Final Report

NASA Study on Flight Software Complexity

Commissioned by the NASA Office of Chief Engineer
Technical Excellence Program
Adam West, Program Manager

"The demand for complex hardware/software systems has increased more rapidly than the ability to design, implement, test, and maintain them. … It is the integrating potential of software that has allowed designers to contemplate more ambitious systems encompassing a broader and more multidisciplinary scope, and it is the growth in utilization of software components that is largely responsible for the high overall complexity of many system designs."

Michael Lyu
Handbook of Software Reliability Engineering, 1996

"While technology can change quickly, getting your people to change takes a great deal longer. That is why the people-intensive job of developing software has had essentially the same problems for over 40 years. It is also why, unless you do something, the situation won't improve by itself. In fact, current trends suggest that your future products will use more software and be more complex than those of today. This means that more of your people will work on software and that their work will be harder to track and more difficult to manage. Unless you make some changes in the way your software work is done, your current problems will likely get much worse."

Watts Humphrey
Winning with Software: An Executive Strategy, 2001

Editor:

Daniel L. Dvorak
Systems and Software Division
Jet Propulsion Laboratory
California Institute of Technology

Contents

Executive Summary

In 2007 the NASA Office of Chief Engineer (OCE) commissioned a multi-center study to bring forth technical and managerial strategies to address risks associated with the growth in size and complexity of flight software (FSW) in NASA's space missions. The motivation for the study grew from problems attributed to flight software in a variety of missions—in both pre-launch and post-launch activities—and concerns that such problems were growing with the expanding role of flight software. The study was tasked to examine the growth in flight software size and complexity, recommend ways to reduce and better manage complexity, and identify methods of testing complex logic. The study gave special attention to fault protection software because of its complexity. Study participants consisted of engineers and managers at Applied Physics Laboratory, Goddard Space Flight Center, Jet Propulsion Laboratory, Johnson Space Center, and Marshall Space Flight Center.

The study adopted a simple definition of "complexity": how hard something is to understand or verify. This definition implies that the main consequence of complexity is *risk*, whether technical risk or schedule risk or mission risk. It also highlights the role of humans in the equation since understandability can be enhanced through education and training, as reflected in some recommendations. The study examined complexity not just in flight software development, but in upstream activities (requirements and design) and downstream activities (testing and operations). The study made a distinction between *essential functionality*, which comes from vetted requirements and is, therefore, unavoidable, and *incidental complexity*, which arises from decisions about architecture, design, and implementation. The latter can be reduced by making wise decisions and is, therefore, the target of most recommendations.

Flight software is a kind of embedded real-time software, a field that has seen exponential growth since its inception. In some areas of NASA, flight software is growing by a factor of ten every ten years. Estimates for Orion's primary flight software exceed one million lines of code. The growth trend is expected to continue because of increasingly ambitious requirements and because of the advantages of situating new functionality in software or firmware rather than hardware. Flight software has become a spacecraft's "complexity sponge" because it readily accommodates evolving understanding, making it an enabler of progress. NASA is not alone in such growth. Over the forty years from 1960 to 2000, the amount of functionality provided by software to pilots of military aircraft has grown from 8% to 80%, and software size has grown from 1000 lines of code in the F-4A to 1.7 M lines of code in the F-22. The newest fighter still under development, the F-35 Joint Strike Fighter, will, according to one source, have 5.7 M lines of code.

This study examined the chain of engineering activities related to flight software, beginning with requirements development and continuing through analysis and design, software development, verification and validation, and operations. Findings in each of these areas led to more than a dozen recommendations in the areas of systems engineering, software architecture, testing, and project management, as summarized below.

- Engineers and scientists often don't realize the downstream complexity (and cost) entailed by their local decisions. Overly stringent requirements and simplistic hardware interfaces can complicate software; flight software descope decisions and ill-conceived autonomy can

complicate operations; and a lack of consideration for testability can complicate verification efforts. We recommend educational materials, such as a "complexity primer," and the addition of "complexity lessons" to *NASA Lessons Learned*.

- Unsubstantiated requirements have caused unnecessary complexity in flight software, either because the requirement was unnecessary or overly stringent. Rationale statements have often been omitted or misused in spite of best practices that call for a rationale for every requirement. The NASA Systems Engineering Handbook states that rationale is important, and it provides guidance on how to write a good rationale and check it. In situations where well-substantiated requirements entail significant software effort, software managers should proactively inform the project of the impact. In some cases this might stimulate new discussion to relax hard-to-achieve requirements.
- Engineering trade studies involving multiple stakeholders (flight, ground, hardware, software, testing, operations) can reduce overall complexity, but we found that trade studies were often not done or done superficially. Whether due to schedule pressure or unclear ownership, the result is lost opportunities to reduce complexity. Project managers need to understand the value of multi-disciplinary trade studies in reducing downstream complexity, and project engineers should raise complexity concerns as they become apparent.
- Good software architecture is the most important defense against incidental complexity in software designs, but good architecting skills are not common. From this observation we made three recommendations: (1) allocate a larger percentage of project funds to up-front architectural analysis in order to save in downstream efforts; (2) create a professional architecture review board to provide constructive early feedback to projects; and (3) increase the ranks of software architects and put them in positions of authority. An insightful examination of complexity in flight software, included as Appendix H, discloses numerous examples of confused thinking that increase complexity in flight software.
- Decisions *not* to implement some functions and *not* to fix some defects in flight software are often made on the basis of whether or not an "operational workaround" exists. This amounts to moving complexity from flight software to mission operations where it is a continuing cost rather than a one-time cost and where it increases the risk of operational error, especially as the numbers of such workarounds accumulate. Operators be consulted early and often in flight/ground trade studies and in flight software descope decisions, and the costs and risks of operational workarounds should be *quantified* to support better decision-making.
- Fault management software is among the most difficult to specify, design, and test; our findings led to three recommendations: (1) inconsistency in fault management terminology among NASA centers and their contractors suggest the need for a NASA Fault Management Handbook; (2) mismatches between mission requirements and fault management approaches suggest the need for fault management reviews in proposals; and (3) lack of attention to fault management in university curricula suggest that NASA should sponsor or facilitate the addition of a fault management course within a university program. Also, the traditional separation of fault management logic from nominal control logic is at odds with control theory and should be corrected, as explained in Appendix G.
- Commercial off-the-shelf (COTS) software is a mixed blessing. Although it can provide valuable functionality, it often comes with unneeded features that complicate testing (due to feature interactions). Before deciding to use a COTS product, it's important to analyze it for

separability of its components and features and weigh the cost of testing unwanted features against the cost of implementing only the desired features.

- An exceptionally good software development process can keep defects down to as low as 1 defect per 10,000 lines of code. This means that a system containing 1 million lines of code will have 100 defects, some of which will manifest during mission operations. Three strategies to deal with this reality are: (1) apply randomized testing to avoid bias toward specific sources of error; (2) extend fault protection to cover software faults, not just hardware faults; and (3) employ fault containment techniques to reduce the number of ways in which residual defects can combine to produce complex error scenarios. Note that these are protective measures; they do not reduce software complexity.

Some of the recommendations summarized above are common sense but aren't common practice. We identified four cultural reasons for the current state. First, cost and schedule pressure lead managers and developers to reduce or eliminate activities other than the production of code. Improvements that require time and training are vulnerable, particularly if their benefits are hard to quantify. Second, there is a lack of enforcement; some recommendations already exist in NASA documents and/or local practices, but aren't followed, possibly because nobody checks (and because of schedule pressure). Third, there is strong pressure to re-use software from a previous mission because it is "flight-proven" and presumed to be of lower risk. That reasoning sometimes has merit, but can inhibit better approaches, particularly when legacy software is partly to blame for difficulties on previous missions. Finally, there is no programmatic incentive for project managers to "wear the big hat" and contribute to infrastructure and process improvement for the benefit of future missions. This *can* be changed, as evidenced by many companies that have transitioned from stovepipe product development to product-line practices.

On a historical note, software was a subject of great concern forty years ago. In 1968 NATO held the first conference on software engineering to address the "software crisis." More than fifty experts from eleven countries, all concerned professionally with software, either as users, manufacturers, or teachers at universities, attended the conference. The discussions examined problems that are still familiar today: inadequate reliability, schedule overruns, unmet requirements, and education of software engineers. One position paper described the situation as a genuine crisis, citing "a widening gap between ambitions and achievements in software engineering," as evidenced by poor performance of software systems and poor cost estimation. The author expressed alarm concerning "the seemingly unavoidable fallibility of large software, since a malfunction in an advanced hardware-software system can be a matter of life and death." Advances since 1968 have enabled software systems three orders of magnitude larger while software cost as a fraction of total budget has remained small. What was once considered complex became routine because we learned better ways; however, our ambitions have also grown, creating new challenges that today seem complex. This report examines those sources of complexity and offers technical and managerial strategies to address them.

1 Introduction

1.1 Origins and Objectives

In 2007 the NASA Office of Chief Engineer (OCE) commissioned a multi-center study of the growth in flight software size and complexity in NASA space missions. The study was motivated by problems attributed to flight software in a variety of missions—in pre-launch activities and mission operations—and concerns that such problems were growing with the expanding role of flight software. While advances in computer hardware have readily accommodated this expansion by providing faster processors and increased memory, it is believed that software engineering and management approaches have not kept pace. Problems with flight software have serious technical, operations, and management implications for space flight missions.

The study was chartered to bring forth technical and managerial strategies to address risks associated with the growth in size and complexity of flight software in NASA's space missions. The study addresses many motivating questions. For example:

- How big is flight software and how fast is it growing?
- What are the causes of growth?
- Is there unnecessary growth and how can it be curtailed?
- Why is fault protection software so difficult?
- When growth is necessary, what are effective strategies for dealing with it?
- How can we ensure adequate testing of complex flight systems?

These and other questions are addressed in this report.

1.2 Work Description

The study was directed to examine four areas of special interest, as described below, in the directions given to the study team.

- *Special Interest 1*:

  Provide a clear exposé of the growth in NASA FSW size and complexity. Describe long-term trends, how the trade space has changed, root causes, problems encountered, benefits gained, and lessons learned.

  - *Approach*. Beginning with the analysis JPL has already performed that characterizes the growth in size and complexity of JPL FSW, extend that analysis to APL, GSFC, JSC, and MSFC and, thereby, provide a comprehensive assessment that encompasses all NASA centers that produce FSW. An important subtask of this activity will be to characterize and define "complexity" in the NASA FSW context by leveraging work performed by JPL engineers on the Common Avionics Study Team activity within the Constellation program.

- *Special Interest 2*:

  How can unnecessary growth in complexity be curtailed? How can necessary growth in complexity be better engineered and managed? Are there effective strategies to make smart trades in selecting and rejecting requirements for inclusion in flight software to keep systems lean? When growth in complexity and software size are necessary, what are the successful strategies for effectively dealing with it?

  - *Approach 1*: Architectures that are specifically designed to enable flight/ground capability trades are critical to moderating unnecessary growth in FSW complexity. (The Mission Data System developed at JPL is an example of such an architecture.) Characterize the interaction between avionics design choices and the impact they have on FSW complexity. Explore this and other architectural approaches as a means to manage FSW complexity.
  - *Approach 2*: Conduct a survey of strategies proposed in the literature to manage the complexity of embedded real-time systems.
  - *Approach 3*: Develop a position paper that provides an "outside-the-box" or controversial approach to complexity management.

- *Special Interest 3*:

  Fault protection logic accounts for a sizable portion of flight system software. Are there techniques that effectively manage the complexity of fault protection systems?

  - *Approach 1*: Fault protection tends to require coordination features, attention to limited resources, goal-driven behaviors, observability requirements, and more that should originate as part of the core architectural concepts. Unfortunately, designs typically address "nominal" capability in an oversimplified way, with fault protection added on top without the benefit of architectural guidance. This self-induced complexity results in a brittle design with numerous add-on patches to make things work, interminable testing, and inoperable designs. Accordingly, it's not necessarily true that fault protection need be overly complicated. There's a danger here that in the zeal to banish this scapegoat the fundamental issues remain unresolved. Leverage the experience JPL has gained in designing fault protection systems for its robotic missions and shared them in a workshop specifically focused on fault protection. Contribute to the NASA Planetary Spacecraft Fault Management Workshop (April 2008) commissioned by the Planetary Sciences Division of the Science Mission Directorate. Provide specific recommendation for techniques that bound the fault protection capabilities required for mission success.
  - *Approach 2*: Investigate and document approaches to fault protection used to date within NASA. What worked and what didn't? Extract this information from the NASA Fault Management Workshop.
  - *Approach 3*: Investigate the feasibility of eliminating or at least minimizing fault protection software as a separate entity. What would it take to achieve a similar effect by deeply integrating fault protection into the "nominal" system? Is there really a need for an independent, global fault protection system?

- *Special Interest 4*:

  Investigate the challenge of testing complex logic for appropriate safety and fault protection provisions. Recent mission delays and the loss of a key spacecraft have been attributed to inadequate testing of complex FSW systems. How can this situation be prevented in the future?

  - *Approach*. Apply the expertise of the JPL Laboratory for Reliable Software (LaRS) and its principal scientist Gerard Holzmann. Techniques developed and utilized by LaRS include model-based verification techniques such as SPIN, theorem checkers, and advanced randomized testing techniques used to efficiently verify the correctness of the MSL flash file system.

This study examined sources of complexity not only in the development of flight software but in engineering activities upstream and downstream of software production, as shown in Figure 1. Requirements are a potential source of unnecessary complexity when they are more demanding than necessary. Analysis and design (including software architecture) can introduce unnecessary complexity when they fail to deal elegantly with all the functional and non-functional requirements. Software implementation can introduce unnecessary complexity in the form of language features that are hard to understand or analyze. Testing can be unnecessarily complex when testability is not properly considered in the software architecture. Finally, operations is left "holding the bag" for all the complexities that have not been handled earlier. Although some kinds of operational complexity are unavoidable, other kinds are a direct result of decisions made earlier.

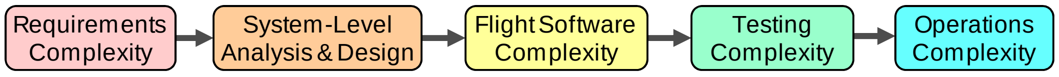


Figure 1. The scope of this study on flight software complexity included both upstream and downstream activities in the engineering lifecycle.

1.3 Organization of Report

This report is designed to provide results in the form of progressive disclosure, with four parts arranged in order of increasing detail. First, the Executive Summary presents the recommendations in a brief form. Next, Section 2 presents the findings and recommendations with more substantiation for each recommendation. Sections 3 through 8 then report on all the study activities, providing much of the raw data that shaped the recommendations. Finally, the appendices provide detailed analysis and information from specific areas of study. Although some appendices are long, they provide considerable information and insight for readers who wish to gain an in-depth understanding of the issues.

In the main body, Section 3 details the growth in size of NASA flight software, with an examination of the causes. Section 4 examines software complexity and its causes, deliberately separate from Section 3 to avoid confusing 'size' with 'complexity'. Appendix B reports on results from meetings at each of the centers regarding architectures, trades, and avionics impacts. Section 5 and the related Appendices F and G present the findings related to fault protection. Section 6 summarizes the position paper "Thinking Outside the Box," provided fully in Appendix H. Section 7 and related Appendices D and E present results related to verification and validation of complex systems. Section 8 and Appendix A discuss two literature surveys: one of open literature about software complexity in industry and one of NASA documents that have addressed software complexity.

A study of this type inevitably identifies some additional topics that, if properly studied, could form the basis of important new recommendations. Section 9, therefore, lists several intriguing topics that were not included in this study.

1.4 Participants

The study was designed to draw on expertise at five institutions responsible for space flight software: Applied Physics Laboratory, Goddard Space Flight Center, Jet Propulsion Laboratory, Johnson Space Center, and Marshall Space Flight Center. Participants at each institution are listed below.

1.4.1 Applied Physics Laboratory Participants

- George Cancro has over 10 years of experience as a Systems Engineer. He was the Fault Protection lead engineer on the STEREO spacecraft.
- Brian Bauer has two years experience as a System Engineer. He is the current Fault Protection lead engineer on the New Horizons mission.
- Adrian Hill has over 20 years experience as a software engineer. He was the flight software lead engineer on MESSENGER, and is currently the Fault Protection lead on MESSENGER. He also served as the Fault Protection lead on New Horizons until after launch.
- Debbie Clancy has over 20 years experience as a software engineer. She was the Fault Protection Processor lead software engineer on MESSENGER. She was the Mission Software System Engineer on New Horizons, and is filling that same role on RBSP.
- Bob Davis has over 20 years experience as a software engineer and as an acceptance tester. He developed the spacecraft emulator software on the STEREO mission, and developed flight software for New Horizons.
- Mark Reid has over 15 years experience as a software engineer. He was on the Fault Protection development and test team on New Horizons. He is currently the Flight Software lead engineer on RBSP.
- Horace Malcom has over 30 years experience as a software engineer. He has written flight software for numerous spacecraft and instruments. He wrote software for the PEPSSI instrument on New Horizons, and developed boot code software on New Horizons.
- Roland Lang has over 22 years experience as a software engineer. His area of expertise is in performing formal acceptance testing of ground and flight software. He served as the Acceptance Test lead on the MESSENGER and New Horizons missions.
- Jacob Firer has over 20 years experience as a software engineer. He was a flight software developer on MESSENGER and New Horizons, and is performing the same role on RBSP.

- Chris Krupiarz has over 15 years of experience as a software engineer. He was a flight software developer on MESSENGER, and is currently the flight software lead on MESSENGER.
- Dan Wilson has over 30 years experience as a software engineer. He has particular expertise in the areas of Guidance and Control systems and software. He was the Guidance and Control software lead on the TIMED and STEREO missions, and is currently the STEREO Mission Software System Engineer.
- Aseem Raval has over 10 years experience as a software engineer. He developed flight software on the STEREO and New Horizons missions, and is performing the same role on RBSP.
- David Artis has over 25 years experience as a software and system's engineer. He was the Mission Software System's Engineer on MESSENGER, and is now the Data System Engineer on RBSP.
- Steve Williams has over 25 years experience as a hardware and software engineer. He was the C&DH software lead on the TIMED mission, and is the flight software lead on New Horizons. Steve was the primary contact at APL for this study, and he also edited the "Complexity Primer" attached as Appendix C.

1.4.2 Goddard Space Flight Center Participants

- Harold "Lou" Hallock is a software systems engineer with extensive experience in flight and ground systems for robotic/scientific missions, and was the primary contact at GSFC for this study. Lou has been supporting the LISA mission and authored Section 4.4 on the LISA case study.
- Kequan Luu served as the Attitude Control System Software Lead of the SAMPEX and TRMM missions, Mission Operations System Engineer of SHOOT, Associate Branch Head of the GSFC Flight Software Branch, and currently serves as the Principal Engineer of the Flight Software System Branch.
- Manuel Maldonado currently serves as the Solar Dynamics Observatory Software Systems Engineer. Before that he served as the Product Development Lead for the team that developed the flight software for ICESat mission GLAS Instrument.
- Jane Marquart is a software systems engineer with extensive experience in flight software systems for robotic/scientific missions. Most recently she has been working on new technologies for flight systems and is the Deputy Data Standards Manager at GSFC.
- David McComas has over 20 years experience working for the GSFC Flight Software Branch. His experience as a FSW developer and tester spans spacecraft bus and instrument FSW including both C&DH and GN&C subsystems. He is currently the FSW Product Development Lead for the Global Precipitation Measurement spacecraft.
- Jonathan Wilmot serves as a software systems engineer and architect for several GSFC technology initiatives. He has extensive experience in full life-cycle software developments for military aviation and spacecraft avionics in both technical and managerial roles.

