated with each function
- for each failure scenario, assess acceptability (FDIR vs. FEPT)
- for each failure effect, assess relevant mission phases/activities; add identified hazards

FDIR necessary to maintain acceptable functionality for each identified failure scenario

**Top-down assessment**
- for each failure scenario, assess acceptability (FDIR vs. FEPT)
- FDIR necessary to maintain acceptable functionality for each identified failure scenario

**Bottom-up assessment**
- for each failure mode, identify failure effects
- FMEA, FTA
- for each failure effect, assess relevant mission phases/activities; add identified hazards

FM Completeness: Requires Top-Down and Bottom-up Analyses

**Top-down assessment**
- Develop necessary FDIR
- FDIR necessary to maintain acceptable functionality for each identified failure scenario

**Bottom-up assessment**
- list of local fault responses
- for each failure mode, identify failure effects
- FMEA, FTA
- for each failure effect, assess relevant mission phases/activities; add identified hazards

**Top-down assessment**
- analyze set of failure scenarios
- determine set of failure scenarios
- for each failure scenario, assess acceptability (FDIR vs. FEPT)

**Bottom-up assessment**
- determine set of failure effects
- for each failure mode, identify failure effects
- FMEA, FTA
- for each failure effect, assess relevant mission phases/activities; add identified hazards

**Top-down assessment**
- determine system functions
- functional analysis, FTA, HA, IHA
- identify state(s) associated with each function

**Bottom-up assessment**
- analyze set of success scenarios
- for each mission phase/activity, determine FDIR necessary to maintain acceptable function

**Top-down assessment**
- determine acceptable ranges
- determine the acceptable values of each state for relevant mission phases/activities (goals); acceptable values may change over course of mission

**Bottom-up assessment**
- determine states associated with each function
- for each failure scenario, assess acceptability (FDIR vs. FEPT)
- for each failure effect, assess relevant mission phases/activities; add identified hazards

**FM Completeness: Requires Top-Down and Bottom-up Analyses**

**Top-down assessment**
- Develop necessary FDIR
- FDIR necessary to maintain acceptable functionality for each identified failure scenario

**Bottom-up assessment**
- list of local fault responses
- for each failure mode, identify failure effects
- FMEA, FTA
- for each failure effect, assess relevant mission phases/activities; add identified hazards

**Top-down assessment**
- analyze set of failure scenarios
- determine set of failure scenarios
- for each failure scenario, assess acceptability (FDIR vs. FEPT)

**Bottom-up assessment**
- determine set of failure effects
- for each failure mode, identify failure effects
- FMEA, FTA
- for each failure effect, assess relevant mission phases/activities; add identified hazards

**Top-down assessment**
- determine system functions
- functional analysis, FTA, HA, IHA
- identify state(s) associated with each function

**Bottom-up assessment**
- analyze set of success scenarios
- for each mission phase/activity, determine FDIR necessary to maintain acceptable function

**Top-down assessment**
- determine acceptable ranges
- determine the acceptable values of each state for relevant mission phases/activities (goals); acceptable values may change over course of mission

**Bottom-up assessment**
- determine states associated with each function
- for each failure scenario, assess acceptability (FDIR vs. FEPT)
- for each failure effect, assess relevant mission phases/activities; add identified hazards
• **Success Trees**
  • Represent system functions and functional decomposition
  • Conditions for success; "light" side
• **Fault Trees**
  • Represent system functions and paths to failure of top event
  • Conditions for failure; "dark" side
• **Directed graphs**
  • Represent components and connections/interfaces
  • Modeling of physical and logical connections enables formal modeling of failure effect propagation
• **Failure Modes and Effects Analyses (FMEA)**
  • Description of the failure modes (mechanisms) and the immediate failure effect
  • Modeled failure effect propagation enables formal and complete development of all failure effects
• **Event Sequences**
  • Describes system functionality as a function of time
  • Provides "triggers" to enable/disable elements of directed graph representation
• **State Machines** (Not Shown)
  • Necessary to assess sequencing of system states, both nominal and off-nominal
Challenges for Current FM Approach

- No underlying, unifying model for various FM representations and bottom-up/top-down analyses
  - Difficult to be confident that the job is “complete” (enough)

- **Show quantitative benefits to support engineering trades**
  - Developing approaches to show value of additional HW and SW
  - Especially - assessing value of applying HW redundancy

- **Accurately estimate and control costs**
  - Better define products and processes, and process metrics

- **Perform adequate V&V**
  - Large failure space makes comprehensive testing infeasible
  - Working on tools and approaches to better verify and validate

