JPL Fault Protection Experiences – Case Studies

Response to:
(1) Growth in Flight Software (FSW) Complexity" V5, NASA OCE
(2) Reducing Complexity in Flight Software, July 11, 2007, JPL

Kevin Barltrop, Jet Propulsion Laboratory, California Institute of Technology
Dan Dvorak, Jet Propulsion Laboratory, California Institute of Technology

.

Case Studies

We'll now look at some case studies exploring:

1. Defining the scope for the fault protection.
2. Matching the requirements and the software solution.
3. Generating software solutions.

Case #1: Context

- The mission is a “Discovery” class mission to excavate material from a comet using the impact of a daughter spacecraft while a parent spacecraft collects up-close images and spectral data of the impact crater growth.

Case #1: Essential Complexity

What essential complexity is established by the key mission requirements and goals?

- Operate for two weeks unattended time without ground-in-the-loop. (actual time was expanded to six-months during extended mission)
- Fail operational during the 24-hour Encounter activity that includes two spacecraft operating in tandem with several fundamentally different operating modes each:

  Parent spacecraft :
  1. Release the daughter spacecraft
  2. Execute a pre-planned large divert maneuver.
  3. Provide real-time communications link for daughter spacecraft.
  4. Perform several minutes of mission critical imaging and relay subset in real-time to ground.
  5. Pass near to comet using shield mode to reduce probability of catastrophic damage.
  6. Perform look-back imaging and play back recorded data.

  Daughter spacecraft:
  1. Separate from Flyby spacecraft and fly on its own for first time.
  2. Acquire comet image and estimate relative motion.
  3. Execute three autonomously computed maneuvers during the last few hours before impact.
  4. Intercept the surface of the comet
- Accommodate unexpected disturbance torques due to comet hill-sphere.
- Comply with single point failure policy by supporting configuration management of two strings of hardware with several cross-strapping regions defined to reduce propagation of faults across subsystems during encounter.
- Allow tuning of recovery priorities at encounter to match relative importance of ongoing activities.
- Support autonomous navigation function and fall-back to degraded operation modes.
- Minimize amount of fault protection commanding during mission. (Goal)

Case #1: Incidental Complexity Systems Engineering Analysis

How did systems engineering analysis affect the growth of incidental complexity?

- Some incidental complexity began with systems engineering decisions early in the project. A number of proposed descopes were rejected:
  - Remove fail operational capability during encounter
  - Eliminate autonomous use of cross-strapping
  - Remove use of targeted hardware responses in addition to blanket system responses
  - Remove autonomous responses of any kind during encounter
  - Don't make software smart enough to inhibit fault monitor at operationally appropriate times.
  - Don't apply belt and suspenders disabling of algorithms.
- Fault tree and failure modes analysis was applied to identify which conditions would be handled by on-board fault protection.
- A home-grown hierarchy based response architecture model was used to generate scenarios that needed to be covered within the flight software architecture. This guided the organizational structure of the responses.

Case #1: Incidental Complexity Software Architecture

How did software architecture affect the growth of incidental complexity?

- The software architecture provided some good provisions to control growth:
  - Design Summary:
    - An inherited architecture for detecting, diagnosing and responding to faults [Matured design descended from Pathfinder and Deep Space 1]
    - A sequence engine shared with nominal activity sequence execution.
    - A common data bus for state information and event notification shared with other flight software tasks.

  Overall -- This provided the typical benefits of a good architecture in many areas of the design.

- However, we had some mismatches between algorithm design and architecture services:
    - The initial design to produce a well-behaved system was modeled using a fault tree hierarchy and time lines for recovery scenarios, but the architecture provided only a flat structure with no concept of recovery time lines, preventing global enforcement of layering and pacing of responses. The hierarchy of recovery and the pacing to allow for completion of recovery attempts was repeatedly represented in the implementation of the individual responses rather than supported in a system model by the architecture.
    - Lack of capability to examine symptom message payloads (such as identify of hardware string in fault) in response algorithms led to proliferation of message IDs representing germane payload combinations, many of which mapped to separate response instantiations with their own parameters. Would have been better to make the payload accessible and then have the response designs individually determine whether it was necessary to key off of that payload.
    - An FP sequence interlock to avoid re-entry was coded within each sequence rather than built into the sequence engine, making FP configuration sequences too risky to use for nominal activities, despite their likely utility.
    - Much of the flight software was driven by tables with selectable entries that lacked identification with abstract modes of the system. This make it difficult to coordinate the “implied” modes of the table entries with the system behavior. For example, one table included multiple thruster-based control configurations and wheel-based control configurations with no flag indicating to which of those two classes the configuration belonged. This made it difficult to adjust performance monitoring according to the system control mode.

  Overall – Because the design was optimized to use functions not supported in the architecture that was available, we introduced some avoidable complexity.

Case #1: Incidental Complexity Tools

How did tools affect the growth of incidental complexity?