1.4.3 Jet Propulsion Laboratory Participants

- Kevin Barltrop is a senior member of the Autonomy and Fault Protection Group at JPL, has worked on a variety of flight projects spanning formulation phase through extended mission, and has worked a variety of research projects. He is currently the GRAIL Lead Flight Systems Engineer as well as the principle investigator for an automated testing research activity. Kevin led the study activities on fault protection and authored the whitepaper in Appendix F.
- Brad Burt is a Systems Engineer with extensive experience in Flight System Fault Protection Systems, Operations, and I&T. Most recently, he has been working on documenting JPL's Fault Protection heritage, as well as working on various activities directed toward improving and extending current best practices in this area.
- Len Day is a Senior Software Engineer in the Flight Software Operating Systems and Avionics Interfaces Group. He specializes in embedded real-time systems and has participated in many space missions going back to Viking and as recently as the Orbiting Carbon Observatory scheduled for launch in January 2008.
- Daniel Dvorak is a Principal Engineer in the Planning & Execution Systems Section, currently working on technology for goal-driven mission operations, including extensions to support human-robotic interactions. He has also worked extensively on a model-based systems engineering methodology and a state- and model-based architecture for control systems. Dan was the lead for this study and the editor of this report.
- Lorraine Fesq is a Principal Engineer within the Engineering Development Office at JPL. She has over 30 years of aerospace experience that spans industry, government, and academia, and has worked all mission phases of spacecraft development including technology research, requirements definition, systems design, hardware/software integration and test, launch and mission operations. Lorraine was Technical Coordinator of NASA's Planetary Spacecraft Fault Management Workshop that brought together one hundred FM practitioners and experts, providing a major source of input to this study.
- Gerard Holzmann is a JPL Fellow who leads the Laboratory for Reliable Software. He is the originator of the SPIN system for model-checking of software. Gerard led the study activities of Special Interest 4: testing of complex flight software.
- Cin-Young Lee is currently a FSW engineer on MSL and Co-I for an R&D project on automatic test case generation using genetic algorithms. He researched and authored the survey of external literature, attached as Appendix A.
- Robert Rasmussen is a JPL Fellow, Chief Engineer in the Systems and Software Division, and formerly the cognizant engineer for Cassini's attitude and articulation control system. He has many years of experience in fault protection and was the original architect of the Mission Data System. Bob was a major contributor to this study in the area of software architecture, and he authored the position paper "Thinking Outside the Box," attached as Appendix H.
- Kirk Reinholtz is a Principal Engineer on staff with the Flight Software and Data Systems Section. Most recently, he has been working with the Constellation program and the Altair project within that program. Kirk made significant contributions in examining NASA

documents and in summarizing the multi-center discussions on architectures, trades, and avionics impacts, as attached in Appendix B.

- Glenn Reeves is Assistant Division Manager for Flight Projects in the Systems and Software Division, and formerly the chief software engineer for the Mars Pathfinder mission and the Mars Exploration Rover missions.

1.4.4 Johnson Space Center Participants

- Mike Brieden served as Software Architect of the Space Station Freedom, Project Manager of the Space Shuttle CAU, Spacecraft Integration Manager for Orion, and currently serves on the Orion Chief Engineer's staff for Orion Avionics and Software.
- Brian Butcher is a software developer for the Spacecraft Software Engineering Branch. He is currently developing software for Low Impact Docking System (LIDS). Brian served as primary point of contact in the latter phase of this study.
- Rick Coblentz is a long-standing Engineering Directorate Flight Software Branch Chief, with extensive software experience for Space Shuttle, Space Station Freedom, the ISS, and a bunch of other projects. His Branch is CMMI-3 certified, and is very heavily involved in Orion software SSM and GFE.
- Pedro Martinez served as Software Manager for Orion, Station MDM software SSM, and several other GFE projects. He currently serves as Deputy Chief of the Vehicle Systems Management Branch, and is responsible for leading the Branch's work toward achieving its CMMI-3 rating. He also has extensive experience in software development in large industrial settings from the commercial sector (Telecom and IP). Pedro was the primary contact for this study at JSC.
- Carl Soderland has extensive experience in Shuttle, Station, and countless other projects, with emphasis in GNC.
- Emily Strickler served as Software Manager for Space Station Freedom and has spearheaded much of the software GFE process work in the Engineering Directorate. At the time of this study, she served as Branch Chief of the Vehicle Systems Management Branch. Her Branch plays a key role in oversight of the Orion software, and is working toward CMMI-3 for software insight/oversight. She also has extensive experience in software simulations, from her days as a government contractor.

1.4.5 Marshall Space Flight Center

- Darrell Bailey is software project lead for the Huntsville Operations Support Center. Darrell is currently serving as Technical Assistant to the Flight and Ground Software Division and WBS manager for Ares flight software.
- Pat Benson is software project lead for Material Science Research Rack, UPA, and Advance Video Guidance Sensor for Orbital Express (OE AVGS). Pat is currently serving as chief of the Software Systems Engineering Branch within the Flight and Ground Software Division.
- Terry Brown has 22 years in embedded software development including rocket engine control software, satellite control, and Space Station Operation Instrument. He has served as software project lead for Low Cost Technologies engine control software, ISS Urine Processor Assembly (UPA) flight software, and is currently serving as Ares flight software

design lead. He has worked Space Shuttle Main Engine software and is currently Technical Assistant and acting day-to-day branch chief for the Software Development Branch within the Flight and Ground Software Division.

- Keith Cornett is software project lead for AVGS OE. Keith is currently serving as design lead for the Ares Upper Stage Command and Telemetry Computer and team lead of the Software Planning and Support Team of the Software Systems Engineering Branch within the Flight and Ground Software Division.
- Robert “Tim” Crumbley is Special Assistant to the Space Systems Department, responsible for Ares development. He has over 20 years of experience in software engineering, particularly in spacecraft and launch vehicle avionics and software systems. Tim contributed to the NASA Software Engineering Requirements NPR and is currently co-chair of the NASA Software Working Group.
- Helen Housch is a senior systems engineer in Cepeda Systems & Software Analysis, Inc., working at MSFC.
- Steve Purinton is lead software engineer for Chandra and numerous other software projects over 40-something years. Steve is currently serving as chief of the Avionics and Software Ground Systems Test Branch within the Flight and Ground Software Division, developing the Systems Integration Lab for Ares.

1.5 Acknowledgements

There are many people who contributed to this study and deserve acknowledgement. However, I first wish to thank NASA’s Office of Chief Engineer and the individuals who initiated and funded this work under the Technical Excellence Initiative. Specifically, Adam West managed the study and was always helpful in answering my questions. Chief Engineers Frank Bauer (ESMD), Stan Fishkind (SOMD), Ken Ledbetter (SMD), and George Xenofos (ESMD Deputy Chief Engineer) identified the need for and importance of a study on this topic.

Next, I thank John Kelly, OCE Program Executive for Software Engineering, and Tim Crumbley, both of whom served as technical advisors and who offered constructive feedback and helpful suggestions. I am also indebted to four individuals at JPL who provided substantial technical inputs to the study and helpful discussions during the study: Gerard Holzmann (Lead scientist of the Laboratory for Reliable Software), Robert Rasmussen (Chief Engineer, Systems and Software Division), and William “Kirk” Reinholtz (Flight Software and Data Systems) had each prepared an analysis of flight software complexity that seeded this study. I gained many insights from discussions with them. Also, Kevin Barltrop (Autonomy and Fault Protection Group) led the entire examination of complexity in fault protection and prepared a report, incorporated as Appendix F.

I offer special thanks to my primary technical contacts at other centers: Harold “Lou” Hallock at GSFC, Cathy White and Helen Housch at MSFC, Pedro Martinez and Brian Butcher at JSC, and Steve Williams at APL. They participated in our biweekly teleconferences and organized visits at their respective centers where much of the key information was collected that led to our findings and recommendations. Although not formally part of the study, I also received helpful feedback and suggestions from Michael Aguilar (Software Lead, NESC), Mary “Pat” Schuler (Head of the

LaRC Software Engineering Process Group), and John Hinkle (Independent Verification and Validation Project Manager [NASA IV&V Facility] for the International Space Station and Space Shuttle Programs).

Finally, there are several individuals who I thank for high-level guidance, both technical and programmatic: Richard Brace, JPL Chief Engineer, gave valuable feedback on an early version of findings and recommendations, as did David Nichols, Manager of JPL’s Systems and Software Division; Bob Vargo, Manager of JPL’s Flight Software and Data Systems Section, was continuously engaged in the study and offered valuable comments and suggestions; and Chi Lin, Manager of JPL’s Engineering Development Office and program manager for this study, provided enthusiastic programmatic support for the study and its recommendations.

2 Findings and Recommendations

This study produced 16 recommendations, each of which is described separately in this section. Completely after the fact, the recommendations were grouped into five categories shown in Table 1. If there is a single motivation common to many of the recommendations, it is that prevention and early detection is cheaper than redesign and rework. As Boehm showed in a 1981 survey of large software projects, the cost of fixing a defect rises in each subsequent phase of a project, costing 20–100 times more to fix a problem during operation than it would have cost to correct the problem during the requirement definition phase [Boehm 1981].

Table 1. Recommendations grouped into five categories.

Category	Sec.	Recommendation
Complexity Awareness	2.1	Raise awareness of downstream complexity
Project Management	2.2	Emphasize requirements rationale
	2.3	Emphasize trade studies
	2.7	Involve operations engineers early and often
	2.10	Stimulate technology infusion
	2.16	Use software metrics
Architecture	2.4	More early analysis and architecting
	2.5	Establish architecture reviews
	2.6	Nurture software architects
	2.9	Invest in reference architecture
Fault Protection	2.12	Standardize fault protection terminology
	2.13	Establish fault protection proposal review
	2.14	Create fault protection education
	2.15	Research fault containment techniques
Verification	2.8	Analyze COTS software for test complexity
	2.11	Apply static analysis tools

2.1 Raise Awareness of Downstream Complexity

2.1.1 Finding

Although engineers and scientists don't deliberately make things more complex than necessary, it sometimes happens because they don't realize the downstream complexity entailed by their local decisions. For example, if a scientist or systems engineer levies an unnecessary requirement, or a requirement with an unnecessarily high performance target, that can spawn a cascade of increased complexity in analysis, design, development, testing, and operations. Similarly, a hardware engineer might choose a hardware interface design that entails a hard-to-meet timing requirement on avionics software; a systems engineer might omit a simple capability from flight software that then complicates operations; a software team might neglect testability issues in flight software that then complicate integration and testing; an instrument engineer might advocate using a legacy instrument without modification, not realizing the difficulties it imposes on other flight software to accommodate it; and a project manager might elect to reuse "flight-proven" software from a previous mission, unaware that it won't scale well to meet the more ambitious requirements of the new mission. In short, flight projects are multidisciplinary activities that involve many specialties; while specialists are often aware of complexities in their own domain, they are much less aware of complexities they cause in other domains.

2.1.2 Recommendation

Raise awareness about complexity through educational materials that provide easy-to-understand examples from previous missions. Such material should be provided on a NASA-internal web site that everyone can see and wherein everyone can contribute their own examples. This could be part of the NASA Lessons Learned Information System, categorized under the keyword “complexity”. As an example of complexity lessons, we have provided a “Complexity Primer” in Appendix C. The primer contains several stories, each ending with a lesson. Certainly there are many other stories that could be captured and shared, so this primer should be regarded as a small example of what can be developed.

This finding about “lack of awareness” is related to two other findings: lack of attention to trade studies (Section 2.3) and the need for more up-front analysis and architecting (Section 2.4). Trade studies bring together engineers from multiple disciplines to help inform decision-making, just as multidisciplinary analysis does in the early stages of systems engineering, including development of architecture.

2.2 Emphasize Requirements Rationale

2.2.1 Finding

Requirements that are unnecessary or that specify unnecessarily stringent performance targets cause extra work and add complexity, whether in analysis, design, software, testing, operations, or some combination thereof. The standard defense against unnecessary requirements is a statement of rationale that substantiates *why* a particular requirement is necessary. However, the study found that rationale statements are often missing or superficial, or even misused in the sense of providing more detail about a requirement (rather than substantiating it). In one mission a scientist levied a requirement for “99% data completeness,” implying that science results would be greatly diminished by any interruptions in science observations. The flight software team took the requirement seriously and designed and developed a system with redundant elements and fast onboard fault detection and response, making for a more complex system. Later in the project, somebody questioned the value of 99%, and the scientist—realizing that it was overly stringent—quickly relaxed the requirement. Unfortunately, the damage was already done; an unsubstantiated requirement had spawned an unnecessary cascade of time-consuming analysis, design, development, and testing, not to mention the dismay of the software team who had worked so hard.

2.2.2 Recommendation 1

Project management should emphasize the importance of requirements rationale to the people who write requirements and ensure that they know how to write a proper rationale. The practice of writing rationale statements should be familiar to anyone involved in flight projects because it *is* recommended in existing NASA documents, though not mandated. NPR 7123.1A (NASA Systems Engineering Processes and Requirements) specifies in an appendix of “best typical practices” that requirements include rationale, though it offers no guidance on how to write a good rationale or check it. The NASA Systems Engineering Handbook (NASA/SP-2007-6105

Rev1) *does* provide some guidance (p. 48), though more details and examples would be helpful. To put further emphasis on this important practice, NASA should at least encourage local practices that mandate rationale, if not actually mandate it in an update of NPR 7123.

Regarding the writing of rationales, sometimes a *group* of requirements are so closely coupled that a rationale for one is essentially the same as for the others. In those cases it is more sensible to write a rationale that applies to the group. Such a rationale may include the objective to be achieved, what the trade studies showed, and decisions about what needs to be done.

2.2.3 Recommendation 2

Recipients of hard-to-meet requirements should not suffer in silence; they should inform project management when a requirement—even a well-substantiated requirement—entails a high degree of complexity in the solution. Requirements are not always absolute; in those circumstances when they are, it's far better to inform project management so that sufficient resources are directed toward a solution.

2.3 Emphasize Trade Studies

2.3.1 Background

Flight projects, by their very nature, are multidisciplinary activities involving several engineering disciplines and specialties. Some of the most important communication among disciplines occurs within trade studies, where engineers get their heads together to decide how best to meet requirements and address design issues. Trade studies are a best practice in the design of complex systems because they yield better decisions for the project as a whole than if made in isolation by individual engineers or teams. The *Best Practices Clearinghouse* at Defense Acquisition University list trade studies as one of the practices having the most evidence (https://bcph.dau.mil). Within flight projects, trade studies are conducted to make choices about how to allocate capabilities between flight and ground, between hardware and software, between flight computer and instruments, and more. Trade studies are especially important to software because software has a wide multidisciplinary span in a flight system. "As the line between systems and software engineering blurs, multidisciplinary approaches and teams are becoming imperative" [Ferguson 2001].

2.3.2 Finding

Our study found that trade studies are often *not* done, or done superficially, or done too late to make a difference. Every trade study not done is a missed opportunity to reduce complexity. Trade studies are *already* recommended in NASA documents and local procedures, so the natural question is "why aren't they done?" One possible answer is schedule pressure; another possible answer is unclear ownership, since trade studies often involve two or more teams. One way to clarify ownership of a trade study between *x* and *y* is to make it the responsibility of the manager who holds the funds for both *x* and *y*. Another approach that facilitates trade studies, albeit informally, is collocation of project members, which is universally praised by those who have experienced it. Collocation makes it more likely that engineers from different disciplines will discuss issues of shared interest, and do so frequently, though the results are less likely to be formally documented.

2.3.3 Recommendation

Emphasize the importance of trade studies to project members and clarify who is responsible for what kinds of trade studies. Like the recommendation to substantiate requirements with rationales, this recommendation is really an alert to project managers to give more emphasis to these good practices. Trade studies are important in reducing incidental complexity for the reason cited in Section 2.1: namely, lack of awareness of downstream complexity. Also, as Section 2.7 will point out, operators should be included as important stakeholders in any decision forums that might affect operational complexity, and some of those forums will be trade studies.

2.4 More Early Analysis and Architecting

2.4.1 Background

It is a truism in engineering projects that the earlier that a problem or mistake is found, the cheaper it is to address; so it is with complexity. As soon as requirements of a certain level have been vetted, what lies ahead is the task of creating a software architecture—an architecture that handles the many functional and non-functional requirements. Every software system has an architecture, just as every building has an architecture; however, some architectures are better suited than others for the task at hand, just as some office buildings are better suited to the needs of their tenants than others. In addition to satisfying the functional requirements, a good software architecture must satisfy a number of quality attributes such as flexibility, maintainability, testability, scalability, interoperability, etc.

2.4.2 Finding

The study team found evidence of poor software architecture. This is unsurprising for two reasons. First, the very topic of "software architecture" is relatively recent in software engineering curricula. For example, one of the earliest books on the subject was published in 1996 [Shaw & Garlan, 1996]. Thus, very few people who are well educated in software architecture have risen into positions of authority in flight projects. Second, the topic has gained importance as the size of software systems has grown, and good architecture can make the difference between a project that succeeds and a project that collapses amidst unending problems in integration and testing. Famous computer scientist Alan Kay says "point of view is worth 80 IQ points", and, indeed, architecture represents a point of view—for better or worse—through which a software design takes shape.

2.4.3 Recommendation

If we view a project as a zero-sum game, meaning that there is a fixed budget to be allocated across different phases of work, what can be done to improve the odds of success? The answer is that most projects should allocate a larger percentage of resources to early analysis and architecture in order to avoid later problems and rework, when it is more costly to fix. The COCOMO II cost estimation model, reflecting industrial experience over a wide range of software projects, supports the value of applying resources to architecting. As Figure 2 shows, there is an investment "sweet spot" for architecting that depends on the size of the system. For example, for a system that is projected to be 100,000 lines of code, 21% of the software budget should be spent on architecting to minimize the overall cost. It follows then, that the larger the system, the larger the percentage to minimize overall cost. Note that underfunding of architecture is worse than overfunding because a weak architecture requires more rework during development and integration and testing.

This recommendation is related to the recommendation 2.9 about investing in a "reference architecture." Briefly, a *reference architecture* is a reusable architecture, designed for the needs of a problem domain (such as spacecraft control) that will be adapted for specific projects. Thus, a reference architecture gives projects a running start in their project-specific architecting work.

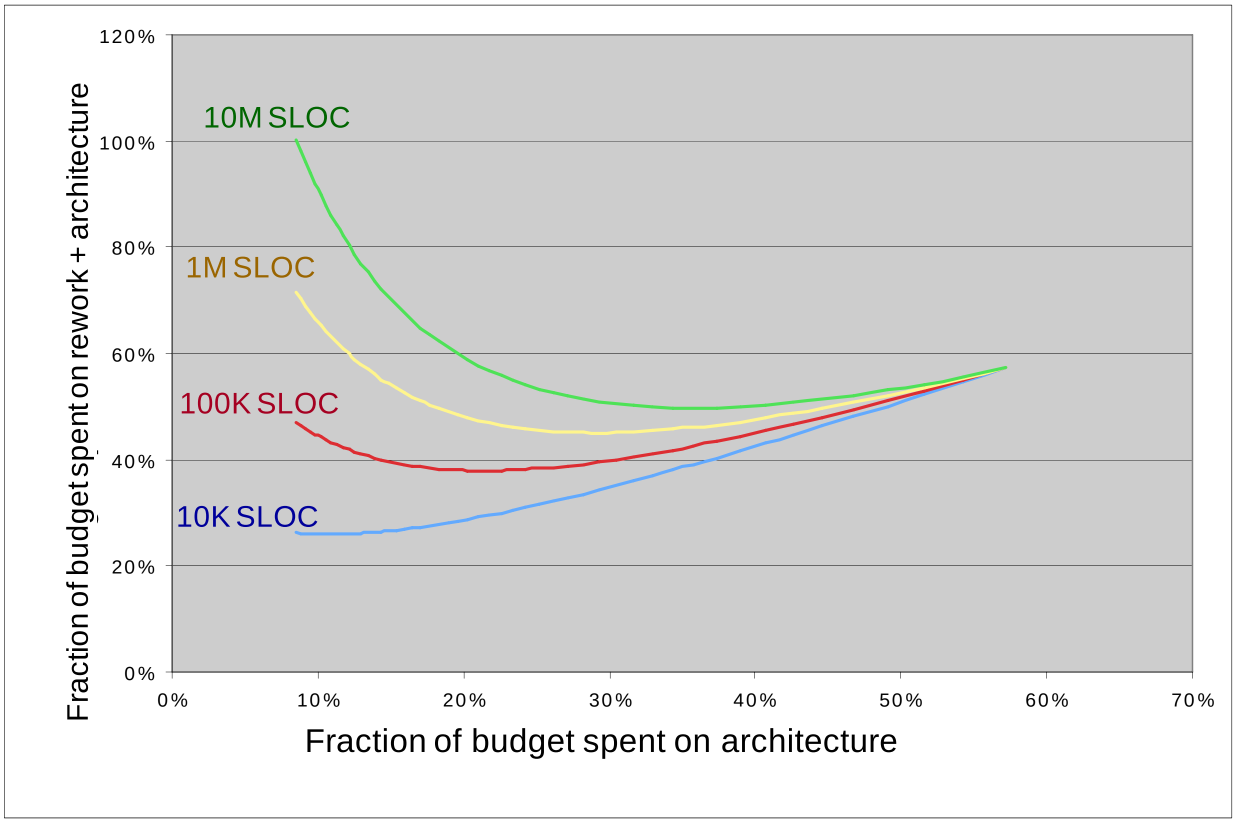


Figure 2. Equations from the COCOMO II model for software cost estimation show that the "sweet spot" for investing in software architecture increases with software size.