- **Write relevant, decomposable requirements**
  - Needs to be more than “Do FP”
  - Better integration with SE requirements process
FUTURE EVOLUTION
• **Advancing the “Science” of Fault Management**
  – Formalization of concepts and terminology
  – Development of unified Theory of FM, leveraging prior work on state analysis and functional analysis

• **Improved Fault Management design process**
  – Integration of FM design into “mainline” systems engineering activities
  – Incorporation of top-down design approaches into sys eng process
  – Application of Model-Based Engineering (MBE) techniques to document FM design and enable difficult (or previously impossible) analyses
• **Resilient system architectures**
  – Development of system architectures that are inherently capable of fault avoidance, tolerance and recovery, rather than fault protection architecture as a “bolt-on” to nominal execution architecture.
  • Integration of fault protection within the nominal control loop
  • Continued migration of “cognizance” from operators to spacecraft

• **Advanced diagnosis & recovery algorithms**
  – Leverages recent advances in model-based reasoning, hybrid (discrete/continuous) system modeling, discrete-event systems and Integrated System Health Management (ISHM) communities
  – Challenges: modeling expressivity, coherent integration of multiple representations and techniques, and scalability to large-scale systems
Directions of Current Research (3)

• **Formal methods, automated analysis, autocoding**
  – System/software architecture specification languages (e.g., AADL, ACME), MBE, and model-driven software development provide greater opportunity for formal V&V techniques, automated analysis and automated code generation
  – Building up “libraries” of code-generation patterns for use in future missions

• **Fault management design environments**
  – Development of model transformation technologies to integrate general-purpose MBE languages (e.g., SysML) & tools with FM-specific design environments (e.g., TEAMS, SAFIRE)
  – Eventual automation of generation of FM analysis artifacts (e.g., FT, FMECA)
Summary

Past:
• JPL has a long history of developing, deploying and operating effective Fault Management capabilities on its spacecraft
• Our FM capabilities have evolved as our missions have become increasingly ambitious and complex, but this evolution was not rigorously “architected” over time

Present:
• JPL Fault Protection philosophies and goals are relatively straightforward and generally consistent from project to project
• FP engineers end up knowing how the Flight System really works (and how it doesn’t work), better than anyone
• Our current practice faces significant challenges due to growing complexity

Future:
• JPL is working with the FM Community to advance the state of the art and practice, to enable future classes of missions
  – Formalize theory, improve and standardize approaches and processes, develop tools (move from an “art” to a science)
  – Increase our collective ability to field safe and reliable systems
  – Enable formulation and development of more complex/capable systems
Opportunities to Continue the Discussion

• 2\textsuperscript{nd} NASA Fault Management Workshop (New Orleans, Louisiana; April/May 2012)
  – By invitation only
  – Contact Dr. Lorraine Fesq for more information: lorraine.m.fesq@jpl.nasa.gov

• Fault Management sessions at AIAA Infotech@Aerospace 2012 (Anaheim, California; June 18-21, 2012)
  – Call for Papers: www.aiaa.org/events/I@A
  – Abstracts due November 22, 2011
Fault Protection Context

Mission Timeline

Critical Activity/Launch

Time-Critical Activity/Deployments

Critical Activity/TCM

Critical Activity/Orbit Insertion/EDL

Orbital Operations

Surface Operations

Flight System

Flight Software Layer
- System Functions
- Application Specific (or Subsystem-specific) Functions
- Hardware Interface

Flight Hardware Layer
- Redundancy
- Cross-Strapping

FSW On-Board FP Autonomy

Flight System FDIR
- * FSW Layers
- * Hardware Layer

Ground System

Ground FDIR
- * Monitor/Trend
- * Diagnosis/Recovery
- * Contingency Plans / Procedures
- * Test-bed/Simulation
• Respond only to unacceptable conditions
• Avoid hair triggers and retriggering
• Tolerate false alarms
• Make parameters commandable
• Corroborate before severe responses
• Ensure commandability and long term safety
• Preserve consumables and critical data
• Log events and actions
• **Core Terms**
  - **Degradation**: The decreased performance of intended function.
  - **Anomaly**: The unexpected performance of intended function.
  - **Failure**: The unacceptable performance of intended function.
  - **Fault**: A physical or logical cause, which explains a failure.
  - **Root Cause**: In the chain of events leading to a failure, the first fault or environmental cause used to explain the existence of the failure.