- Visualization and communication
  - The good
    - State machine representations were used for responses and allowed the designers to understand the behavior.
    - Flow chart representations were used for monitors.
    - Mapping tables clearly showed progression from symptoms to responses
  - The bad
    - Because state machines were used "as is" for autocoding, the level of detail was too much for most review audiences. Should have had a filter to present more abstract versions of the diagrams.
    - The flow chart representations used a non-standard convention that required frequent explanation.
    - The proliferation of IDs due to the architecture led to large tables and we did not do a good job of distinguishing between cases where these IDs represented used payload combinations versus un-used.
    - We lacked tools to illustrate the behavior of critical sequencing roll-back and roll-forward. As a result there were late changes in timing for this area and at least one design breakage (fortunately benign) that made it to the final activity.

Overall – We did only a fair job of presenting the design in a way that facilitated understanding.

Case #1: Incidental Complexity Tools

How did tools affect the growth of incidental complexity (cont'd)?

- State machine autocoding tool
  - The good
    - We were able to unit test much of the algorithm behavior in the state machine design environment (Matlab Stateflow) before even going to code.
    - Software defects in the autocoded algorithms were *negligible*. The only example of something resembling a defect was when an identical structural change was made to a dozen state machine diagrams, a copy and paste error was introduced into one.
    - The command and telemetry database was always perfectly consistent with the documented design and the generated code.
  - The bad
    - As "source code" for the software, the state machine diagram files were de facto software artifacts, but we overlooked the cost and effort to CM them like software.
    - The manually written code to unpack spacecraft telemetry into the state data accessible by the response algorithms had a high defect rate. Would have been better to have integrated data infrastructure for both the fault protection code and the rest of the flight code.

Overall – We did a good job of generating code with less additional complexity for the specified system design compared to past missions.

Case #1: Incidental Complexity Tools

How did tools affect the growth of incidental complexity (cont'd)?

- Verification and Validation
  - The good
    - Perl script models of the initial fault hierarchy and recovery time-line created the initial confirmation of system behavior.
    - Stateflow models of the monitors and responses allowed comprehensive testing of the paths and states.
  - The bad
    - We attempted to apply model checking with Promela and Spin, but got so bogged down in formulation of the "correctness properties" that we really didn't get useful results.

Overall – We did a good job of exploiting early design validation techniques to flush out key features.

Case #1: Complexity Consequences

What where some key consequences of the complexity?

- Confusion during design reviews.
  - Contractor management frequently complained that they didn’t understand what was being presented.
  - Too much time was spent explaining how to interpret state machine diagrams and the unconventional monitor flow charts.
  - There was frequent disagreement over the autonomy scope for the mission.
- Unclear and delayed verification and validation.
  - A key system recovery capability for Encounter was not completed and tested until after launch.
  - A key undercharge scenario repeated failed in simulation, but passed on the flight hardware.
  - At least five key simulation models were incorrect, leading to inaccurate test results in those areas. Most were discovered during operations anomalies.
- Frustration and cascade effects during code maintenance.
  - A small late change in an ACS algorithm setting “broke” the launch sequence less than two months prior to launch.
  - The Encounter sequence required numerous small fault protection tweaks, some quite late in the game.
  - Encounter sequence triggered an unexpected, but benign, alarm due to a small late command timing shift whose effect wasn’t modeled in the simulations.
- Gaps in testing due to schedule crunches.
  - Some tests were combined, but original test scope largely intact.

Case #2: Context

- Mission is a “Discovery” class mission to sequentially rendezvous with and orbit two asteroids.

Case #2: Essential Complexity

What essential complexity is established by the key mission requirements?

- Operate for two weeks unattended time without ground-in-the-loop.
- Fail safe during the mission.
- Comply with single point failure policy by supporting configuration management of two strings of hardware with several cross-strapping regions defined to reduce propagation of faults across subsystems during encounter.
- Minimize amount of fault protection commanding during mission. (Goal)
- Support protection of difficult power subsystem design needed for the ion propulsion system.

Case #2: Incidental Complexity Systems Engineering Analysis

How did systems engineering analysis affect the growth of incidental complexity?

- Performed fault tree and failure modes analysis to identify coverage.
- Approach was to augment legacy earth-orbit design with additional capabilities needed to satisfy mission.
- Had few if any system modeling or global behavioral strategies to improve implementation behavior.

Case #2: Incidental Complexity Software Architecture

How did software architecture affect the growth of incidental complexity?