2.5 Establish Architecture Reviews

2.5.1 Finding

The telecommunications industry has long been a major developer of software-intensive systems, especially since the advent of computer-controlled electronic switching systems in the 1960s. Over time, AT&T noticed that large, complex software projects were notoriously late to market, often exhibited quality problems, and didn't always deliver on promised functionality. In an effort to identify problems early in development, AT&T began a novel practice of "architecture reviews" starting in 1988 and conducted more than 700 reviews between 1989 and 2000. They found that such reviews consistently produced value—saving an average of $1 M each on projects of 100,000 non-commentary source lines—and they preserved the process even in the most difficult and cost-cutting times [Maranzano 2005].

Maranzano et al. stated that architecture reviews are valuable because they:

- Find design problems early in development, when they're less expensive to fix.
- Leverage experienced people by using their expertise and experience to help other projects in the company.
- Let the companies better manage software components suppliers.
- Provide management with better visibility into technical and project management issues.
- Generate good problem descriptions by having the review team critique them for consistency and completeness.
- Rapidly identify knowledge gaps and establish training in areas where errors frequently occur (for example, creating a company-wide performance course when many reviews indicated performance issues).
- Promote cross-product knowledge and learning.
- Spread knowledge of proven practices in the company by using the review teams to capture these practices across projects.

AT&T's pioneering practice caught the attention of the Department of Defense; consequently, the Defense Acquisition University now recommends it on their *Best Practices Clearinghouse* web site (https://bpch.dau.mil) as one of the practices that have the most supporting evidence. Similarly, the Software Engineering Institute states that "a formal software architecture evaluation should be a standard part of the architecture-based software development life cycle," and has published a book that describes three methods for evaluating software architectures [Clements 2002]. It's important to understand that architecture reviews, as practiced at AT&T, were designed to help projects find problems early in the lifecycle. These reviews were invited by the projects, not mandated.

2.5.2 Recommendation

NASA should establish an architecture review board, either on a center-by-center or NASA-wide basis and begin offering reviews to willing projects. The article in *IEEE Software* provides an excellent description that includes principles, participants, process, artifacts, lessons learned, and organizational benefits [Maranzano 2005]. Among those benefits is enhancement of cross-organizational learning and assistance in organizational change. Thus, benefits extend beyond the reviewed projects. NASA should tune AT&T's review process for flight projects, leveraging existing materials for software architecture reviews within NASA, such as "Software Architecture Review Process" [Weiss 2008] and "Peer Review Inspection Checklists" [Schuler 2007]. Also, NASA could strengthen NPR 7123 regarding *when* to assess software architecture.

2.6 Nurture Software Architects

2.6.1 Background

In a fascinating book about human-made disasters due to unintended consequences, author Dietrich Dörner illustrates that many well-educated people have difficulty understanding complex systems [Dörner 1989]. Complex systems are interrelated in that an action meant to affect one part of a system might also affect other parts (side effects) and might have long-term repercussions. Spacecraft are like this, so it's important to have engineers on each project who deeply understand the interdependencies among a system's variables and how to control such a system. A good architecture is extraordinarily important in helping engineers understand a system, better manage essential complexity, and minimize incidental complexity.

2.6.2 Finding

The two preceding recommendations (2.4 and 2.5) have highlighted the importance of software architecture, a responsibility that falls to a project's software architect. Unfortunately, relatively few software engineers are skilled at architecting or even have formal training in the subject; this situation confronting NASA is not uncommon in industry. General Motors, for example, recognized a need for better training in software engineering about 10 years ago and enlisted Carnegie Mellon University (CMU) to establish an internal training program via distance delivery. The program is similar to CMU's on-campus Masters in software engineering degree, but tailored for GM's focus on embedded, real-time systems, and includes a one-semester course on software architecture. During a span of 10 years, the program graduated several hundred students. The project component of the program is done on GM projects. As a result, a significant number of students have received awards for the work done in their projects based on patents and overall company impact. CMU has also developed various practitioner-oriented industry courses on software architecture, and has taught them at Sony, Samsung Electronics, LG Electronics, and Boeing.

2.6.3 Recommendation

Increase the ranks of well-trained software architects and put them into positions of authority on projects. One way to increase the ranks is through strategic hiring of experienced software architects. Another way is to nurture budding architects within our ranks through education and mentoring. In particular, NASA could establish an internal training program similar to that of GM, perhaps as APPEL courses that are created or extended to train software architects and systems engineers. Similarly, the Systems Engineering Leadership Development Program (SELDP) should include training in architecture because systems engineers are responsible for the "architecture of behavior," and behavior in modern spacecraft is mostly controlled by software.

2.7 Involve Operations Engineers Early and Often

At the tail end of the engineering lifecycle sits Phase E (operations). During Phase E, any forms of complexity that haven't already been addressed fall to the operations staff. Of course, many requirements (and their entailed complexity) are deliberately assigned to operations to take advantage of human abilities to deal with the unusual or unexpected or difficult, but others are less well reasoned, and any decisions that increase operational complexity also increase risk of operational error.

2.7.1 Finding

Generally, there are two sources of incidental complexity that complicate testing and operations: shortsighted flight software decisions and operational workarounds. In the former category are decisions about telemetry design, sequencer features, data management, autonomy, and testability. In one example from Cassini, a decision to descope a sequencer feature was eventually reversed when an operations engineer showed that certain kinds of science activities would be impossible. In other spacecraft, a lack of design-for-testability made it impossible to run many test cases that, in principle, required only a few minutes of testbed time for each. Instead, the testers had to essentially "launch" the spacecraft, go through detumble, acquisition, etcetera in order to get the flight system into the desired state for the test. The setup procedure could take as long as 3 hours; that procedure was repeated for every test! The result is that far less testing was accomplished, so failures that could have been caught before launch were likely to manifest during operations, when they're much harder to diagnose.

In the second category, operational workarounds typically result from decisions made about flight software to descope some capabilities and/or *not* fix some defects. Such decisions are often made under schedule pressure, and the first question asked is "Does an operational workaround exist?" If the answer is 'yes,' then that's often the end of discussion, without much consideration of the increased burden on operations. To be fair, each operational workaround is typically a small increase in complexity, but the number of such workarounds can grow into the hundreds, and *that* becomes dangerous because it requires operators to be aware of many exceptional cases.

2.7.2 Recommendation

Projects should involve experienced operators early and often. They should be involved in trade studies and descope decisions, and they should try to quantify the growing workload (or risk) as operational workarounds accumulate, one by one. Also, as a way to raise awareness of operational difficulties, systems engineers and flight software developers should take rotational assignments as operators. The Apollo Program recognized the value of operational experience: "To use the experience gained from Project Mercury and the Gemini Program, engineers with operational background from these programs were involved in all major Apollo design reviews. This procedure allowed incorporation of their knowledge as the Apollo design evolved. This involvement proved a key factor in producing spacecraft that have performed superbly so far" [Kleinknecht 1970].

2.8 COTS Software: A Mixed Blessing

2.8.1 Finding

Commercial off-the-shelf (COTS) software can provide valuable and well-tested functionality, but sometimes comes bundled with additional features that are not needed and cannot easily be separated. Since the unneeded features might interact with the needed features, they must be tested too, creating extra work. Also, COTS software sometimes embodies assumptions about the operating environment that don't apply well to space borne applications. If the assumptions are not apparent or well documented, they will take time to discover. This creates extra work in testing; in some cases, a *lot* of extra work.

2.8.2 Recommendation

Make-versus-buy decisions about COTS software should include an analysis of the COTS software to: (a) determine how well the desired components or features can be separated from everything else, and (b) quantify the effect on testing complexity. In that way, projects will have a better basis for make/buy and fewer surprises.

2.9 Invest in Reference Architecture

2.9.1 Finding

Although every NASA mission is unique in some way, all flight systems must provide a common list of capabilities: navigation, attitude control, thermal control, uplink, downlink, commanding, telemetry, data management, fault protection, etc. Details vary from mission to mission, of course, but the existence of these capabilities and their interdependencies with other capabilities do not; as a result, projects rarely "start from scratch". Instead, they typically adapt software from a previous similar mission with high hopes of substantial cost savings. Often, the real cost savings is lower than projected because the re-used software was never designed for adaptation, so the software team ends up examining and touching almost every line of code.

What's needed to climb out of this "clone-and-own" approach to software re-use is a reference architecture and core assets, designed for adaptability. IBM Rational regards reference architecture as "the best of best practices" and defines a reference architecture as "a predefined architectural pattern, or set of patterns, possibly partially or completely instantiated, designed, and proven for use in particular business and technical contexts, together with supporting artifacts to enable their use" [Reed 2002]. Importantly, a reference architecture can embody a huge set of lessons learned, best practices, architectural principles, and design patterns, so it can "operationalize" many good ideas that are otherwise just words in documents. A lesson is not learned until it is put into practice, and reference architecture provides a vehicle for doing that.

2.9.2 Recommendation

Earmark funds, apart from mission funds, for development and sustainment of a reference architecture at each center. The investment should include "core assets" that include software components—designed for adaptation and extension—that conform to the reference architecture.

Note: NASA/GSFC has taken steps in this direction to develop a reusable flight software system known as the "Core Flight Software System" [Wilmot 2007]. The architecture emphasizes messaging middleware and plug-and-play components, and has proven valuable at reducing development costs on new missions. Similarly, NASA/JPL has been developing the "Mission System Architecture Platform." In all such cases, a reference architecture should be subjected to careful review (Recommendation 2.5) since it will shape the architecture of many missions.

2.10 Stimulate Technology Infusion

2.10.1 Finding

Flight software engineers typically move from project to project, often with little time to catch up on advances in technology, and, therefore, tend to use the same technology, tools, and techniques as before. Project managers who wish to use only flight-proven technology, believing that that reduces risk, implicitly encourage this state of affairs. However, the discipline of software engineering is still evolving rapidly. Technical growth in this area is important not only for our flight software engineers but also for NASA's technical leadership in the aerospace industry.

2.10.2 Recommendation

Take proactive steps to stimulate technology infusion by helping make flight software engineers aware of newer technologies, tools, methodologies, best practices, and lessons learned. One option is to hold a days-long technical kickoff meeting at the point in a project when the core flight software team is formed to expose them to the new ideas and, hopefully, inspire some of them to champion certain ideas within the project. Another option is to provide a 4-month in-house "sabbatical" for each software engineer to learn a TRL 6 technology, experiment with it, give feedback for possible improvements, and possibly infuse the technology into the next flight project.

2.11 Apply Static Analysis Tools and Code Compliance Checkers

2.11.1 Background

"Static analysis" is the analysis of computer software that is performed without actually executing programs built from that software. Static analysis tools analyze source code to detect a wide range of problems including uninitialized variables, memory leaks, array overrun, erroneous switch cases, null pointer dereference, use of memory after freeing it, and race conditions [Coverity 2008]. Static analysis tools can discover in minutes problems that can take weeks to discover in traditional testing. Some tools are able to analyze millions of lines of code. Importantly, the tools can be used early, when source code first becomes available, and continuously on all software builds.

2.11.2 Finding

Commercial tools for static analysis of source code are mature and effective at detecting many kinds of software defects, but are not widely used. Examples of such tools—with no implied endorsement—include GrammaTech CodeSonar™, Coverity Prevent™ and Klocwork Insight™.

In addition to static analysis, most flight software organizations in NASA have coding rules intended to instill good software coding practices and avoid common errors. Unfortunately, observations of actual flight software show that such coding rules are often violated, negating the value of the rules. Some static analysis tools provide extensions for code compliance checking.

2.11.3 Recommendation

Software developers for both flight *and* ground software should adopt the use of static analysis tools and coding rule compliance checkers. Widespread use should be the norm. Funding should be provided for site licenses at each center along with local guidance and support. Software experts within NASA and industry should be polled regarding the best tools.

2.12 Standardize Fault Protection Terminology

2.12.1 Finding

Within the fault protection area of systems engineering there isn't consistent terminology across NASA centers and aerospace contractors. Common terms used in discussion—such as *fault, failure, error,* and *single fault tolerance*—are defined differently at different institutions, causing confusion and miscommunication within project teams. Further, the industry lacks good reference material from which to assess the suitability of different fault protection approaches.

2.12.2 Recommendation

NASA should publish a NASA Fault Protection Handbook or Standards Document that: (a) provides a standard lexicon for fault protection and (b) a set of principles and features that characterize software architectures used for fault protection. The handbook should provide a catalog of design patterns with assessments of how well they support the identified principles and features.

2.13 Establish Fault Protection Proposal Review

2.13.1 Finding

The current proposal review process does not assess in a consistent manner the risk entailed by a mismatch between mission requirements and the proposed fault protection approach. This leads to late discovery of problems.

2.13.2 Recommendation

Each mission proposal should include an explicit assessment of the match between mission scope and fault protection architecture. Penalize proposals or require follow-up for cases where proposed architecture would be insufficient to support fault handling strategy scope. At least one recent mission recognized the fault handling strategy scope problem, but did not appreciate the difficult of expanding the fault handling strategy using the existing architecture. The handbook or standards document (Section 2.12) can be used as a reference to aid in the assessment and provide some consistency.

2.14 Develop Fault Protection Education

2.14.1 Finding

Fault protection and autonomy receive little attention within university curricula, even within engineering programs. This hinders the development of a consistent fault protection culture needed to foster progress and exchange ideas. Further, it means that most fault protection engineers must learn the skills through apprenticeship from a master, and the results vary depending on the skills of the master.

2.14.2 Recommendation

NASA should sponsor or facilitate the addition of fault protection and autonomy courses within university programs, such as a Controls program. As an example, the University of Michigan could add a “Fault Protection and Autonomy Course.” Other specialties (such as circuit design) have developed specialized cultures that allow for continuing improvement; autonomy and fault protection should do the same. A good outcome would be clarity and understanding in discussions of fault protection among different institutions and less need to recruit new practitioners from outside the field.

2.15 Research Fault Containment Techniques

2.15.1 Finding

The following thought experiment motivates the next recommendation. At the current state of the practice in software engineering, a large software development project having a good software development team and a very good process will produce approximately one residual defect per thousand lines of code. (A *residual defect* is a defect that remains in the code after all testing is complete.) Thus, if a flight system has one million lines of flight software, then it will have 1000 residual defects. If we further assume that the number of defects decreases by an order of magnitude with each increase in severity, then the system will harbor approximately 900 benign defects, 90 medium-severity defects, and 9 potentially fatal defects. These numbers are approximate, of course, and NASA's vulnerability might be less than the numbers suggest due to the practice of "test as you fly and fly as you test", but the prediction is worrisome nonetheless. Given current defect rates and exponential software growth, what can be done to further reduce risk?

2.15.2 Recommendation

Extend the techniques of onboard fault protection to cover a wider range of software failures. As currently practiced, fault protection engineers largely focus on hardware because it is subject to infant-mortality failures, wear-out failures, and random failures. Less attention is given to software because it doesn't exhibit such failures; the fact remains, however, that residual defects in software are ever-present, waiting for the right conditions to cause a failure. In particular, one area of increasing vulnerability is algorithmic complexity, where sophisticated algorithms are used to improve performance along some dimension, albeit at the cost of more complex code that is harder to verify. One way that the automotive industry addresses this kind of vulnerability is to: (a) add the ability to detect misbehavior from an algorithm, where possible, and (b) when misbehavior is detected, switch to a simpler algorithm that provides "limp home" ability. In order for this approach to reduce risk, the simpler algorithm should be an order of magnitude smaller so that it is much more verifiable. Appendix E discusses this idea in more detail. The recommendation here is to fund research to apply this approach on the most suitable areas of flight software.

2.16 Collect and Use Software Metrics

2.16.1 Background

This study of flight software complexity begs an obvious question: "How do you define and measure software complexity and other forms of complexity (requirements complexity, architecture/design complexity, testing complexity, and operations complexity)?" Some of these topics are well studied in the literature, others less so. This study could have spent a lot of time trying to precisely define and measure various forms of complexity. Instead, we adopted some fairly general descriptions of complexity (see Section 4) and devoted most of our time to identifying areas ripe for improvement because there is so much "low-hanging fruit." Nonetheless, software metrics deserve attention. If NASA was to commission a follow-up study on flight software complexity in 5 or 10 years, the availability of better software metrics could help focus the study.

In principle, metrics can answer questions such as:

- What types of flight software exhibit the largest growth?
- What types of flight software exhibit the highest defect rates?
- Where have the most serious defects originated (requirements, architecture, design, implementation, operations)?
- How well do existing complexity measures predict defect rates?
- How well do different architectures reduce incidental complexity?
- What coding guidelines have provided the most benefit?
- How do defect rates differ among programming languages?

2.16.2 Finding

Some of the questions listed above have been (or will be) addressed by academic research. It's important, however, to know answers that are specific to NASA, or at least to the broader aerospace community. However, there is currently no consistency in collection of software metrics across NASA or even within a center, so it is currently difficult to answer such questions based on objective, historical data. NASA Software Engineering Requirements (NPR 7150.2) already recommend that "software measurement programs at multiple levels should be established to meet measurement objectives" with respect to planning and cost estimation, progress tracking, software requirements volatility, software quality, and process improvement. However, these recommendations are not specific and not mandated.

2.16.3 Recommendation

NASA centers should collect software metrics and use them for management purposes. The NPR 7150.2 guidelines, while useful, could be made more specific in order to provide the data needed to answer questions such as those listed above. In truth, those questions are merely suggestive; the first step is to identify the questions whose answers will provide the most benefit in updating NASA software policies, procedures, and practices. Having said that, it will be difficult for any group of software experts to anticipate all the questions that will be important 5 or 10 years in the future. Fortunately, software source code is easy to archive for the purpose of future, unspecified analyses. Thus, this recommendation includes archival storage of major flight software releases, including data about in-flight failures and the causative defects.

In attempting to define stronger guidelines for software metrics, it's important to be aware of some concerns and issues that will complicate such an effort, as evidenced by a prior attempt to define NASA-wide software metrics. Will the data be used to compare productivity among centers, or compare defect rates by programmer, or reward/punish managers? How do you compare class A software to class B software? Should contractor-written code be included in a center's metrics? Is a line of C code worth more than a line of assembly language? Should auto-generated code be counted? How should different software be classified (flight vs. ground vs. test, spacecraft vs. payload, new vs. heritage vs. modified vs. COTS, software vs. firmware, etc.)? It's hard to answer such questions in the abstract, but much easier in the context of a specific study. For that reason, the creation of a flight software archive will simplify future studies because it doesn't require anticipating all the relevant questions and devising metrics for them.

3 Flight Software Growth

3.1 Flight Software Described

Flight software, as the name suggests, is software that executes onboard a spacecraft[1], whether it be a rocket, satellite, observatory, space station, lander, rover, or lunar habitat. Flight software is commonly known for the mission-level capabilities it provides, such as: guidance, navigation, and control (GN&C); entry, descent, and landing (EDL); surface mobility; science observations; environmental control; instrument control; and communications. Less well known, but equally important, is a large body of flight software sometimes described as "platform" or "framework" or "infrastructure" software that provides important services and that is re-used from mission to mission, though often customized for the avionics hardware. This software provides services such as computer boot-up and initialization, time management, hardware interface control, command processing, telemetry processing, data storage management, flight software patch and load, and fault protection. The last of these—fault protection—has been such a source of difficulty in several missions that it has been given special attention in this study, as described in Section 5.