• **System Terms**
  - **System**: A combination of interacting elements organized to achieve one or more stated purposes.
  - **State**: The value of a set of physical or logical state variables at a specified point in time.
  - **Behavior**: The temporal evolution of a state.
  - **Function**: The process that transforms an input state to an intended output state.
  - **Control Error**: The deviation between the estimated state and the ideal intended state.
  - **Nominal**: The state of the system when the output state vector matches the intentions of the designer and/or operator.
  - **Expectation**: The most likely predicted state or behavior.
Terminology Concept Diagram

- **Fault**: may be, from the perspective of a prior event in a causal chain, a root cause, which requires no further explanation.
- **Failure**: is explained by an anomaly.
- **Anomaly**: is the unexpected performance of intended behavior.
- **Error**: is difference from ideal.
- **Objectives**: describes intended behavior.
- **Function(s)**: describes expected behavior.
- **Model(s)**: used to develop lower-level controls.
- **State**: as a function of time is an incorrect system.
- **Environment**: is explained by a function(s).
- **System**: Has a set of model(s).

31
Progression of Anomalous/Failed States

Anomaly, no Failure
1) current value of state reaches an unexpected value
2) review of system data indicates that model/expectation is invalid, and state is expected (expectations changed) [e.g., noise in RF link due to un-modeled effect]
   • model reviewed and parameters adjusted until model predicts current behavior (e.g., if RWA unhealthy, will have larger attitude errors)
   • review of system data indicates that this is an unacceptable value (indicative of a failure; the goal is adjusted)

Anomaly, with Failure
a) current value of state unexpectedly reaches an unacceptable value
b) model reviewed and parameters adjusted until model predicts current behavior (e.g., if IMU1 unhealthy, will have attitude failure)
   • review of system data indicates that model/expectation is invalid, and state is acceptable (expectations changed)
   • recover intended functionality by restoring state to acceptable value and/or changing functional goal

Failure, no Anomaly
i. expected condition results in failure
ii. recover intended functionality by restoring state to acceptable value and/or changing functional goal
Simplified Fault Management Loop

1. **System State**
2. **Objectives**
3. **Detect**
4. **Diagnose**
5. **Decide**
6. **Adjust**
7. **Respond**

- **Recovery Actions**
- **Goal Changes**
- **Failure**
- **Anomaly**
- **Incorrect Expectation**
- **Unresolved Anomaly**
- **Changes to Model**

**Flow Diagram**:
- System State → Objectives → Detect → Diagnose → Decide → Adjust → Respond → System State
FM Functions

fault management

detection
- anomaly detection
- failure detection

prediction
- prognosis
- model adjustment

diagnosis
- fault isolation
- fault identification

decision
- fault response determination

response
- goal change
- failure recovery
- failure preclusion

Estimation Control
FM Completeness: Requires Top-Down and Bottom-up Analyses

Develop necessary FDIR

- list of local fault responses
- FME, FTA

Top-down assessment

- FDIR necessary to maintain acceptable functionality for each identified failure scenario
- for each failure scenario, assess acceptability (FDIR vs. FEPT)
- for each failure effect, assess relevant mission phases/activities; add identified hazards
- for each failure mode, identify failure effects

Bottom-up assessment

- FDIR necessary to maintain acceptable functionality through all mission phases
- FMEA, FTA

- determine fault set
- determine set of failure effects
- determine set of failure scenarios
- analyze set of failure scenarios

- determine system functions
- identify state(s) associated with each function
- determine acceptable ranges
- determine the acceptable values of each state for relevant mission phases/activities (goals); acceptable values may change over course of mission
- for each mission phase/activity, determine FDIR necessary to maintain acceptable function

- analyze set of success scenarios

- functional analysis, FTA, HA, IHA
System States

System states with identified Failure Modes (SF_M)

System states associated with objectives (SOBJ)

Set of system states

SF_M \setminus SOBJ \quad \text{set of "don't care" states w.r.t. FM design?}

SF_M \cap SOBJ \quad \text{FM approach can include detection of failure mode causes (TTC can be inferred from FMEA data)}

SOBJ \setminus SF_M \quad \text{FM approach (if not ignored) limited to detection of anomaly in state (since no causes identified)}

SOBJ \setminus SF_M \quad \text{set of states must be assessed for compliance with failure tolerance and reliability requirements}

SF_M \quad \text{set of states referenced in the set of failure effects. Includes time to effect data}
Relevant Representations and Relationships

- **Success Trees**
  - Represent system functions and functional decomposition
  - Conditions for success; "light" side

- **Fault Trees**
  - Represent system functions and paths to failure of top event
  - Conditions for failure; "dark" side

- **Directed graphs**
  - Represent components and connections/interfaces
  - Modeling of physical and logical connections enables formal modeling of failure effect propagation

- **Failure Modes and Effects Analyses (FMEA)**
  - Description of the failure modes (mechanisms) and the immediate failure effect
  - Modeled failure effect propagation enables formal and complete development of all failure effects