- The software architecture consisted of a telemetry monitoring task that triggered simple fault protection sequences via a custom fault protection sequence engine.
- Monitors were used to trigger behaviors for off-nominal events as well as nominal events such as launch initialization.
- The architecture required some work-arounds to support deep space operations instead of the legacy earth-orbit ops:
  - Interactions were managed within the sequences through mutual enabling and disabling of other sequences and fault monitors
  - Sequence chains were used to compensate for size limitations on individual sequences.
  - The standard monitoring was augmented through “derived functions” that incidentally obscured visibility into what quantities were actually being checked
- Required exhaustive test case exploration, fixing particularly bad interactions as they were discovered.
- Small changes in any of the fault protection sequences led to high risk of introducing new interactions due to shifts in time lines.
- Accommodating the increase scope of fault coverage with this architecture allowed the system to over-respond to multiple symptoms resulting from an underlying fault.

Overall – The architecture was probably “too simple” for the specified application, which forced the development of work-arounds that circumvented the architecture and to strive for good overall system behavior via as-needed solutions for logic problems.

Case #2: Incidental Complexity Tools

How did tools affect the growth of incidental complexity (cont'd)?

- Visualization and Communication
  - The design was presented in terms of its simple look-up table architecture, but the actual behavior resulting from the interaction of the components was not describable with any of the tools available.

Overall – Visualization was oversimplified, which curtailed discussions, but underrepresented the complexity of the actual behavior.

- Verification and Validation
  - Software-only simulations of the flight system were used to perform brute force testing via thousands of fault scenario test runs. This contributed greatly to flushing out some bad design interactions.
  - However, there was insufficient time to even begin multi-fault stress cases prior to launch, as is often done to test out the limits of a system.

Overall – Lack of analysis tools on top of a shallow architecture and large implementation complexity led to a dangerously low level of V&V.

Case #2: Complexity Consequences

What were some key consequences of the complexity?

- Confusion during design reviews.
  - Reviews were not perceived to be especially confusing, but it turns out that the real behavior of the system was not really discussed.
- Unclear and delayed verification and validation.
  - At least one critical problem was exhibited in a discretionary test case and not found in any of the incompressible cases.
  - Testing of core launch scenarios completed very late.
  - General sense that a body of undiscovered interaction cases remained.
- Frustration and cascade effects during code maintenance.
  - A couple of fault protection sequence changes before launch produced side effect breakage that wasn't detected until after delivery.
  - The fault protection team routinely argued against even minor externally directed changes due to cascade concerns.
- Gaps in testing due to schedule crunches.
  - Significant test descopes were required during the six months before launch.
  - Many of the ion thrusting test cases were not completed before start of the ion thrust mission phase.
  - Little or no intentional multi-fault stress cases to probe the limits of the system.

Case #3: Context

- The project was to develop a prototype control system for a next-generation Deep Space Network consisting of a large array of small antennas. The project demonstrates:
  - A control architecture that shapes both systems engineering and software engineering
  - Requirements specified in a structured form rather than free-form English “shall” statements
  - Fault protection that is integrated with and uses the same control mechanisms as nominal operation
  - A reusable software framework that turns most development into adaptations of architectural components

Case #3: Essential Complexity

What essential complexity is established by the key project requirements?

- Automate all routine activities (antenna setup, tracking, tear-down)
- Coordinate multiple elements (antennas, front-end electronics, signal processing) and multiple concurrent tracking activities
- Allocate/reallocate assets according to policies (especially in oversubscribed situations)
- Perform on-the-fly replacement of a failed antenna during a tracking activity

Case #3: Incidental Complexity Systems Engineering Analysis

How did systems engineering analysis affect the growth of incidental complexity?

- Used State Analysis to:
  - Make clear distinction between control system and system under control
  - Understand the ‘physics’ of the system under control via a “state effects diagram” and associated models
- State Effects Diagram and corresponding models proved very valuable in communication between systems and software engineers
- Continually made distinctions between framework functionality and adaptation
- Lack of “projection” capability induced some complexity/workaround in adaptation

Case #3: Incidental Complexity Software Architecture

How did software architecture affect the growth of incidental complexity?

- The state/model/goal-based architecture shaped both the software architecture and the representation of requirements, making the translation straightforward
- Software architecture made some key distinctions:
  - Measurements are not the same as state estimates
  - Estimation should be separated from control
  - State timelines distinguish between operational intent and state knowledge
- Had to always fight the urge for a “software solution” that fixed the immediate symptom but not the real cause

Case #3: Incidental Complexity Tools

How did tools affect the growth of incidental complexity?

- Lack of a tool to check for architectural compliance leaves as-built system vulnerable to hacks
- Want a tool to visually compare simulated state, estimated state, and desired state
- A Wiki, structured according to architectural elements, proved useful for keeping alignment between systems engineering analysis and software

Case #3: Complexity Consequences

What were some key consequences of the complexity?

- Happily, small changes in requirements had small localized effects on software
- Occasional confusion between intent and knowledge was easy to identify
- Control architecture helped bridge the gap between systems and software engineers … but both need training in this new approach
- Had to devote more resources to verification of framework software (but that’s a good investment because it’s reusable software)
- Operations plans easy to review with domain experts because it follows the state effects diagram