Four characteristics distinguish flight software from most of the familiar commercial software used in daily engineering and management work [Aldiwan & Wang 2007]:

1. Flight software has no direct user interfaces such as monitor, keyboard, and mouse. This means that during mission operations all interactions between operators and the flight system occur through uplink and downlink processes. This lack of direct interface, coupled with round-trip light-time delays, makes problem diagnosis a tedious process.
2. Flight software interfaces with many flight hardware devices such as thrusters, reaction wheels, star trackers, electric motors, science instruments, radio transmitters and receivers, and numerous sensors of temperature, voltage, position, etc. A key role of flight software is to monitor and control all these devices in a coordinated way.
3. Flight software typically runs on processors and microcontrollers that are relatively slow and memory-limited compared to mainstream processors, such as those in commercial notebook computers. Programming in an environment of limited resources and hardware interfaces requires additional expertise, which is why not every software engineer is qualified to work on flight software.
4. Flight software performs real-time processing, meaning that it must satisfy various timing constraints, such as periodic deadlines in control loops, accurate execution of timed events, and maximum reaction time to asynchronous events. Real-time software that performs the right action, but too late, is the same as being wrong.

One characteristic erroneously attributed to flight software is that it is nondeterministic, particularly in discussions about "autonomy software." In truth, flight software is *deterministic*, meaning that given the exact same states, it will behave exactly the same way every time. If operators on Earth could have perfect knowledge of the evolving state of a spacecraft and the environment it is sensing, then they could predict perfectly a spacecraft's behavior. However,

[1] "Flight software" also applies to aircraft software, though not considered in this study.

they don't have perfect knowledge because models of the environment and models of spacecraft hardware are imperfect. The real issue is not determinism, but *predictability*. Unpredictability and uncertainty are inherent characteristics of operating a spacecraft, and most other physical systems, but it is not due to software. Nonetheless, engineers have developed means of robust control that accommodate some uncertainties. For example, in the case of an attitude control system (ACS) for a 3-axis-stabilized spacecraft, operators on Earth cannot predict the exact moments when ACS will fire thrusters, but they trust its estimation and control algorithms to achieve a desired attitude through the principles of closed-loop control.

The four characteristics listed above are characteristics of what industry calls "embedded real-time software" (ERTS). Flight software for NASA's spacecraft, rovers, and rockets are examples of ERTS. Non-NASA examples include military and civil aircraft (e.g., engine controls, flight controls), automobiles (e.g. power-train control, anti-lock brakes), and industrial processes (e.g., power generation, chemical processing). Thus, NASA's engineers are members of a large community that share similar concerns.

3.2 Growth of NASA Flight Software

The first objective of this study was to measure the growth in flight software size and identify trends within NASA. There are several ways to measure software size; we chose a metric based on source lines of code (SLOC), since it is an objective measure that is easily available. SLOC is simply a count of the number of lines in a program's source code, typically adjusted for logical statements rather than physical lines. When further adjusted to remove comment lines, the metric is non-comment source lines (NCSL). It's important to note that NCSL is *not* a good measure of functionality or complexity; short software components can be very complex, and long software components can be very simple. NCSL was chosen because our main purpose in this section was to identify trends and roughly quantify growth rate. Section 4 discusses the meaning and sources of complexity in more detail. Section 2.16 lists some of the challenges of obtaining and using software metrics in general.

Figure 3 shows growth in flight software size for human missions (in red) and robotic missions (in blue) from 1968 through 2005. The 'year' used in this plot is typically the year of launch or completion of primary software. Line counts are either from best available source or direct line counts (e.g., for the JPL and LMA missions). Note that the scale is logarithmic, so the trend lines depict an exponential growth rate of a factor of 10 approximately every 10 years. Interestingly, this trend line conforms with an observation made by Normal Augustine, past chairman of Lockheed Martin Corporation, when he noted in one of his laws ("Augustine's Laws") that "software grows by an order of magnitude every 10 years" [Augustine 1977].

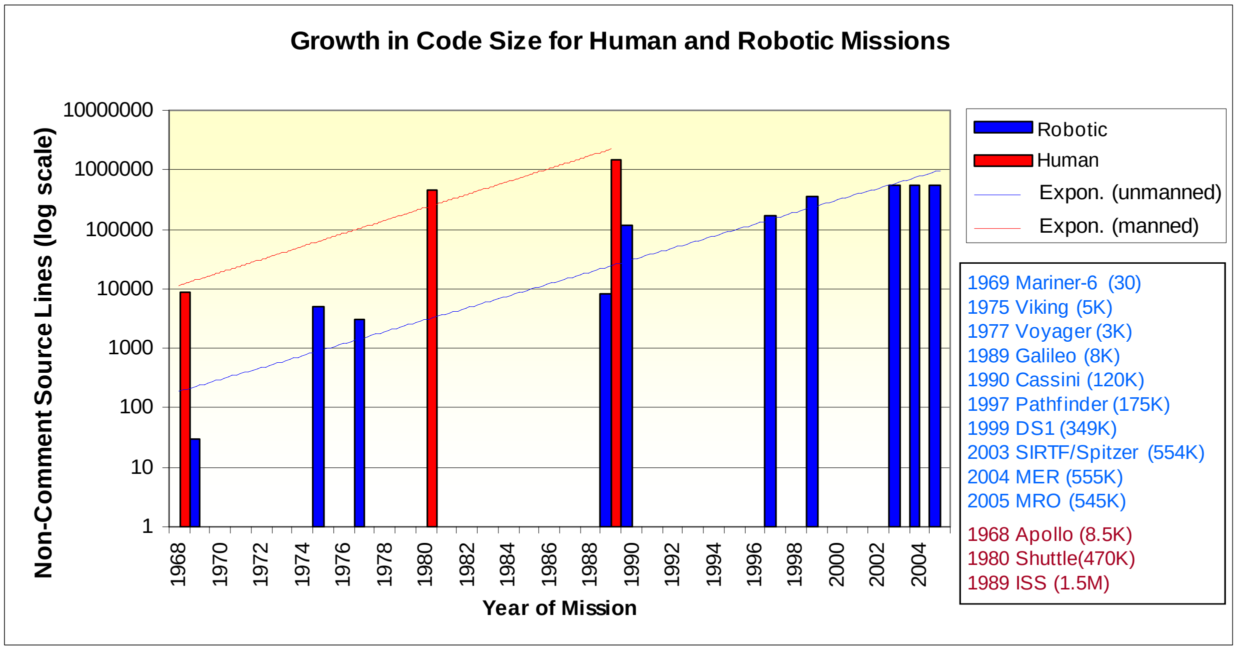


Figure 3. History of flight software growth in human and robotic missions.

In the realm of human spaceflight, there are only three NASA data points (Apollo, Space Shuttle, and International Space Station), so there is much less confidence in projecting a trend. Orion flight software size estimates exceed 1 million lines of code, which means that it will be more than 100 times larger than Apollo flight software and will run on avionics with thousands of times the RAM and CPU performance of the Apollo avionics.

Not all NASA centers show a growth trend. Figures 4 shows flight software sizes for selected GSFC missions from 1997 through 2009, with NCSL ranging from 28,400 to 149,500. The uncharacteristically large amount of code for the 2001 MAP mission was due to the need for an 'executive' for the Remote Serving Nodes (RSN) processors, programmed in assembly language. The lack of a growth trend matched local expectations in that many of Goddard's science missions are similar in nature, with routine variations in instruments and observation requirements. However, one mission currently in development is expected to have significantly more software, though size estimates are not yet available. Specifically, the Laser Interferometer Space Antenna (LISA) mission comprises three formation-flying spacecraft with distances among them measured to extraordinarily high levels of precision. Flight software will be larger due to a doubling of issues: twice as many control modes, twice as many sensors and actuators, and fault detection on twice as many telemetry points. Section 4.4 examines LISA in more detail as a case study in growth of software complexity.

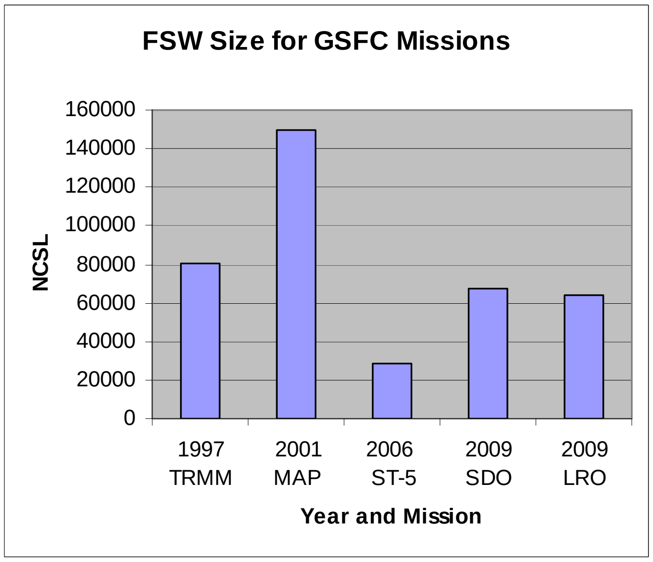


Figure 4. Flight software sizes for GSFC missions.

Figures 5 and 6 show flight software sizes for APL and MSFC, respectively, with no apparent growth trend. Figure 5 shows APL missions from 1995 through 2007, with the larger missions in the range of 100,000 lines of code. Figure 6 shows MSFC projects, mostly in the period from 1997 through 2007, with software sizes in the range of 30,000 to 60,000 lines of code. The MSFC projects include rocket engine controllers and subsystems such as video guidance, furnace facility, and a microgravity science research system, and are, therefore, different in character than the spacecraft flight software reported for GSFC and JPL.

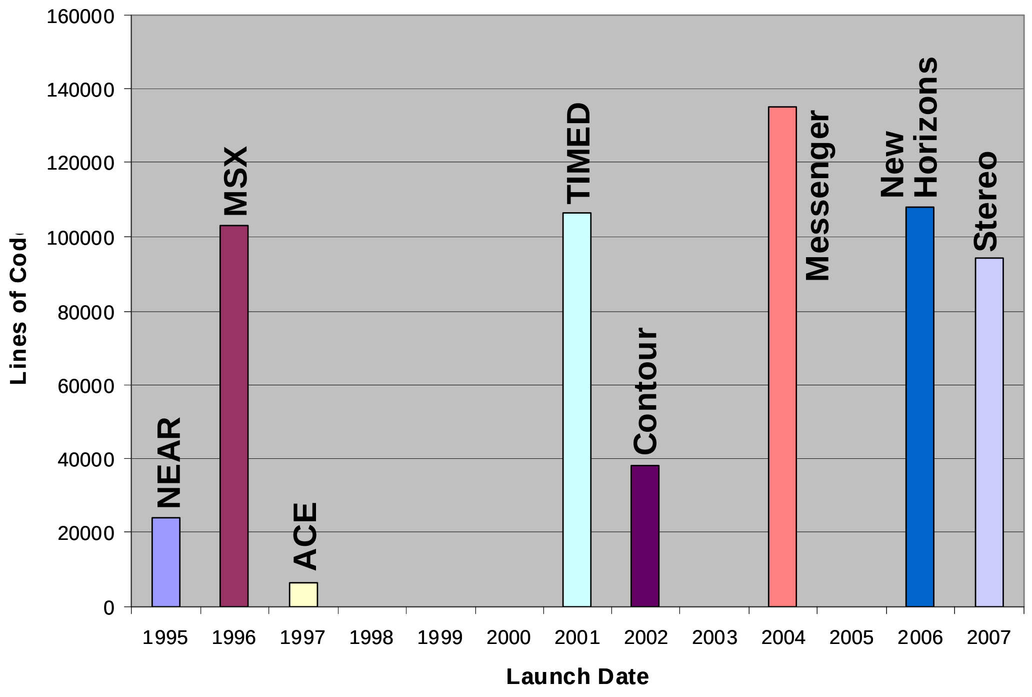


Figure 5. Flight software sizes for APL missions.

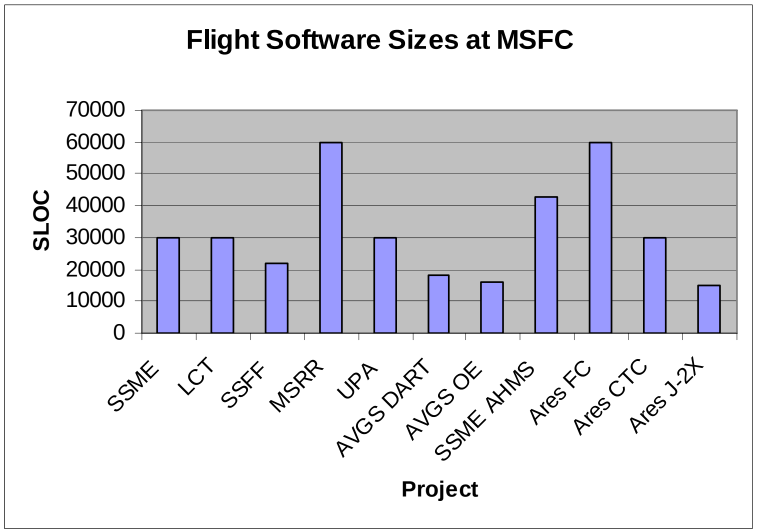


Figure 6. Flight software sizes for MSFC projects.

3.3 Growth of Embedded Systems Software

Spacecraft, aircraft, and automobiles are all examples of embedded real-time systems (ERTS), and all have witnessed major growth in software size and complexity over the past few decades. The closely related aeronautics industry has reported *exponential growth* in both the number of signals to be processed and source lines of code (SLOC) required for civil aircraft over a thirty year period [Gillette 1998]. In a period of forty years, the percent of functionality provided to pilots of military aircraft has risen from 8% in 1960 (F-4 Phantom) to 80% in 2000 (F-22 Raptor), as shown in Figure 7. The F-22A is reported to have 2.5 million lines of code.

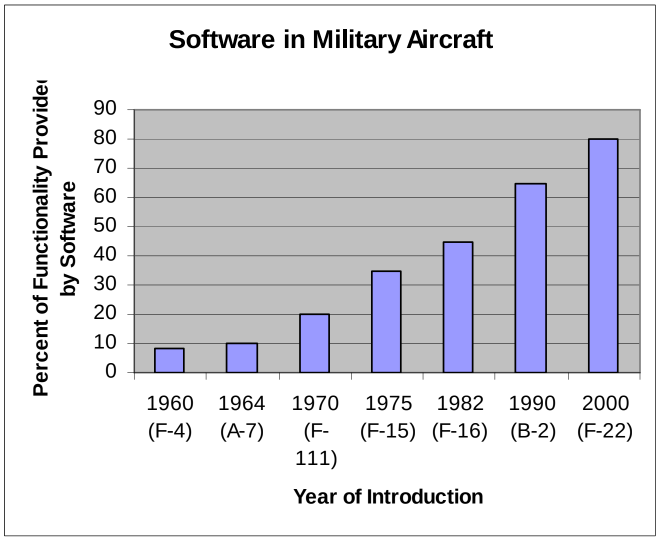


Figure 7. Growth in functionality provided by software to pilots of military aircraft.

The reasons for such growth are easy to understand. The exponentially increasing capability of microprocessors (Moore's Law) along with their decreasing mass and power requirements have made them the tool of choice for adding functionality in the most cost-effective way. As Figure 8 shows, the average General Motors automobile had a million lines of code in 1990 and is projected to have 100 million lines of code by the year 2010; while all of the code is not critical to the car's operation, it is indicative of the continuing trend with embedded systems.

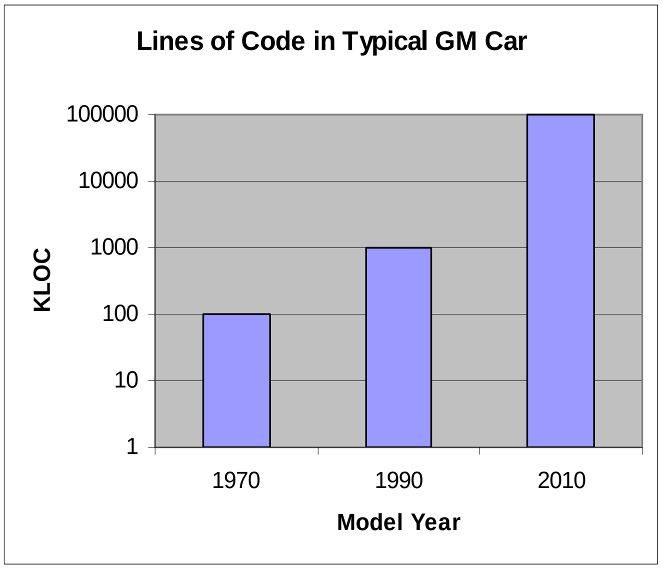


Figure 8. In 1990 the typical GM car had one million lines of code, growing to 100M by 2010.

3.4 Sources of Growth

The main source of growth in flight software is in the requirements of increasingly ambitious missions. Many of these requirements are fundamentally about coordination and control: i.e., coordination of a spacecraft's many devices and control of its activities in real-time. Writing in 1996, author Michael Lyu stated it well: "It is the integrating potential of software that has allowed designers to contemplate more ambitious systems encompassing a broader and more multidisciplinary scope." [Lyu 1996]

Designers situate so much functionality in flight software for three basic reasons. First, the dynamics of a system or its environment often require response times that prohibit Earth-in-the-loop control, either due to round-trip light-time delays or limitations of human ability to react. Examples include rocket ascent control, landing on Mars, and recording short-lived science events. Second, even when light-time delay is not an issue, many spacecraft are out of contact with Earth for periods of hours or days and must, therefore, make decisions without human oversight. Examples in this category include autonomous fault protection. Third, the volume of data generated onboard and limitations on downlink data rate sometimes necessitate onboard data processing and prioritization. The prime historical example here is the onboard data compression used on the Galileo mission to compensate for the loss of its high-gain antenna. More recently, the Autonomous Sciencecraft Experiment flying on EO-1 demonstrates autonomous decision-making based on the content of data collected by instruments [Chien 2005].

To help further understand the growth in functionality assigned to flight software, as well as its continuation into the future, Table 2 shows a long list of functions that are implemented in flight software (in green) and continues with additional functions planned for software (in brown), and others farther in the future (in red). This table is illustrative, not exhaustive. The key message in this story of software growth is that it will continue as long as embedded processors and their software provide the most cost-effective way to satisfy emerging requirements.

Table 2. Functionality implemented in flight software in the past (green), planned (brown), and future (red).

Command sequencing	Guided descent & landing	Parachute deployment
Telemetry collection & formatting	Trajectory & ephemeris propagation	Surface sample acquisition and handling
Attitude and velocity control	Thermal control	Guided atmospheric entry
Aperture & array pointing	Star identification	Tethered system soft landing
Configuration management	Trajectory determination	Interferometer control
Payload management	Maneuver planning	Dynamic resource management
Fault detection & diagnosis	Momentum management	Long distance traversal
Safing & fault recovery	Aerobraking	Landing hazard avoidance
Critical event sequencing	Fine guidance pointing	Model-based reasoning
Profiled pointing and control	Data priority management	Plan repair
Motion compensation	Event-driven sequencing	Guided ascent
Robot arm control	Relay communications	Rendezvous and docking
Data storage management	Science event detection	Formation flying
Data encoding/decoding	Surface hazard avoidance	Opportunistic science

3.5 A Caution about Software Size Metrics

Given the preceding figures of software size measured in source lines of code (SLOC), it is tempting to make inferences that can be misleading. For example, if software size for mission 'A' is twice as large as mission 'B', one might assume that 'A' is twice as complex or contains twice as much functionality as 'B'. Neither of these is necessarily true because SLOC is not a good measure of complexity or functionality. To be sure, there is often some correlation, but there are many factors that complicate such comparisons. For example, there are differences in how lines of code are counted. Some measurements count raw physical lines, some count 'logical' lines, and some exclude comment lines. Some projects count only the code developed in-house while others include code libraries and other third-party software. Some projects count auto-generated software, while others count only the inputs to the code generator. Some projects include firmware, while others do not. Also, lines of code in one programming language might provide inherently more functionality than code in another language. Finally, projects delivering Class A software are subject to more rigorous standards than Class B software and, therefore, require more effort per line of code.