- **Event Sequences**
  - Describes system functionality as a function of time
  - Provides "triggers" to enable/disable elements of directed graph representation

- **State Machines** (Not Shown)
  - Necessary to assess sequencing of system states, both nominal and off-nominal
Redundancy and Cross-strapping Guidelines

Flight System Fault Tolerance Policy Development Considerations

No Fault Tolerance Policy

Considerations for Adding Selected Redundancy to a Single String Design

Fault Tolerance Policy

Redundancy Types and Considerations

Single String
Selective Redundancy
Block Redundancy
Functional Redundancy
Cooperative Redundancy

Cross-Strapping Considerations

Cross-Strapping

Exemptions
Exceptions
Fault Protection Components:  
Flight System FDIR / H/W Layer

• Key JPL design practice requires definition of **Fault Containment Regions** (FCRs):
  - “A fault containment region (FCR) is a segment of the system, the design of which is such that faults internal to the fault containment region do not propagate and cause irreversible damage beyond the limits of the fault containment region. **Note:** Fault propagation can be both direct/obvious (e.g. damage, disabling) and indirect/subtle (e.g. contention, interference).”

• **Fault containment boundaries in the flight equipment are always drawn around each of the following [8]:**
  1. any **redundant elements** (either functional or block redundant).
  2. any **non-critical functions or equipment** (e.g. any item where it's function is not required for mission success, such as engineering telemetry, instruments etc.).
  3. any **protective functions or equipment** that are conditionally needed, (e.g. OV/OC protect)
  4. any functional area or equipment the project requires to be fault tolerant.
  5. any functional area or equipment the projects requires fault containment for **development risk** (e.g. difficult to replace, long-lead, unique, or costly items are prime candidates for fault containment boundaries for development risk.)

• **FCRs are also important in Single String Designs**
## Fault Management Architectures

<table>
<thead>
<tr>
<th>Practitioner</th>
<th>Lockheed</th>
<th>Goddard</th>
<th>Orbital</th>
<th>APL</th>
<th>JPL</th>
<th>Ball Aerospace</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Family:</strong></td>
<td>&quot;Spider&quot; + VML</td>
<td>TMON</td>
<td>Rule-Based</td>
<td>Parallel State Machines</td>
<td>Smart Sequences</td>
<td>Local Software Logic</td>
</tr>
<tr>
<td><strong>Missions:</strong></td>
<td>MRO / Phoenix / MPL / MO</td>
<td>Dawn</td>
<td>GALEX</td>
<td>New Horizons / Messenger</td>
<td>Cassini AACS</td>
<td>Cassini CDS</td>
</tr>
</tbody>
</table>

| Deployment | Local | Central | Central | Central | Central | Central | Local | Central | Central | Central | Central |
| Thread Control | TBD | Parallel | Parallel | Parallel | Parallel | Parallel | Parallel | Serial | Serial | Serial |
| Interaction Management | State Checks | Enables /Disables in FP Seqs | Enables /Disables in FP Seqs | Enables /Disables in FP Seqs | State Checks | State Checks | TBD | Resource Contention Checks | None | Resource Contention Checks | TBD |
| Fault Response | Table-Defined Tiers | Table-Defined Tiers | Table-Defined Tiers | TBD | Macro Logic | State Machines withTiers | TBD | TBD | TBD | TBD | TBD |
| Behavior Selection | TBD | Monitor Persistence | Monitor Persistence | Monitor Persistence | TBD | Response Delay Logic | TBD | TBD | TBD | TBD | TBD |
| Behavior Pacing | Sequence | Sequenced | Sequenced | Sequenced | Sequence (Macros) | In-line | Sequenced | In-line | In-line | Sequenced | Sequenced |
| Primary Means of Command Execution | Via Sequence Syntax | N/A | N/A | Via Rule Syntax | Via Sequence Syntax | Via Response Code | Via Sequence Syntax | Via Response Code | Via Response Code | Via Response Code | TBD |
| Responsiveness to System State | | | | | | | | | | | |
Model-based programs interact with the system’s (hidden) state directly:

- Read state
- Set state

Model-based Executives automatically reason through interactions between states and sensors/actuators.

Programmers must reason through interactions between state and sensors/actuators.
Typical Spacecraft Execution Architecture

Command Sequence

Sequence Execution, Real-Time Behaviors, & Fault Protection

Observations

System Under Control

Commands
System Under Control

Command Sequence

Typical Spacecraft Execution Architecture

Sequence Execution

Fault Protection

Real Time Behaviors

Observations

System Under Control

Commands
Time-tagged sequences of low-level commands and “macros”

… executed by a nominal sequencing engine…

… with fault protection software running in parallel, ready to “take over” from nominal sequence execution when a fault monitor is triggered.