Differences in *what* code is counted and *how* it is counted exist not just across NASA centers but also within centers, making comparisons problematic. Recommendation 2.16 speaks to that issue, concurring with NPR 7150.2 (Software Engineering Requirements), Section 5.3.1 (Software Metrics Report), which requires measures of software progress, functionality, quality, and requirements volatility, while allowing for local decisions about how to measure such attributes.

4 Flight Software Complexity

The preceding section reported on the growth of flight software *size* and linked that growth to increasing ambitions in space missions. While it is certainly true that software *complexity* has grown at the same time, it's important not to equate complexity with size. This section describes software complexity, lists sources of growth, and highlights areas where positive changes can be made.

4.1 Complexity Described

The IEEE Standard Computer Dictionary defines 'complexity' as "the degree to which a system or component has a design or implementation that is difficult to understand and verify" [IEEE 1990]. The phrase "difficult to understand" suggests that complexity is not necessarily measured on an absolute scale; complexity is relative to the observer, meaning that what appears complex to one person might be well understood by another. For example, a software engineer familiar with the systems theory of control would see good structure and organization in a well-designed attitude control system, but another engineer, unfamiliar with the theory, would have much more difficulty understanding the design. More importantly, the second engineer would be much less likely to design a well-structured system without that base of knowledge. Thus, training is an important ingredient in addressing software complexity, as emphasized in recommendations 2.1, 2.6, 2.10, and 2.14.

The definition above also includes the phrase "difficult to verify," which highlights the amount of work needed to verify a system and the sophistication of tools to support that effort. Most modern flight software systems manage so many states and transitions that exhaustive testing is infeasible. Thus, methods for testing complex logic are an important topic in this study, as covered in Section 7.

4.1.1 Essential Functionality versus Incidental Complexity

This study adopted an important distinction about software complexity that helped clarify discussions: namely, the distinction between *essential functionality* and *incidental complexity*[2]. "Essential functionality" comes from requirements and, like well-written requirements, specify the essence of what a system must do without biasing the solution. Essential functionality can be *moved*, such as from software to hardware or from flight to ground, but cannot be *removed*, except by eliminating unnecessary requirements and descoping other requirements. In contrast, "incidental complexity" arises from choices made in building a system, such as choices about avionics, software architecture, design of data structures, programming language, and coding guidelines. Wise choices can reduce incidental complexity and are, therefore, the target of several recommendations.

It's important to recognize that essential functionality must be addressed *somewhere*. A decision to situate some functionality in ground software rather than flight software, or a decision to

[2] In a widely cited paper on software engineering, Fred Brooks made a distinction "essential complexity" and "accidental complexity" [Brooks 1986]. We prefer "essential functionality" for the former term because required functionality is not necessarily "complex", and we prefer "incidental complexity" for the latter term since design choices are deliberate, not "accidental".

address a flight software defect through an operational workaround, does not make the problem go away, though it might lessen the load on the flight software team. Similarly, a "code scrubbing" exercise that generates numerous operational workarounds simply moves complexity from one place (and one team) to another. The location of functionality, and its attendant complexity, should be the subject of thoughtful trade studies rather than urgent decisions made by one team at the expense of another. As noted in Sections 2.3 and 2.1, we found that trade studies are often skipped and that engineers and scientists often have little awareness of how their local decisions affect downstream complexity.

4.1.2 Interdependent Variables

In a study of failures in complex systems, Dietrich Dörner [Dörner 1996], a cognitive psychologist, provides perhaps the best definition of complexity that motivates recommendations regarding the importance of system-level analysis and architectural thinking.

> "*Complexity* is the label we give to the existence of many interdependent variables in a given system. The more variables and the greater their interdependence, the greater that system's complexity. Great complexity places high demands on a planner's capacities to gather information, integrate findings, and design effective actions. The links between the variables oblige us to attend to a great many features simultaneously, and that, concomitantly, makes it impossible for us to undertake only one action in a complex system. … A system of variables is 'interrelated' if an action that affects or is meant to affect one part of the system will also affect other parts of it. Interrelatedness guarantees that an action aimed at one variable will have side effects and long-term repercussions."

Although Dörner's book never uses space missions as examples, his emphasis on interdependent variables is exactly the cause of much complexity in flight software. For example, for some spacecraft: the antenna cannot be pointed at Earth while the camera is pointed at a target; thrusts for attitude control unintentionally affect trajectory; one instrument might cause electromagnetic interference with another instrument; and heaters must be temporarily turned off when thrusting to avoid tripping a power bus. Dörner's book is insightful about the nature of complexity and the difficulties that people have with it. The book is very readable and even comes recommended as a management title by *Business Week* and *Library Journal*.

4.1.3 Tight Coupling and Complex Interactions

The main concern about growth in complexity is the growth in risk. In a classic study of high-risk technologies, Charles Perrow examined numerous accidents in complex systems, including the 1979 Three Mile Island nuclear accident, the 1969 Texas City refinery explosion, the 1974 crash of a DC-10 near Paris, and several close calls in NASA missions, including Apollo 13 [Perrow 1984]. Perrow made the observation that the systems most prone to what he called "system accidents" exhibited *complex interactions* and *tight coupling*, as shown in the upper-right quadrant of Figure 9. Briefly, complex interactions are those of unfamiliar, unplanned, or unexpected sequences, and either not visible or not immediately comprehensible. Information about the state of components or processes is more indirect and inferential, and complex systems have multiple feedback loops that can baffle operators. Tightly coupled systems have more time-dependent processes; they cannot wait or stand by until attended to. Reactions, as in chemical plants, are almost instantaneous and cannot be delayed or extended. The sequences in tightly coupled systems are more invariant and have little slack.

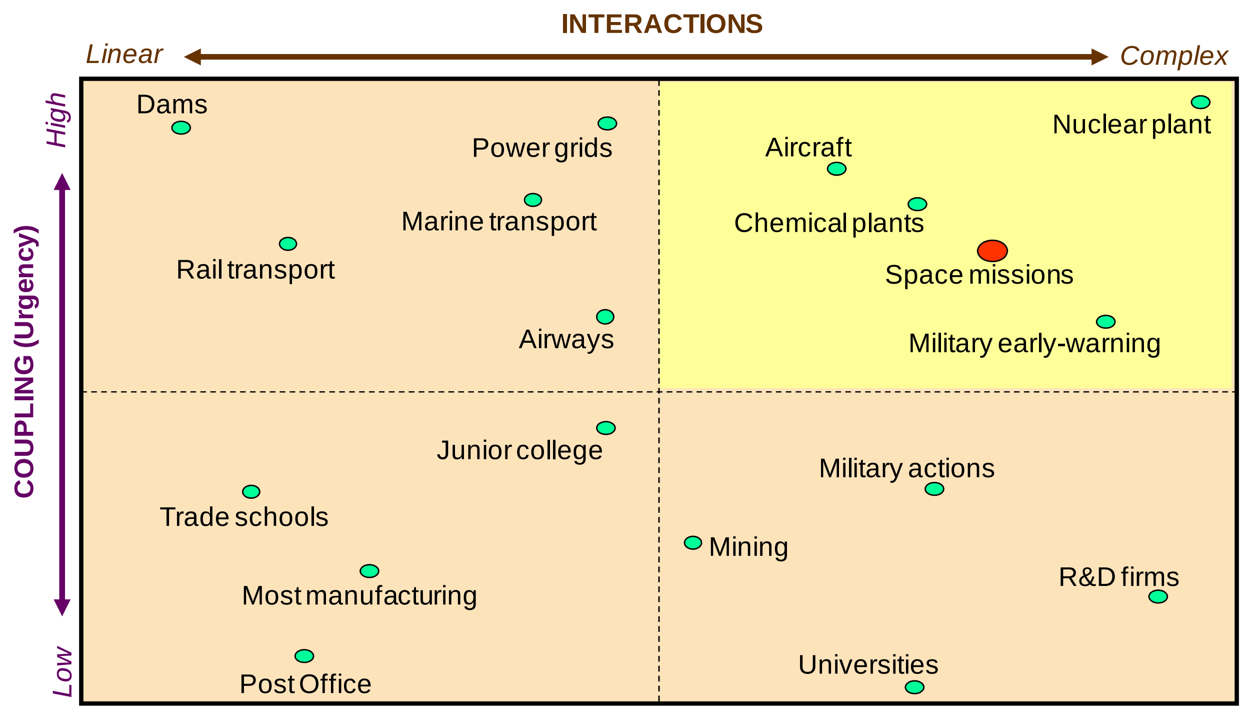


Figure 9. Systems that must manage complex interactions and high coupling are more prone to accidents. Space missions are among these high-risk systems.

4.1.4 Discrete States versus Continuous Functions

In a historical review of the software crisis that provides insight into why software engineering has had such difficulty, Shapiro cites a principal cause.

> "From the 1960s onward, many of the ailments plaguing software could be traced to one principal cause—complexity engendered by software's abstract nature and by the fact that it constitutes a digital (discrete state) system based on mathematical logic rather than an analog system based on continuous functions. This latter characteristic not only increases the complexity of software artifacts *but also severely vitiates the usefulness of traditional engineering techniques oriented toward analog systems* [emphasis added]. Although computer hardware, most notably integrated circuits, also involves great complexity (due to both scale and state factors), this tends to be highly patterned complexity that is much more amenable to the use of automated tools. Software, in contrast, is characterized by what Fred Brooks has labeled 'arbitrary complexity'" [Shapiro 1997].

Shapiro goes on to explain that software complexity is really a multifaceted subject. Note, however, that the italicized excerpt above offers at least a partial explanation to the other engineering disciplines who wonder why software engineering isn't more like them. The mathematics that so well describe physics and so well support other engineering disciplines do not apply to the discrete logic that comprises so much of flight software. As Brooks points out, physics deals with terribly complex objects, but the physicist labors on in a firm faith that there are unifying principles to be found. However, no such faith comforts the software engineer. [Brooks 1986]

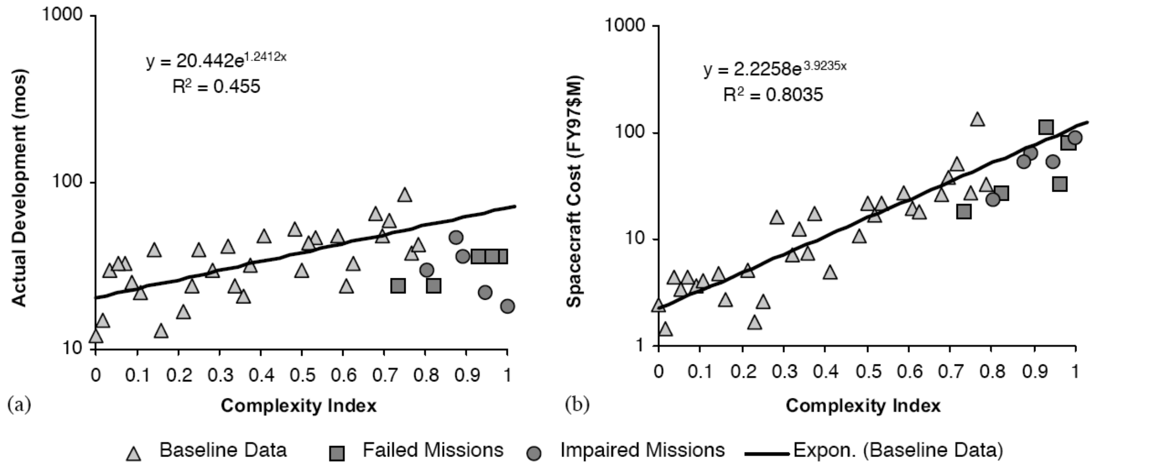
4.2 Complexity through the Lifecycle

Although the topic of this study is "flight software complexity," the study's scope extended into upstream and downstream activities, as noted earlier in Section 1.2. Incidental complexity can originate anywhere, from requirements to operations, and in a variety of forms: poorly written, misleading, or unnecessary requirements; superficial analysis that ignores important functionality until late in development; inadequate software architecture; lack of attention to software testability; language features that complicate verification; and software descope decisions that complicate operations.

4.3 Views on Complexity

The overall impression one is left with on reviewing flight software experiences across NASA Centers is that when new technologies, capabilities, and requirements are introduced in NASA missions, the software needed to support these new technologies, capabilities, and requirements can grow dramatically in size. The growth in software size will be accompanied by a corresponding increase in the effort/cost required to implement it, often with an out-of-proportion increase in the testing effort needed to retire the increase in perceived risk.

Given that new technologies, capabilities, and requirements are not only inevitable but are, in fact, both mandatory to support the agency's Exploration and Science goals (and are, in fact, enthusiastically embraced by NASA's engineering staff in order to advance their state-of-the-art), it is incumbent on us to find effective ways to identify and characterize software complexity and to find better ways to minimize the impact on cost and schedule of necessary increases in software complexity, while reducing the growth of unnecessary complexity. The potential impact to Mission cost and success as a consequence of high mission complexity can clearly be seen in the Figure 10 (originally generated by Bearden).



Cost and schedule as a function of complexity. (a) Schedule as function of complexity, (b) Spacecraft cost as function of complexity.

Figure 10: Impact of Complexity on Mission Cost and Success.

In the subsections that follow we will examine in more detail what FSW complexity is, what are its signatures, what are its drivers, and how complexity's impact can be managed.

4.3.1 Characterizing Flight Software Complexity

Characterizing FSW complexity is not a straightforward process. Although increasing the size of a system can lead to a higher level of complexity, the complexity can be managed through the use of modular design, hierarchical structure and data/control flow, and well-documented “need-to-know” interfaces. By contrast, a smaller system can be more difficult to understand and maintain if it consists of a few very large modules, moves data & control through circuitous paths, and passes the whole kitchen sink from module to module. In a practical sense, FSW complexity is a measure of:

- How difficult it is for a programmer to implement the requirements the code must satisfy?
- How difficult it is for a tester to verify that the code satisfies the requirements and operates in an error-free fashion?
- How difficult it is for a lead developer to manage the development of the FSW within cost and schedule?
- How difficult it is for a FSW maintenance programmer to understand the original programmer’s work if the software must be modified after launch?
- How difficult it is for a new programmer on a later mission to adapt the original FSW as heritage for the new mission?
- From a risk standpoint, how difficult it is to predict the FSW’s behavior, which in turn can drive much more extensive testing and more operational “hand-holding” along with their associated higher labor costs?

Another aspect of FSW complexity is the way it can couple into other subsystems. One can describe complexity as “Pareto-optimal,” meaning that once the easy, straightforward improvements have been made, for a given cost and schedule you can’t improve one difficulty without making one or more difficulties worse. It is a zero-sum game, where the various complexities of the FSW and the rest of the spacecraft (and, for that matter, the ground system and operations) cannot be addressed in isolation. They must be managed as a whole and viewed through a system engineering lens; otherwise, an improvement in one area could easily increase the overall complexity and, thus, the overall cost and risk of the entire mission.

Assuming best programming and testing practices are utilized, what will drive a software system’s complexity will be the complexity of the underlying problem the software must solve (referred to as essential complexity) and the influence of external sources that might add unnecessary complexity to the implementation (referred to as extraneous complexity). Some sources of complexity are inevitable, especially in a relative sense. For example, human-rated missions FSW functionality will always be implemented in a more complex fashion than the equivalent robotic mission FSW functionality because of human-rated missions’ far higher safety demands, particularly in the area of redundancy support. New FSW functionality is usually accompanied by new Fault Detection, Diagnostics, and Recovery (FDDR) checks, and FDDR testing costs rarely increase only linearly when new checks are added due to the potentially geometric expansion of failure paths. An explosion in test costs can develop when FSW is designed with multi-threads of execution with coupled timing constraints. The need to protect flight systems from hacking and other security concerns adds further FSW complexity, especially where commercial network protocols introduce vulnerabilities that must be "patched" with custom code.

Factors that can measure a FSW system's essential complexity include:

- How many functions the FSW must execute and monitor?
- How many hardware components the FSW must monitor, command, control, and query for information?
- How many connections (both hardware and software) between components the FSW must monitor and manage?
- How many control modes must be managed and executed?
- How many software modules must be implemented in order to satisfy the FSW's requirements?
- How much coupling there is between software modules?
- How intricate/convoluted the code is within a module (assuming best programming practices, this is a measure of the complexity of an associated requirement or algorithm itself)?
- How many tests must be created and executed in order to verify that the FSW has satisfied its requirements and, in fact, whether it is even possible given limited time and cost to verify satisfaction of those requirements under all likely scenarios?
- How "state-of-the-art" the requirement is (reflected in how demanding performance and accuracy requirements are relative to contemporary, heritage systems)?

Note that the Flight Operations Team (FOT) effort required to operate the FSW subsystem does not appear on the list. This omission is not meant to imply that a very complex FSW subsystem might not also result in complex operations, thereby resulting in the worst of both worlds: i.e., very expensive FSW development costs and very expensive operations costs. However, it also might be the case that the operations costs are high because the FSW lacks autonomous features and, as a result, requires that the FOT conduct a myriad of potentially risky routine operational procedures (ROPs). By contrast, low operational costs might have been enabled via implementation of a highly autonomous, and very complex, FSW subsystem.

Therefore, in an era where steadily increasing mission capabilities are not only to be expected but welcomed by the engineering teams that will make them a reality, it is clear that if FSW costs are to be controlled without damaging reliability we must become more proficient at identifying the sources of FSW complexity, managing that complexity, and, where possible, eliminating incidental complexity. In the next section we will examine the sources of complexity and suggest means to minimize it.

4.3.2 Identifying and Minimizing Incidental FSW Complexity

The sources of FSW complexity, both essential and incidental, originate with the FSW requirements. Particularly for state-of-the-art missions, much of the requirements-originated complexity will be critically needed to satisfy the mission's goals. One of the keys to limiting unnecessary complexity is to write a thorough rationale along with each FSW requirement.

Requirements rationales are specifically called-out in an appendix of “best typical practices” in NPR 7123 (Software Engineering Requirements). The rationale provides a means to challenge “gold-plated” requirements and “nice-to-haves,” as well as requirements that have been included because “that’s the way we’ve always done things.” The rationale also documents the requirement’s justification, so that if circumstances change downstream in the mission lifecycle an “obsolete” FSW requirement can be identified and deleted. For heritage missions, the rationale can provide an excellent means to document the results of trade studies, both from the standpoint of justifying the requirement’s inclusion in the current mission where a consistency exists relative to the previous mission and from the standpoint of justifying the requirement’s deletion where circumstances have changed. Lastly, because a strong accompanying set of rationales can help stabilize FSW requirements early in the mission lifecycle, much of the requirements volatility that can cause large increases in FSW costs (i.e., “requirements creep”) downstream can be avoided.

Often flight software becomes a “sponge” for complexity. Sound engineering trade studies are an excellent opportunity to apportion requirements and their associated functionality optimally from the standpoint of overall mission costs. But to provide these cost benefits, the trades must be done early and must involve all impacted subsystems (including operations personnel). The trades also require strong Project system engineering support to enforce adoption of the most cost-effective options. Hardware avionics architecture decisions can have a particularly strong impact on FSW complexity and its associated costs. A decision to save money by utilizing a “cheap” processor can result in far larger cost increases for the FSW that will run on the processor. In other cases, the engineering trades might be driven by science requirements. For example, a science requirement to guarantee reception of 99% of the science data might produce spacecraft designs with highly cross-strapped hardware with many distributed and redundant boxes. The associated large increases in hardware costs might be accompanied by equally large increases in FSW costs, especially with regards to FSW testing costs. A properly designed trade requiring science customers to justify their data efficiency needs can provide the “push-back” needed to bring down both hardware and FSW complexity yielding large corresponding cost savings in both areas.

On the other hand, some trades might demonstrate that increased FSW complexity can yield a reduction in overall mission costs. This often proves to be the case where adding cost via a new onboard, autonomous FSW function produces much larger savings in operational costs, especially when those operational savings are factored in over many years of mission life. Furthermore, an “operational workaround” that must be performed on a routine basis can add much more risk to the mission, along with the excessive operations costs compared with an autonomous FSW function solution.