… augmented with event-driven behaviors when necessary…

System Under Control

Observations

Command Sequence

Real Time Behaviors

Fault Protection

Sequence Execution

Commands
Fault Protection is often considered an “add-on” capability, adjunct to the nominal control system and developed late in the project lifecycle, despite the fact that its design can uncover problems with the nominal control design.

Sequence designers’ intent is not explicit in the sequence.

System requirements and understanding of behavior are not always directly traceable to the flight software design.

The boundary between State Determination and State Control is sometimes blurred, with no explicit representation of “State” in the software.

Complex interactions between these elements make it difficult and costly to validate flight software, and to have confidence that it will work reliably and robustly.

Sequence

Fault Protection

Real Time Behaviors

Observations

Commands
Desirable Architectural Features

Control Specification

Onboard Executive

Fault-awareness (in-the-loop recoveries)

Models that are writable/inspectable by systems engineers

Separation of state determination from control, with an explicit notion of state at the boundary

Automated reasoning through low-level plant interactions

System Under Control

Observations

Commands

Simple state-based control specifications with explicit intent
Titan Model-based Executive

- Control layer has flexibility in achieving goal
- Enables integration of tiered fault management capabilities
- Enables integration of state-of-the-art autonomy software

Remote Agent Experiment on DS-1

Deep Space One

Goals
- Mission Manager
- HSTS: Planner/Scheduler
- Scripted Executive
- Livingstone: Diagnosis & Repair

Scripts

Planning models
Component models
Mission Data System Reference Architecture

- System operation via overt, objective statements of intent
- Clear delineation between control system and system under control
- State is explicit
- Models inform all aspects of control system
- Separation of estimation from control
Challenges and Opportunities

**Challenges:**
- Closing the mid- to high-TRL gap
- Must assure reliability ("bullet-proof" the implementation)
- Changes the operational paradigm – need new tools, training
- Cultural hurdles to acceptance of software technologies ("trust" issues)

**Opportunities:**
- Autonomy is an enabler for certain missions
- Evidence of significant cost savings in operations (EO-1)
- Model-based design lends itself well to development via MBSE methodologies
- Once general-purpose reasoners have been validated, V&V reduced to mission-specific models
- Amenability to formal V&V
• **UML Modeling**
  – Explicitly capture the intent of the requirements
  – Formally capture the behavior in a model
  – Create a crisp notion of **state**

• **State-based Framework**
  – Supports the UML standard
  – Allows developers to think and work with higher constructs – states, events and transitions

• **Auto-coding**
  – Light-weight Java program
  – Reads in the Model which is stored in a non-proprietary data format (XML)
  – Converts the input model into an internal data structure
  – Has multiple back-ends to support different project requirements

• **Test harness**
  – Ability to run the model stand-alone – module test environment

• **Model checking**
  – Automatic generation of Verification models
  – Exhaustively explore the state-space of the model
  – Checks for various correctness properties within the model
QSTATE Safing::Idle(QEvent const *e) {
   string stateName = objName + " Idle";
   switch (e->sig) {
      case Q_ENTRY_SIG:
         LogEvent::log(stateName + " ENTRY");
         return 0;
      case Q_EXIT_SIG:
         LogEvent::log(stateName + " EXIT");
         return 0;
      case ActivateSafingEv:
         LogEvent::log(stateName + " ActivateSafingEv");
         QF::publish(Q_NEW(QEvent, RequestSafingEv));
         Q_TRAN(&Safing::WaitForSC);
         return 0;
      case RunSafingCmdEv:
         LogEvent::log(stateName + " RunSafingCmdEv");
         Q_TRAN(&Safing::Active);
         return 0;
   }
   return (QSTATE)&Safing::Enabled;
}
STAARS Benefits

• **Lessened the gap between System and Software Engineering**
  – Formal specification of state behavior which can be implemented directly into flight software
  – Build rapid executable models for early prototype testing

• **Increased efficiency**
  – Software developers can greatly increase their output

• **Increased maintainability**
  – Rapid turn-around from specification changes to a software build

• **Increased reliability**
  – Fewer defects are introduced
  – Auto generated code based on a reliable statechart framework that conforms to the UML statechart semantics

• **Full control of the process**
  – Drawing tools can be swapped in and out
  – Autocoder can be customized for specific projects
    • Output in C or C++
    • Add more UML features – Deferred events, etc
    • Currently based on the Quantum Framework’s Publish/Subscribe – but could be customized to be based on other Frameworks