Finally, some incidental complexity might be introduced in an effort to reduce costs and risks by utilizing well-tested Commercial Off the Shelf (COTS) products or heritage FSW that are accompanied by extra features that exceed requirements. Additional custom code might be needed to “turn-off” those features and additional testing might be needed to verify that those extra features do not manifest themselves on-orbit.

4.4 A Case Study: Laser Interferometer Space Antenna (LISA)

Compared to previous GSFC missions, the Laser Interferometer Space Antenna (LISA) mission represents a significant step-up in flight software (FSW) complexity. Lines between spacecraft and payload become blurred as the LISA science instrument is created via laser links connecting three spacecraft forming approximately an equilateral triangle of side length 5 million kilometers. The science measurement is formed by measuring to extraordinarily high levels of precision the distances separating the three spacecraft via the exchange of laser light. The individual spacecraft maintain their positions in inertial space by following the movements of the proof masses along the sensitive axes of their gravity sensors.

From the standpoint of Bus FSW providing spacecraft attitude and position control, the number of sensors and actuators that must be interrogated and commanded is at least twice the number associated with a more traditional mission. Similarly, the number of control modes is double that of a typical astrophysics mission, as are the number of parameters solved for by the state estimator. These factors also suggest the Bus FSW will need to perform fault detection on at least twice the normal number of telemetry points, with a correspondingly large number of fault correction responses. So it’s clear simply by counting pieces of functionality, LISA’s Bus FSW will be significantly more elaborate than most previous GSFC missions. For the most part, the nature of the processing to be performed in support of these functions will be comparable to what is typically done. Output from sensors & actuators will be converted from raw data to engineering units by polynomial functions often as simple as linear. The command interface to the sensors & actuators also should be fairly straightforward. The control laws implemented within the control modes will be the usual Proportional-Integral-Derivative (PID) ones, although the number of inputs and outputs will be larger for both the control laws and estimators. As with most projects, the specification of fault detection & correction (FDC) currently is less mature than for other onboard functionality. However, the basic logic of the processing is well understood: i.e., there will be distributed fault checking based on limits and tolerances and a table-driven centralized fault correction. There will be counters identifying repeated violations of the limits/tolerances; if a counter limit is tripped, a flag will be set indicating an anomaly has occurred. Probably, most individual detected anomalies will be dealt with on an individual rather than coupled basis, but the details of the correction process cannot be known until all the possible failure mechanisms and scenarios have been thoroughly analyzed; hence, the comment regarding the lower level of maturity.

Furthermore, many of the LISA Bus FSW functions support new technologies with which there has not been a great deal of in-flight experience. LISA will use Micro-Newton thrusters to make the very small corrections to spacecraft attitude and position required to enable the spacecrafts’ to follow their proof masses motions in Drag Free control. Both the Micro-Newton thrusters and gravity sensors are new technology whose concepts will be flight-tested by the LISA Pathfinder mission. Similarly, the control laws enabling Drag Free control are new, developed at GSFC to support LISA Pathfinder and LISA. Although laser metrology is not new, the precision at which LISA will be performing its metrology is quite unique. Over the measurement baseline of 5 million kilometers, LISA will be seeking to measure distance changes on the order of the size of an atom. These extraordinarily demanding precision requirements on the science measurement translate into unprecedented accuracy requirements for the control laws that enable the spacecraft to perform its Drag Free tracking of the proof masses. Therefore, the complexity level and accuracy demands of many LISA functions will be a step up from their counterparts on previous GSFC missions.

The high precision/accuracy requirements suggest a spill-over into FSW testing. Not only must algorithms be developed and implemented that enable these severe requirements to be met but, in addition, testing methodologies must be developed that will verify that the requirements have been met. Traditional mission-pointing accuracy requirements are of order arcsecond. Even for the state-of-the art Hubble Space Telescope (HST), milliarcsecond pointing stability is the standard. For LISA, mispointings on the order of milliarcseconds would be enough to disrupt the laser links; an order of magnitude worse might suffice to break the links entirely. Performance of LISA science requires pointing stability to a tenth of a milliarcsecond or better, an order of magnitude more demanding than that for HST. FSW validation will, therefore, probably need to be able to see "errors" to a couple orders of magnitude better: i.e., micro-arcsecond level, which might necessitate the utilization of special software tools capable of identifying variations from "truth" at this level. In other words, simple visual examination of a graphical plot of position and orientation angles vs. time will probably not suffice in determining whether a control law is meeting LISA's stringent requirements. The data might, instead, need to be processed statistically to determine goodness-of-fit, or to determine if the apparent noise in the errors is truly random. These additional efforts, should they be required, probably will appear largely in the form of extra hours of analytical support for FSW build testing and analyst-created statistical/graphical tools.

Strictly speaking, the LISA mission does not require implementing an onboard capability for formation flying between spacecraft in a constellation. Primarily, what a LISA spacecraft will do is follow its own individual proof masses in Drag Free control independent of the other two spacecraft. This formation flying between a LISA spacecraft and its proof masses must be controlled to within a nanometer or better accuracy. The triangle formation between the three LISA spacecraft is largely maintained by orbit selection, and small variations in triangle vertex angles over time are accommodated via Telescope Articulators that permit modifying the angles between the lasers and Point-ahead Actuators that place the laser light where the target spacecraft will be (i.e., compensate for light travel time). These angle changes are orbit dependent and largely predictable, so a portion of the angle change can be ground calculated and a table can be uplinked to the Bus FSW for use in open-loop commanding. The remaining angle change will be measured via feedback between spacecraft and compensated for via closed-loop commanding.

In other aspects, the LISA Bus FSW complexity is comparable to traditional GSFC missions. Specifically, onboard autonomy has been restricted to functionality required to maintain health & safety and to provide an acceptable level of science efficiency. Currently, there will be no onboard Time Delay Interferometry (TDI) processing of science data, although some diagnostic-oriented processing might be performed to help assess the stability of the laser links. Processing of the science data will be purely a ground responsibility. There will be no onboard autonomous overall management of the LISA Constellation. That also will be a ground responsibility. Had these functions been implemented in FSW, the size and complexity of the LISA Bus FSW would have been greatly increased. GSFC also has had some previous experience developing Bus FSW for an in-house constellation mission. ST-5, which comprised a constellation of three spacecraft, has been launched and is being operated successfully. So on LISA, GSFC will not have to figure out for the first time how to perform configuration management on the FSW for more than one spacecraft, each with its own unique identification and independent commanding needs.

In the future, many in the science community would like to build on the LISA constellation experience to develop new constellation missions. Astrophysicists have conceptualized many missions that would autonomously formation-fly many individual lensing spacecraft to create a virtual mirror whose light would be integrated at a hub spacecraft. Many missions have been conceptualized by Earth scientists that would autonomously coordinate the data collection efforts of many “smart” instruments distributed on multiple spacecraft to improve data collection efficiency and provide enhanced target of opportunity (TOO) response. As always, budget limitations will dictate which of these exciting mission concepts will become real missions and which will remain on the drawing board.

4.5 Software Complexity Metrics

In order to present a picture of growth of flight software within NASA, Section 3.2 displayed several software *size* measurements in terms of source lines of code. This kind of metric is objective and widely used in software engineering, but is not considered a good measure of complexity. In fact, there is no single metric for complexity, but rather several metrics for different kinds of complexity. Table 3 lists six different complexity measures. The Halstead measures, introduced in 1977, are among the oldest measures of program complexity. The measures are based on four numbers derived directly from a program’s source code, none of which is the number of source lines of code. Some programmers in NASA use commercial tools to measure their code complexity, mainly as a personal practice, so that they re-factor any code whose measurement exceeds some level of complexity. We did not discover any projects that mandate such metrics.

Table 3. Software complexity metrics

Complexity Metric	Primary Measure of ...
Cyclomatic complexity (McCabe)	Soundness and confidence; measures the number of linearly-independent paths through a program module; strong indicator of testing effort
Halstead complexity measures	Algorithmic complexity, measured by counting operators and operands; a measure of maintainability
Henry and Kafura metrics	Coupling between modules (parameters, global variables, calls)
Bowles metrics	Module and system complexity; coupling via parameters and global variables
Troy and Zweben metrics	Modularity or coupling; complexity of structure (maximum depth of structure chart); calls-to and called-by
Ligier metrics	Modularity of the structure chart

In this study we did not devote any resources to the study of complexity metrics because the findings did not lead us in this direction.

5 Fault Management

5.1 NASA Planetary Spacecraft Fault Management Workshop

This study gave special attention to fault protection software because it accounts for a large and complex portion of flight software. Fault protection software must, among other things, monitor for misbehavior, pay attention to limited resources, coordinate responses, and perform goal-driven behaviors. Even if fault protection was no more complex than routine subsystem functions elsewhere in the flight system, it would still be a lightning rod for attention during the months before launch because, as a cross-subsystem function, it requires a higher level of maturity from the flight system and test infrastructure than other functions of the system. Recent missions have experienced technical issues late in the development and test cycle, resulting in both project schedule delays and cost overruns, and raising questions about traditional approaches.

In recognition of these problems, NASA held a planetary spacecraft fault management workshop in April 2008 that brought together ~100 experts from NASA, Defense, Industry, and Academia [NASA 2008a]. The purpose of the three-day workshop was to review recent mission experiences, understand common issues, identify lessons learned and best practices, and explore emerging technologies. The key findings of the workshop are documented in a whitepaper [Fesq 2008]. The NASA workshop was a major focus of attention in this study with respect to two work items—Special Interest 3, Approaches 1 and 2—and gave rise to three recommendations about fault protection, as described in Sections 2.12, 2.13, and 2.14. These recommendations are described in a thorough 40-page report prepared by Kevin Barltrop in Appendix F. The report is recommended reading for anyone wanting a better understanding of fault protection and its software complexity. To quote from the report's conclusions on the technical state of the practice:

> "The main lesson from the survey of the technical state of the practice is that practitioners build on past work without much critical examination of the assumptions and principles that underlie the task. Without that critical examination and understanding, design solutions poorly fit to new applications perpetuate, needlessly increasing the cost and risk of missions. All institutions moving into deep space missions have encountered this problem, with similar consequences."

In effect, this observation is yet another symptom of the need for more up-front architecting (recommendation 2.4) and the need to nurture architectural thinkers (recommendation 2.6).

The NASA workshop included four invited talks that offered expert views. Gentry Lee, one of the leading system engineers at JPL, made three points: (1) missions of the future need to have their systems engineering deeply wrapped around fault management; (2) fault management will be the biggest limitation on what we can do in future missions; and (3) a revolution is necessary, and now is the time, because the current fault management approaches aren't adequate. Professor David Garlan of CMU, a national expert on software architecture, spoke about how to improve *system* quality through *software* architecture. His objective was not to recommend any particular architecture but rather to emphasize attention to software architecture in order to avoid problems later in implementation, integration, and verification (in perfect agreement with recommendation 2.4). Dr. John Rushby of SRI, an expert in the design and assurance of critical systems, made three main points: (1) goal-based (claims-based) assurance cases provide a promising framework for improving cost effectiveness of verification and validation; (2) model-based design opens the door to automated reachability analysis; and (3) strong interfaces promote compositional assurance. Both Dr. Garlan's and Dr. Rushby's points apply broadly to flight software, not just fault management.

The final invited talk was given by Professor Brian Williams of MIT, formerly of NASA Ames, who spoke on "Model-Based Monitoring of Complex Systems," making the point that NASA *has* better fault protection technology but is not using it. Dr. Williams described fault protection technology that he and others developed and flew as part of the Remote Agent Experiment on Deep Space 1 in 1999. The technology showed that, given models of behavior of a system's components, model-based reasoning can determine the most probable faults and can even automatically synthesize a series of steps to respond appropriately. Compared to traditional methods, the approach is much more disciplined and grounded in theory. The MIR technology (Mode Identification and Reconfiguration) offers a good example of how model-based software can *simplify* fault protection engineering and improve system robustness. As to why this kind of technology has not been adopted in NASA's flight systems, the most likely explanation is that it is viewed as "risky," not because of any weakness (quite the opposite, in fact,) but because it represents a paradigm shift, with few engineers trained in that type of modeling.

5.2 Integrating Fault Protection with Nominal Control System

Special Interest 3, Approach 3 directed this study to investigate the feasibility of eliminating or at least minimizing fault protection software as a separate entity. The issue here is that the traditional approach treats fault protection software as an add-on to a sequencing system designed for nominal execution. Engineers at some centers speak of fault protection software as "autonomy software" because, in contrast to the sequencing system, it is designed to react to observations autonomously and initiate responses to recover from faults.

The problem with the traditional add-on approach is that it introduces unnecessary complexity. A traditional design contains *two* top-level control mechanisms: an open-loop command sequencer for nominal activities and closed-loop event-driven logic for fault protection; however, both need the same success criteria, and nominal activities increasingly need closed-loop control for *in situ* operations. The existence of two top-level control mechanisms creates a difficult-to-engineer interface that must ensure that each one gets control when appropriate, and that they don't interfere with each other.

Is there really a need for an independent, global fault protection system? Can a similar effect be achieved by deeply integrating fault protection into the nominal system? These questions were asked in Special Interest 3, Approach 3 and are answered in Appendix G, in a paper by Robert Rasmussen that examines exactly these questions [Rasmussen 2008]. The paper goes back to fundamentals—the systems theory of control—and explains that *all* control decisions, whether for nominal or fault situations, should be based on three things: state knowledge, models of behavior, and desired state (goals). A single, top-level control mechanism suffices, integrating fault protection within the nominal control system and simplifying its engineering and testing. The paper is very readable and highly instructive about how systems engineering and software architecture go hand-in-hand.

6 Thinking Outside the Box

Several recommendations in this study emphasize the importance of software architecture but provide no examples of the kind of architectural thinking needed to arrive at a good architecture. Some engineers treat the topic of architecture minimally in the sense of focusing on *what* gets built: i.e., naming the major components, describing interfaces among them, and specifying details of assembly and integration. However, in the larger sense, architecture is about *why* it gets built the way it does. Architecture is about: identifying properties of interest beyond just the requirements, and from all essential points of view; defining workable abstractions and other patterns of design that give the design its shape and reflect fundamental concepts of the domain; guiding the design and maintaining principles throughout the development lifecycle; and building on a body of experience and refining concepts as necessary.

How do we know if an architecture is good? To be accurate, there is no such thing as an inherently good or bad architecture. An architecture is either more or less fit for some stated purpose, and should be evaluated with respect to a prioritized list of *quality attributes* [Clements 2002]. Quality attributes are sometimes referred to as the "ilities" of an architecture, such as availability, modifiability, testability, usability, scalability, and portability. In addition to these qualities of a system, there are other qualities directly related to architecture itself. In particular, *conceptual integrity* is the underlying theme or vision that unifies the design of the system at all levels, meaning that an architecture should do similar things in similar ways. Fred Brooks writes emphatically that a system's conceptual integrity is of overriding importance and that systems without it fail [Brooks 1975, Bass 2003].

In a position paper titled "Thinking Outside the Box: to Reduce Complexity in NASA Flight Software," attached as Appendix H of this report, author Robert Rasmussen examines the nature of complexity, assesses current practices, motivates the need for architecture, and sets forth key architectural concepts that apply broadly to NASA flight systems. Importantly, these concepts provide great conceptual integrity because they are based on an underlying theme of transparency of control. Although this position paper is long (~60 pages), it is targeted for readers with a general engineering background, and it lays out key ideas in an easy-to-understand way. It is perhaps the most important deliverable from this study because it shows a way out of "the box" that has confined our thinking in software architecture and shows how to reduce incidental complexity.

As Rasmussen states in the introduction, the approach presented is "outside the box" only in the original sense of that phrase: as a solution that is obvious once it is described. Perhaps the more important question is: "How did we develop limiting preconceptions that constrained our thinking to be inside an unnecessary 'box'?" As a partial explanation, and as an enticement to read the full paper, the introduction says this:

> "This seems to be due largely to inertia in our risk-averse culture, which has nurtured old expedients in the name of conservatism. For example, we find little abstraction at the system level to deal with common system level issues, even though system level interactions are one of the knottiest elements of the complexity problem. Instead, we see only modest refinements of ideas from the early years of spacecraft software development, when that was all that would fit into tiny computing capacity. Trying something markedly different is no longer out

of reach, yet we hold tenaciously to old ideas that have long since proven their limitations, mainly because they are the devils we know and have mastered. Thus, what was once a tentative response to severe computing limitations and nascent capabilities in embedded software has resolved itself over time into dis-innovative skepticism of the very methods required to address growing software complexity."

7 Prevention, Detection, and Containment

As the findings of this study have shown, some of the problems that appear in flight software are symptoms of errors or inadequacies in *earlier* activities, prior to software-intensive activities such as software design, coding, integration, and testing. However, these software-intensive activities are themselves the source of many problems. Figure 11 shows that many defects are "inserted" during design and coding. Although many defects are removed during design and coding and testing, some escape detection and, thus, remain in the operational software. Many of these "residual defects" will never cause a failure unless a specific combination of conditions occurs. The odds of that happening are reduced by a common practice known as "test as you fly and fly as you test," meaning that the flight system will be tested according to its intended operation and that actual operation will be constrained to stay within the envelope of what has been tested. This practice has surely avoided many failures, but it's also a symptom of a problem: namely, that traditional scenario-based testing doesn't generate much confidence in the trustworthiness of a system in untested situations. It is also important to realize that testing can only show the presence of defects, not their absence, and it never reduces complexity.

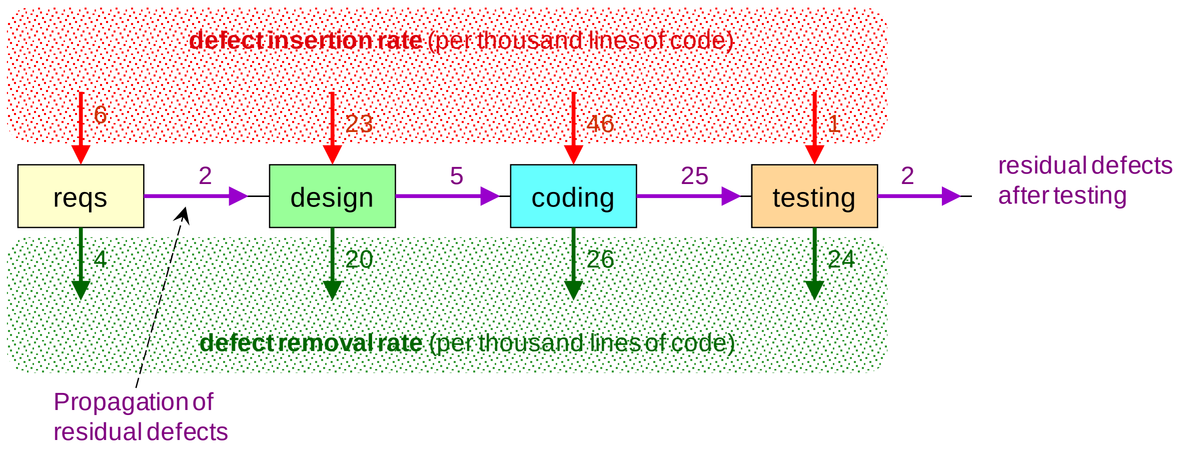

defect insertion rate (per thousand lines of code)
6
23
46
1
reqs
2
design
5
coding
25
testing
2
residual defects after testing
4
20
26
24
defect removal rate (per thousand lines of code)
Propagation of residual defects

Figure 11. Defects are inserted and removed at different rates in each part of the development lifecycle. The difference between insertion and removal rates determines defect propagation rate. This example shows that 2 defects per thousand lines of code remain after testing. (S.G. Eick, C.R. Loader et al., Estimating software fault content before coding, Proc. 15th Int. Conf. on Software Eng., Melbourne, Australia, 1992, pp. 59-65)

Of course, some of the residual defects *will* cause a failure given a certain combination of inputs or ordering of events. Although many of the failures will be minor, as Gerard Holzmann points out in "Software Complexity" (Appendix D), minor failures sometimes combine to create a mission-ending failure, as in the case of Mars Global Surveyor. Viewed from this perspective, we should do everything possible to avoid even minor failures. Within the domain of software engineering, there are three forms of control that contribute to dependable software systems: prevention, detection, and containment. *Prevention* includes strong requirements capture and analysis, architectural principles, model-based engineering, coding standards, and code-structuring rules. *Detection* includes not only traditional testing but also compliance checkers, static source code analyzers, logic model checking, and test randomization techniques. *Containment* includes hierarchical backup methods, memory protection techniques, and broad use of software safety margins.

It’s important to note that all three forms of control—prevention, detection, and containment—can target complexity as well as defects. Architectural guidelines, coding standards, and code complexity measures all help reduce software complexity. For example, a common software measure termed “cyclomatic complexity” is considered a good testability metric because it determines the number of test cases needed to test all paths in a software module. Developers who measure cyclomatic complexity and see a high value can often re-factor the code, thereby reducing complexity and simplifying testing. Also, hierarchical backup methods reduce dependence on the most complex software components by providing simpler, fallback capabilities. Appendix E discusses these approaches in three reports, emphasizing practical measures that can be more broadly adopted.

8 Literature Survey

Special Interest 2, Approach 2 directed this study to conduct a survey of strategies proposed in the literature to manage the complexity of embedded real-time systems. In fact, so much has been written in the general literature about software challenges that the job was more about selection than searching. One particularly relevant source of information came from the Department of Defense (DoD) and the National Defense Industrial Association (NDIA), and is perhaps the most valuable source to this study for three reasons: (1) it deals primarily with embedded real-time systems (weapon systems), so it is directly relevant to flight software; (2) the findings and recommendations are based on software-intensive systems that are generally larger than NASA flight systems and, therefore, portend the future; and (3) the results are very recent (2007), which is important in the rapidly evolving world of software engineering.

In 2007, a relatively new organization in DoD—the Software Engineering and System Assurance Deputy Directorate—reported their findings on software issues based on approximately 40 program reviews in the preceding 2½ years [Baldwin 2007]. They found several software systemic issues that were significant contributors to poor program execution:

- Software requirements not well defined, traceable, testable.
- Immature architectures, COTS integration, interoperability.
- Software development processes not institutionalized, planning documents missing or incomplete, reuse strategies inconsistent.
- Software test/evaluation lacking rigor and breadth.
- Unrealistic schedules (compressed, overlapping).
- Lessons learned not incorporated into successive builds.
- Software risks/metrics not well defined, managed.

Later, in partnership with the NDIA, they identified the seven top software issues that follow, drawn from a perspective of acquisition and oversight:

1. The impact of system requirements upon software is not consistently quantified and managed.
2. Fundamental system engineering decisions are made without full participation of software engineers.
3. Software life cycle planning and management by acquirers and suppliers is ineffective.
4. The quantity and quality of software engineering expertise is insufficient for dealing with the scale of complexity of modern systems.
5. Software verification techniques are costly and ineffective for dealing with the scale of complexity of modern systems.
6. There is a failure to ensure correct, predictable, safe, secure execution of complex software in distributed environments.

7. Inadequate attention is given to total life cycle issues for COTS/NDI impacts on life cycle cost and risk.

In partnership with the NDIA, they made seven corresponding top software recommendations:

1. Enforce effective software requirements development and management practices.
2. Institutionalize the integration and participation of software engineering in all system activities.
3. Establish a culture of quantitative planning and management.
4. Collaborate on approaches to attract, develop, and retain qualified talent to meet current and future needs in government and industry.
5. Develop guidance and training to improve effectiveness in ensuring product quality across the life cycle.
6. Develop approaches, standards, and tools addressing system assurance issues throughout the acquisition life cycle and supply chain.
7. Improve and expand guidelines for addressing total life cycle COTS/NDI issues.

The DoD/NDIA recommendations have led to several DoD initiatives that are being pursued. One category of issues known as “human capital” got much more attention in the DoD/NDIA study than in our NASA study. ‘Human capital’ encompasses workforce and skill sets, where they report a lack of system and software engineers in key leadership positions, a shortage of highly experienced software managers and architects, and fewer young people going into systems and software engineering.

Another valuable perspective on software complexity appeared in a 1997 journal article that examined the preceding 30 years of thought in the software engineering community [Shapiro 1997]. This well-written article provides an even-handed look at the software community’s quest for “dramatic singular solutions” for getting a handle on complexity. The article traces the history of ideas about and approaches to *modularity* in software, including structured programming, information hiding, coupling and cohesion, programming-in-the large versus programming-in-the-small, data abstraction, object-oriented programming, and others. In the end, the author concluded that the software community needs “a more pluralistic mindset revolving around synthesis and trade-offs” rather than a singular approach to solve all problems. The recommendations presented earlier in Section 2 certainly reflect this sentiment since no single idea is touted as the key to salvation. Rather, the ideas span the engineering lifecycle and apply to project managers and systems engineers as well as software engineers.

Appendix A provides the literature survey. Readers should understand that the survey was necessarily selective and was intended, in some degree, to identify any key topics that might have received insufficient attention in the rest of the study.

9 Topics for Future Study

Any study such as this one that asks "What are the real problems and how should we address them?" will attract many different opinions. Many readers from the software community will not see their favorite topic or hot-button issue discussed in this report for the simple reason that the study was directed to examine the specific issues and questions listed in Section 1.2. Although the study was not strictly confined to those issues, they provided the main focus for framing the findings and recommendations. Having said that, there are several worthy topics that could be considered in future studies, some of which are as follows:

- *Model-Based Systems Engineering (MBSE)*. MBSE is gaining acceptance as a best practice for well-founded systems engineering. MBSE differs from traditional document-centric systems engineering in that it emphasizes the role of models in capturing requirements and design decisions. That emphasis, in turn, encourages software designs that more directly represent and use such models.
- *Reference architecture*. Recommendation 2.9 advocates that centers invest in reference architecture. However, that begs the question of what architectural principles should shape it and how it should be evaluated. Section 6 suggests answers that deserve study in terms of architectural principles, quality attributes, and architectural patterns.
- *Formal methods*. Many computer scientists believe that formal methods and supporting tools are essential for addressing the challenges of complexity. As the late Edsger Dijkstra stated: "The required techniques of effective reasoning are pretty formal, but as long as programming is done by people who don't master them, the software crisis will remain with us and will be considered an incurable disease."
- *Firmware and Field Programmable Gate Arrays (FPGA)*. Microcontrollers and FPGAs are programmable devices that have seen significant growth, but the growing problems of programming and debugging them have escaped the kind of scrutiny given to software because they are often viewed as "hardware," though many issues are the same.
- *Pair Programming*. "Pair programming" is a software development practice in which two programmers work together, side-by-side, on the same module. Studies have shown increased productivity, increased quality, and less risk of knowledge loss due to staff loss. The practice is barely used within NASA's flight software community.
- *Programming Language*. The choice of programming language is important because it affects how programmers think about software design. The C language is currently the dominant programming language for flight software, but it is a minimalist language that lacks support for some abstractions. NASA and the rest of the aerospace industry should examine the impact of programming language on productivity and quality.
- *Human Capital*. A study by the U.S. Army indicates a diminishing national capability in engineering. Fewer U.S. residents are getting technical degrees. There is a lack of system and software engineers in key leadership positions, a shortage of highly experienced software managers and architects, and fewer young people going into systems and software engineering. This trend is affecting the whole aerospace industry and, unless mitigated, will limit NASA's abilities relative to other space agencies.

10 Past and Future

The concerns about flight software that led to this study are similar to the concerns that led to the 1968 NATO conference on software engineering. Software systems were unreliable, often didn't meet requirements, and often exceeded schedule and budget. Especially pertinent was a concern voiced in 1968— "the seemingly unavoidable fallibility of large software, since a malfunction in an advanced hardware-software system can be a matter of life and death"—a risk that exists to this day. Since then, computer science and software engineering has made progress to the extent that our concerns are now about million-line programs rather than thousand-line programs. Much of that progress can be attributed to improved education for software engineers and better tools, such as better programming languages, reusable libraries, design patterns, software frameworks, and verification tools and techniques.

Looking back across forty years of history since the NATO conference, we're still left with the uncomfortable prospect that software will only become more complex as we continue to push the boundaries of space exploration. What Fred Brooks asserted in 1975 is still true:

> "The tar pit of software engineering will continue to be sticky for a long time to come. One can expect the human race to continue attempting systems just within or just beyond our reach; and software systems are perhaps the most intricate and complex of man's handiworks" [Brooks 1975].

In the present day (2009), a new program initiated by the National Science Foundation is seeking proposals to deal with the increased complexity of systems that tightly conjoin and coordinate computational and physical resources, of which NASA's space systems are one example. The following text from the NSF introduction clearly describes the challenges, all of which should be familiar to NASA project managers:

> "Despite the rapid growth of innovative and powerful technologies for networked computation, sensing, and control, progress in cyber-physical systems is impeded on several fronts. First, as the complexity of current systems has grown, the time needed to develop them has increased exponentially, and the effort needed to certify them has risen to account for more than half the total system cost. Second, the disparate and incommensurate formalisms and tools used to deal with the cyber and physical elements of existing systems have forced early and overly conservative design decisions and constraints that limit options and degrade overall performance and robustness. Third, in deployed systems, fears of unpredictable side-effects forestall even small software modifications and upgrades, and new hardware components remain on the shelf for want of true plug-and-play infrastructures. Fourth, current systems have limited ability to deal with uncertainty, whether arising from incidental events during operation or induced in systems development. These problems are endemic to the technology base for virtually every sector critical to U.S. security and competitiveness, and *they will not be solved by finding point solutions for individual applications* (emphasis added). The solutions that are needed are central to the gamut of cyber-physical system application domains. It is imperative that we begin to develop the cross-cutting fundamental scientific and engineering principles and methodologies that will be required to create the future systems upon which our very lives will depend" [NSF 2008].

The text above offers small comfort in that NASA is not alone in its concerns about the present state. Viewed more positively, the NSF program on Cyber-Physical Systems provides an opportunity for NASA technologists to engage with a broader community of researchers seeking to "develop the cross-cutting fundamental scientific and engineering principles and methodologies" that NASA will need to achieve ever-more-ambitious missions.

Looking farther into the future, some of the solution will likely come from a technology that seems like science fiction, even though it is being used selectively today. As a leading thinker on self-organization and the science of complexity as applied to biology, Stuart Kaufmann writes:

> "The sensitivity of our most complex artifacts to catastrophic failure from tiny causes—for example, the Challenger disaster, the failed Mars Observer mission, and power-grid failures affecting large regions—suggests that we are now butting our heads against a problem that life has nuzzled for enormously longer periods: how to produce complex systems that do not teeter on the brink of collapse. Perhaps general principles governing search in vast spaces of possibilities cover all these diverse evolutionary processes, and will help us design—or even evolve—more robust systems" [Kaufmann 1995].

Here, Kaufman is referring to promising results in the field of evolutionary computation that are already being applied to improve designs in a variety of fields. Kaufman is forecasting the days when the computational power of computers will help us design (evolve) software systems that are far more robust than today's handcrafted systems.

11 Summary

Problems in recent NASA missions—broadly attributed to software—have focused attention on the growth in size and complexity of flight software. In some areas, the history shows exponential growth in flight software size, similar to the growth seen in other embedded real-time software (ERTS), particularly DoD weapon systems. Even though some NASA missions under development will approach or exceed one million lines of flight code, they are not among the largest ERTS, such as the F-35 Joint Strike Fighter (5.7 M lines of code) and the Boeing 787 (7 M lines of code). The growth of ERTS software is driven by ambitious requirements and by the fact that software (as opposed to hardware) more easily accommodates new functions and evolving understanding (easier to modify).

The scope of the complexity problem is not limited solely to flight software development, but extends into requirements, design, testing, and operations. The multidisciplinary nature of space missions means that decisions made in one area sometimes have negative consequences on other areas, often unknown to the decision maker. Fault management was given special attention because it is among the most complex software to design and verify.

This study generated 15 recommendations, some of which are relatively simple and some of which require investment and training. For example, some complexity can be reduced by eliminating unnecessary requirements through close attention to rationale, and some design decisions can be improved through better communication among multiple disciplines, particularly in trade studies and descope decisions. For example, a simple hardware decision can make timing requirements difficult to achieve in software; lack of attention to testability can make verification more difficult and time-consuming; and a seemingly innocent software descope decision can complicate operations and increase the risk of operational error.

After the “easy” recommendations have been implemented, we’re still left with the fact that mission requirements are increasingly ambitious and that flight systems are typically built on heritage software that is architecturally weak for today’s challenges. For these reasons, recommendations about *architecture* have emerged as the most effective way to reduce and better manage complexity across the engineering lifecycle. Architecture, and architectural thinking, must span software *and* systems because software manages or controls most of the functionality in flight systems. Architecture must address issues that span the engineering lifecycle and cut across the different disciplines; in so doing, it facilitates inter-team communication, reducing errors of interpretation and errors of omission. Recommendations include more up-front analysis and architecting, establishment of software architecture reviews, development of multi-mission reference architecture, and measures to grow and promote software architects.

12 References

[Aldiwan & Wang 2007] Wafa Aldiwan and James Wang. Flight Software 101: Ten Things JPL Engineers Should Know about Flight Software. Flight Software and Data Systems Section (JPL).

[Augustine 1977] Augustine, Norman, Augustine's Laws, 6th Edition, American Institute of Aeronautics and Astronautics, June 1977.

[Baldwin 2007] Kristen Baldwin, DoD Software Engineering and System Assurance – New Organization, New Vision. https://acc.dau.mil/CommunityBrowser.aspx?id=175649&lang=en-US

[Bass 2003] Len Bass, Paul Clements, Rick Kazman, 2003. Software Architecture in Practice, Second Edition. SEI Series in Software Engineering, Addison-Wesley, 2003.

[Boehm 1981] Barry W. Boehm, *Software Engineering Economics*, Englewood Cliff, Prentice Hall, 1981.

[Brooks 1975] Frederick Brooks, Jr., The Mythical Man-Month – Essays on Software Engineering, Addison-Wesley Publishing Company, Inc, 1975.

[Brooks 1986] Frederick Brooks, Jr., No Silver Bullet: Essence and Accidents of Software Engineering, *Proceedings of the IFIP Tenth World Computing Conference*, pp. 1069-1076, 1986.

[Chien 2005] S. Chien, R. Sherwood, D. Tran, B. Cichy, G. Rabideau, R. Castano, A. Davies, D. Mandl, S. Frye, B. Trout, S. Shulman, D. Boyer. Using Autonomy Software to Improve Science Return on Earth Observing One, Journal of Aerospace Computing, Information, and Communication, April 2005.

[Clements 2002] Paul Clements, Rick Kazman, Mark Klein. Evaluating Software Architectures: Methods and Case Studies. Addison-Wesley.

[Coverity 2008] "Controlling Software Complexity: The Business Case for Static Source Code Analysis," Coverity, Inc. http://www.coverity.com

[Dörner 1989] Dietrich Dörner, The Logic of Failure: Recognizing and Avoiding Error in Complex Situations, Rowohlt Verlag GMBH 1989. English translation: Basic Books, 1996.

[Doyle 2007] Richard Doyle, Monte Goforth, C. Douglass Locke. ESMD Software Workshop (ExSoft 2007): Workshop Summary Report, April 10-12, 2007, Houston

[Dvorak 2008] Gripe Session on Operations Complexity, JPL internal report, 4/24/2008.

[Fesq 2008] NASA SMD/PSD Planetary Spacecraft Fault Management Workshop White Paper.

[Gillette 1998] Walter B. Gillette, Vice President of Engineering, Boeing Commercial Airplanes. Presentation at 17th Digital Avionics Systems Conference, 11/1998.

[Humphrey 2001] Watts S. Humphrey. "Winning with Software: An Executive Strategy", 2001, Addison-Wesley Professional, part of the SEI Series in Software Engineering.

[IEEE 1990] *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries.* New York, NY: 1990.

[Kaufmann 1995] At Home in the Universe: The Search for the Laws of Self-Organization and Complexity. Oxford University Press, 1995. ISBN-13: 978-019-511130-9.

[Kleinknecht 1970] “Design Principles Stressing Simplicity”, Chapter 2 of “What Made Apollo a Success?” NASA SP-287, http://history.nasa.gov/SP-287/sp287.htm

[Lyu 1996] Michael R. Lyu, editor, *Handbook of Software Reliability Engineering*, McGraw-Hill, 1996.

[Maranzano 2005] Joseph Maranzano, Sandra Rozsypal, Gus Zimmerman, Guy Warnken, Patricia Wirth, and David Weiss, Architecture Reviews: Practice and Experience. *IEEE Software*, March/April 2005.

[NASA 2008a] NASA Planetary Spacecraft Fault Management Workshop, http://icpi.nasaprs.com/NASAFMWorkshop.

[NATO 1968] NATO Software Engineering Conference, 1968, Garmisch, Germany

[NESC 2007] Design, Development, Test and Evaluation (DDT&E) Considerations for Safe and Reliable Human-Rated Spacecraft Systems”, NESC RP-06-108, May 1, 2007.

[NPR 7123.1A] NPR 7123.1A, NASA Systems Engineering Processes and Requirements.

[NPR 7150.2] NPR 7150.2, NASA Software Engineering Requirements.

[NSF 2008] Cyber-Physical Systems, program solicitation, NSF 08-611, Directorate for Computer & Information Science & Engineering and Directorate for Engineering, http://www.nsf.gov/pubs/2008/nsf08611/nsf08611.htm

[Perrow 1984] Charles Perrow, *Normal Accidents: Living with High-Risk Technologies*, Basic Books, 1984. Updated by Princeton University Press, 1999.

[Rasmussen 2008] Robert D. Rasmussen, GN&C Fault Protection Fundamentals, Proceedings of 31st Annual AAS Guidance and Control Conference, American Astronautical Society.

[Reed 2002] Paul Reed. Reference Architecture: The Best of Best Practices, http://www.ibm.com/developerworks/rational/library/2774.html, accessed Oct. 2008.

[Shapiro 1997] Stuart Shapiro, Splitting the Difference: The Historical Necessity of Synthesis in Software Engineering, *IEEE Annals of the History of Computing*, Vol. 19, No. 1, 1997.

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

Product Requirements Development and Management Procedure, LMS-CP-5526 (LaRC)

[Schuler 2007] Pat Schuler (LaRC), Peer Review Inspection Checklists (a collection of checklists at LaRC).

[Weiss 2008] Kathryn Weiss (JPL), Software Architecture Review Process, draft, https://bravo-lib.jpl.nasa.gov/docushare/dsweb/View/Collection-46223

[Wilmot 2007] Jonathon Wilmot (GSFC), “A Core Plug and Play Architecture for Reusable Flight Software Systems”, Space Mission Challenges for Information Technology 2007, July 2007, Pasadena.
http://ntrs.nasa.gov/search.jsp?R=304271&id=2&qs=N%3D4294669721

Final Report

NASA Study on Flight Software Complexity

Appendix A — Literature Survey on Software Complexity

Cin-Young Lee,
Jet Propulsion Laboratory,
California Institute of Technology

Contents

1 Setting the Scene

1968 was the year that brought us the **software crisis**, which referred “to the tendency for there to be a gap, sometimes a rather large gap, between what was hoped for from a complex software system, and what was typically achieved” (Naur 1968).

Nearly two decades later, in 1987, the crisis remained. Software projects were “capable of becoming a monster of missed schedules, blown budgets, and flawed products” [Brooks 1987]. There had been no silver bullet developed in the intervening two decades “to make software costs drop as rapidly as computer hardware costs” [Brooks 1987]. Moreover, Brooks argued that “we cannot expect ever to see twofold gains (in software productivity) every two years” [Brooks 1987]. In other words not only had no silver bullet been found, but no silver bullet would ever be found. Two decades later, in 2008, that silver bullet has yet to be found, as evidenced by remarks made by a panel celebrating the 20th anniversary of Brooks’ seminal paper [Fraser 2008].

In 1994, Scientific American revisited the software crisis and rechristened it **software’s chronic crisis** since it had yet to be solved [Gibbs 1984]. The following year the Standish Group offered up its Chaos report that provided a grim view of software project success. Their research showed that “a staggering 31.1% of projects will be cancelled before they ever get completed. Further results indicate 52.7% of projects will cost 189% of their original estimates” [The Standish Group 1995]. Six years later in 2001, the Standish Group followed up their Chaos report with Extreme Chaos in which they reported significant improvements in project success rates. Nonetheless “Nirvana is still a long way off — 137,000 projects were late and/or over budget, while another 65,000 failed outright” [The Standish Group 2001].

So, the question is: Why is there still a software crisis in 2008, forty years after its first mention at the inaugural NATO Software Engineering Conference in Germany? The answer was the same then as it is now: software complexity is tied to the system/problem complexity, which has been continually increasing. This relationship is a result of software becoming the “glue” to integrate large systems [Belady 1989]. Put another way, “software is the system’s complexity sponge” [Rasmussen 2007].

The picture becomes bleaker when considering embedded software systems, which have become more prominent in the last two decades. The added complexity of physical constraints for embedded software systems has bucked traditional computer science and software engineering.

> Existing software design techniques aren’t suitable [for embedded software]. Partly because it is recent, and partly because of the domain expertise it requires, embedded software is often designed by engineers who are classically trained in the domain, for example in internal combustion engines. They have little background in the theory of computation, concurrency, object-oriented design, operating systems, and semantics. In fact it is arguable that other engineering disciplines have little to offer to the embedded system designer today because of their mismatched assumptions about the role of time and because of their profligate use of hardware resources. But these disciplines will be essential if embedded software is to become more complex, modular, adaptive, and network aware [Lee 2000].

2 Describing the Problem

In his 1984 paper, “No Silver Bullet”, Brooks puts forth the idea that complexity is made up of **essential** and **accidental complexity** [Brooks 1987]. More precisely:

> **Accidental complexity** is complexity that arises in computer programs or their development process (computer programming) which is non-essential to the problem to be solved. While **essential complexity** is inherent and unavoidable, accidental complexity is caused by the approach chosen to solve the problem. While sometimes accidental complexity can be due to mistakes such as ineffective planning, or low priority placed on a project, some accidental complexity always occurs as the side effect of solving any problem. For example, the complexity caused by out of memory errors is an accidental complexity to most programs that occurs because one decided to use a computer to solve the problem [Wikipedia].

Paraphrasing, essential complexity arises from **what the system should do** and accidental complexity is a result of **how the system is built**. Given this distinction, it is important not to fall into the trap of specifying the “how” before identifying and understanding the “what”. After all, as the adage goes, “when you have a hammer, everything looks like a nail.” Consequently, a better approach to dealing with complexity is to first identify the problem (the what), then develop or select a solution (the how) that is well suited for the identified problem. This approach is universally adopted across nearly all mature engineering disciplines and is often described as design patterns or toolboxes [Gamma, et al. 1995, McConnell 2004, Rasmussen 2007]. The toolbox metaphor is used here. A toolbox implies that there is a set of re-usable tools that can be used to solve problems in different domains. However, the space of all possible problem types and the corresponding tools is too large to investigate exhaustively due to cost and time constraints. Hence, it becomes important to focus tool development on the most important aspects driving essential and accidental complexity.

The Standish Group produced a series of *Chaos* reports that identified the top ten factors most responsible for a software project’s success or failure, as shown in

Table A.**1**[1]. In general, the factors for project failure are the opposite of the factors for project success.

[1] Success is defined here as a software project that finishes on time, on budget, and on specification (or with no missing functionality). Failure is defined here as a software project that is cancelled and never delivered.

Table A.1. Standish Group Top 10 Factors Most Responsible for Project Success or Failure[2].

Factor for Project Success (The Standish Group 2001)	Factor for Project Failure (The Standish Group 1995)
Executive Support	Lack of Executive Support
User Involvement	Lack of User Involvement
Experienced Project Manager	Lack of IT Management
Clear Business Objectives	Didn't need it any longer
Minimized Scope	Unrealistic Expectations
Standard Software Infrastructure	Lack of Resources
Firm Basic Requirements	Changing Requirements & Specifications
Formal Methodology	Incomplete Requirements
Reliable Estimates	Lack of Planning
Other (small milestones, planning, competent staff)	Technology illiteracy

By categorizing each of these factors into either accidental or essential complexity drivers, we can focus our efforts on the facets of essential and accidental complexity that will return the most "bang for the buck." This categorization is shown in Table A.2.

Table A.2. Top 10 Factors Mapped to Complexity Type.

Essential Complexity Factors	Accidental Complexity Factors
User Involvement	Executive Support
Clear Business Objectives	Experienced Project Manager
Minimized Scope	Formal Methodology
Incomplete Requirements	Standard Software Infrastructure
Firm Basic Requirements / Changing Requirements	Reliable estimates
	Competent Staff / Technology Illiteracy
	Planning
	Lack of Resources
	Lack of IT Management

The essential complexity factors are equivalent to having the correct, unambiguous, necessary, complete, and stable requirements, respectively. Evidently, the biggest issue facing essential complexity is requirements engineering and analysis. Conversely, on the accidental complexity side of things, the number and range of issues is more diverse and spans a whole spectrum of issues that include process, organization, standards, etc.

[2] The Standish list does not include the factors identified by the NASA software community, which include difficulty in software verification and the use of real-time embedded systems. These factors are addressed in the remaining sections.

3 Resolving the Problem

The software engineering toolbox consists of many tools with specific domains of applicability and corresponding advantages and disadvantages. A preliminary attempt at cataloging the tools for software engineering is provided in the sections that follow[3]. Note that by "tools" we mean solution approaches and not specific software tools or implementations.

In the tool descriptions that follow, the tools are separated largely according to the identified essential and accidental complexity factors. Each tool is then followed by a description of the tool and a discussion of its applicability that describes the added accidental complexity the choice of that tool entails. Note that the tools specified for each development activity are not either/or propositions. Often times, a good approach is to take a middle ground between the prescribed tools to solve the engineering problem at hand.

3.1 Essential Complexity

3.1.1 Requirements Engineering

As described in Section 2, the primary issue of essential complexity is the actual identification of the problem, or the requirements engineering process. Recent developments have pushed model-driven engineering (requirements and design) as compared to traditional methods, which have been primarily statement based.

Table A.3. Engineering approaches.

Engineering Approach	Description	Applicability
Model-based engineering	Model-based engineering express functional and behavioral specifications and designs in terms of models.	Models provide an additional level of abstraction for removing unnecessary details/complexity. Moreover, use of a standard modeling language allows unambiguous interpretation of requirements and design. However, there is considerable overhead required for tools, training, etc.
Non-model-based engineering	Most non-model-based engineering methodologies use statement-based requirements engineering in which requirements and design are specified in sentences or statements.	Spoken languages are notoriously imprecise and ambiguous, which often leads to misinterpretation and misunderstanding. However, there are fewer upfront costs to this approach.

3.2 Accidental Complexity

3.2.1 Formal Methodology

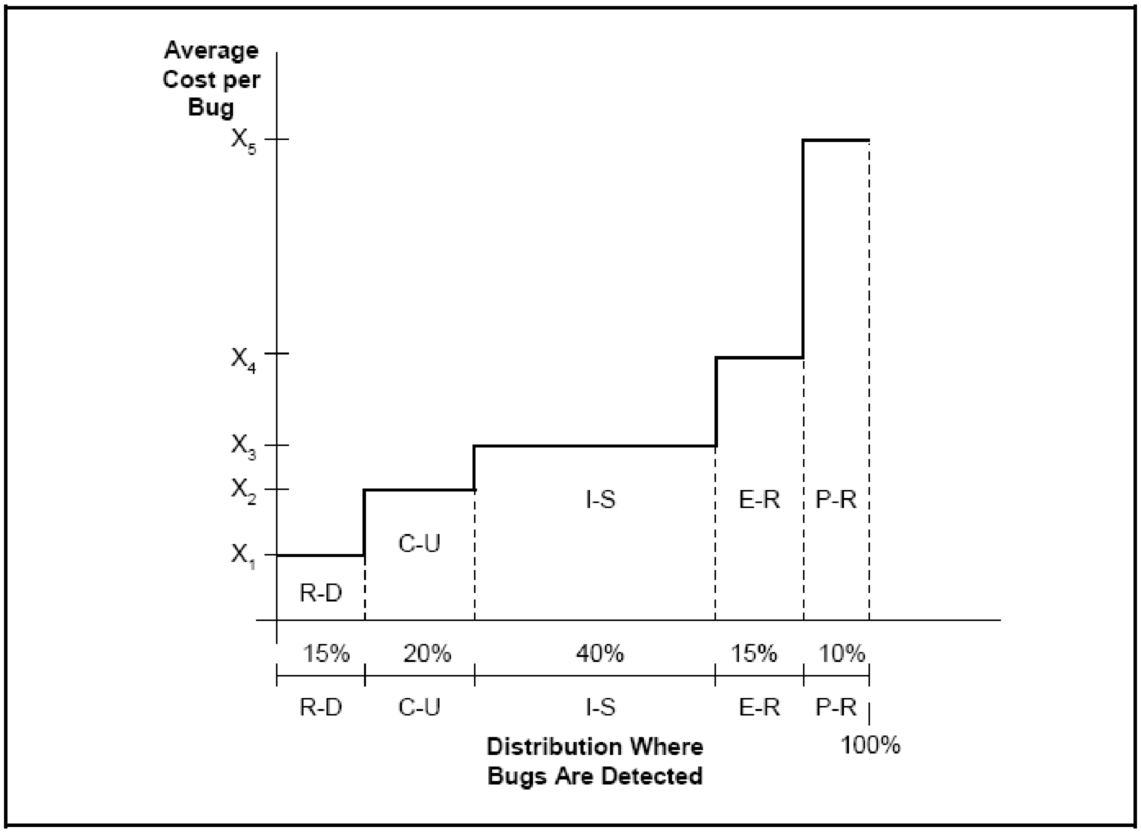
While there are many different software development methodologies, most agree that software development consists of five different activities. These activities are typically conducted sequentially and consist of:

1. Requirements Analysis and Definition.

[3] The catalog of software engineering tools included here is not intended to be comprehensive, but rather a representative view of the currently available tools.

2. System and Software Design.
3. Implementation and Unit Testing.
4. Integration and System Testing.
5. Operation and Maintenance.

Development methodologies structure and order the development activities to maximize the likelihood that software is delivered on time, on budget, and on specification. Studies have shown that defects found in the later activities are more difficult or more costly to fix than those found earlier (see Figure A.1 for an example). Hence, much of the methodologies are developed with the idea of finding defects earlier rather than later. Current methodologies can be divided along a scale of two contrasting paradigms: Big Design Up-Front (BDUF) and You Ain't Gonna Need It (YAGNI). Typically, most developers adopt a happy medium between the two.



Legend:
R-D: Requirements Gathering and Analysis/Architectural Design
C-U: Coding/Unit Test
I-S: Integration and Component/RAISE System Test
E-R: Early Customer Feedback/Beta Test Programs
P-R: Post-product Release

Figure A.1. Software Testing Costs Shown by Where Bugs are Detected (Reproduced from Figure 5-3 in [RTI May 2002]).

Table A.4. Development methodologies.

Development Methodology	Description	Applicability
Big Design Up-Front (BDUF)	BDUF is most commonly associated with waterfall development in which each development activity is not started until the preceding activity has been completed. The idea is to design everything up front such that defects are found at the earliest activity, leading to reduced costs later in the development lifecycle.	BDUF is important for strategic development to ensure that proper frameworks are considered and put into place before they are needed. However, when requirements change due to scope creep or a better understanding of the system, it is often difficult to modify the original design to accommodate the changes. As with defects, changes become more costly the further down the activity chain one goes.
You Ain't Gonna Need It (YAGNI)	YAGNI is most commonly associated with agile development in which multiple cycles through the development activities are taken to iteratively and incrementally build the system on an as needed basis.	The rapid cycles in YAGNI lend themselves to constant integration and test allowing defects to be found and dealt with in an incremental fashion. This approach easily accommodates the injection of new or changed requirements at each iteration. However, by developing only to what is currently needed, it is possible that requirements for later iterations, which have not been considered, may not be easily accommodated in the as built software. In other words, each new YAGNI iteration introduces new constraints that will preclude or complicate certain types of future capabilities.

Each methodology has associated practices that are worth investigating and are useful in reducing certain types of accidental complexity. For example, pair programming (a typically YAGNI development type process), in which each software task has a pair of developers allocated to it, reduces the risks of staff turnover or absences. However, there is an apparent reduction in productivity, as each task now requires two people. Some evidence suggests that paired programmers are less likely to slack off, reducing the overall effect of double allocation.

3.2.2 Software Program Verification and Integration

Verification of the correctness of software programs has grown exponentially in relation to the growth in system design. What used to be a small portion of the entire development effort has now become 50% or sometimes even 70% of the total development effort [Ra 2002, Bailey 2007]. This rapid increase in the complexity and cost of program verification has brought about the need for automatic verification methods. These methods can largely be separated into formal verification and software testing approaches. It is widely accepted now that both are needed for adequate verification coverage [Shapiro 1997]. A variety of automated tools have been developed. For formal verification, there are tools such as SPIN [Holzmann 2004] and NuSMV [NuSMV Project n.d.]. For software testing, due to the current cost and time-prohibitive nature of exhaustive testing, factorial experiments or random test generation approaches are taken. Some examples include Monte Carlo methods and evolutionary computation approaches such as EvoTest [EvoTestWebsite 2008].

3.2.3 Organization and Planning

An often-overlooked driver for accidental complexity is the structure of the software development organization. The observation that a system's design takes the same form as the development organization's structure was first documented and investigated by Conway [Conway 1968][4]. His conclusions appear generally applicable even now and are repeated here (emphasis added).

> The basic thesis of this article is that organizations which design systems (in the broad sense used here) are constrained to produce designs which are copies of the communication structures of these organizations. We have seen that this fact has important implications for the management of system design. Primarily, we have found a criterion for the structuring of design organizations: a design effort should be organized according to the need for communication. This criterion creates problems because the need to communicate at any time depends on the system concept in effect at that time. Because the design which occurs first is almost never the best possible, the prevailing system concept may need to change. **Therefore, flexibility of organization is important to effective design** [Conway 1968].

3.3 Embedded Software Systems

Embedded systems provide additional challenges to current software engineering practice because of the interplay required between computational and physical constraints of embedded systems. This challenge is not well suited for existing tools (software engineering or otherwise).

> We see the main culprit as the lack of rigorous techniques for embedded systems design. At one extreme, computer science research has largely ignored embedded systems, using abstractions that actually remove physical constraints from consideration. At the other, embedded systems design goes beyond the traditional expertise of electrical engineers because computation and software are integral parts of embedded systems [Henzinger 2007].

3.4 Conclusions

The good news is that software complexity can be managed through identification of problems and use of the tools most appropriate for those problems. However, the set of existing tools is not complete nor is generation of new tools or improvement of old tools free. Tools have research, development, implementation, and maintenance costs. The payoff, of course, is that, once effective tools have been developed for specific problems, they can be re-used for little or no cost across all problems within the same domain, thereby saving time and money in the long run.

The bad news is that there aren't many avenues to pursue the development, analysis, and improvement of new tools, particularly for real-world applications. In large part, this is due to the observation that anything that works will be used in progressively more challenging applications until it fails[5]. Let us hope that a catastrophic failure is not what is needed to bring about the

[4] The thesis of Conway's paper was later coined as Conway's Law in Fred Brooks' seminal book *The Mythical Man-Month*.

[5] This is often referred to as the generalized Peter Principle as coined by Dr. William R. Corcoran in his work on the Corrective Action Programs at nuclear power plants.

development of new tools that are more appropriate for today's problems. As the movie director James Cameron put it:

> You have to balance the yin and yang of caution and boldness, risk aversion and risk taking, fear and fearlessness. No great accomplishment takes place, whether it be a movie or a deep ocean expedition or a space mission, without a kind of dynamic equipoise between the two. Luck is not a factor. Hope is not a strategy. Fear is not an option [James Cameron].

3.5 References

Bailey, Brian. "Dealing with SoC hardware/software design complexity with scalable verification methods." *embedded.com.* April 27, 2007. http://www.embedded.com/columns/technicalinsights/199202018?_requestid=414778.

Belady, Laszlo A. "Software is the Glue in Large Systems." *IEEE Communications Magazine*, August 1989: 33-36.

Brooks, Frederick P. "No Silver Bullet: Essence and Accidents of Software Engineering." *Computer* (IEEE), April 1987: 10-19.

Conway, Melvin E. "How Do Committees Invent?" *Datamation*, April 1968.

EvoTestWebsite. *EvoTest.* 2008. http://evotest.iti.upv.es/.

Fraser, Steven and Mancl, Dennis. "No Silver Bullet: Software Engineering Reloaded." *IEEE Software* (IEEE), January/February 2008: 91-94.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

Gibbs, W. "Software's Chronic Crisis." *Scientific American,* September 1984: 72-81.

Henzinger, Thomas A. and Sifakis, Joseph. "The Discipline of Embedded Systems Design." *Computer* (IEEE), October 2007: 32-40.

Holzmann, G. J. *The SPIN Model Checker: Primer and Reference Manual.* Addison-Wesley, 2004.

Lee, Edward A. "What's Ahead for Embedded Software?" *Computer* (IEEE), September 2000: 18-26.

McConnell, Steve. *Code Complete 2.* Microsoft Press, 2004.

Mellor, Stepehen J., Kendall Scott, Axel Uhl, and Dirk Weise. *MDA Distilled: Principles of Model-Driven Architecture.* Boston: Addison-Wesley, 2004.

Naur, P. and Randell, B. *Software Engineering: Report of a conference sponsored by the NATO Science Committee.* Brussels, Germany: First NATO Software Engineering Conference, 1968.

NuSMV Project. http://nusmv.fbk.eu/ (accessed 2008).

Ra, Oystein and Strom, Torbjorn. "Trends and Challenges in Embedded Systems - CoDeVer and HiBu Experiences." *Norsk Informatikkonferanse (NIK) 2002* . Kongsberg, Norway, 2002.

Rasmussen, Robert. "Software Complexity and Architecture." *JPL Internal Presentation.* 2007.

RTI. *Planning Report 02-3: The Economic Impacts of Inadequate Infrastructure for Software Testing.* National Institute of Standards & Technology, May 2002.

Shapiro, Stuart. "Splitting the Difference: The Historical Necessity of Synthesis in Software Engineering." *IEEE Annals of the History of Computing* 19, no. 1 (1997): 20-54.

The Standish Group. "CHAOS." The Standish Group Report, 1995.

The Standish Group. "Extreme Chaos." The Standish Group Report, 2001.

Final Report

NASA Study on Flight Software Complexity

Appendix B — Architectures, Trade Studies, Avionics Impacts

Kirk Reinholtz,
California Institute of Technology,
Jet Propulsion Laboratory

Daniel Dvorak,
California Institute of Technology,
Jet Propulsion Laboratory

Contents

Executive Summary

In 2007 the NASA Office of Chief Engineer commissioned a study of growth in size and complexity of flight software in NASA missions. Part of the study seeks answers to questions about the sources of complexity, ways to reduce unnecessary complexity, and ways to better manage necessary complexity, particularly with respect to software architectures, trade studies, and avionics. This report summarizes discussions on these and other questions in meetings held at GSFC, JSC, MSFC, APL, and JPL.

Acknowledgment

This report was only possible because of the many deep and insightful observations made by the interviewees. The list of interviewees and their titles/positions are shown in Section 1.4 of the main report, of which this is an appendix.

1 Introduction

The last several decades have seen a steady growth in the size of NASA flight software, the requirements on the software, its cost, and its perceived complexity. This trend is seen in both the human and robotic space missions. The Orion flight software load, for example, is estimated at several hundred times the size of the Apollo flight software, and runs on avionics with thousands of times the RAM and CPU performance of the Apollo avionics.

In September of 2007 the NASA Office of the Chief Engineer (OCE) initiated a study of this growth because it has serious technical, operations, and management implications for space flight missions. The closely related aeronautics industry has reported exponential growth in both the number of signals to be processed and source lines of code (SLOC) required for civil aircraft over a thirty-year period. While advances in computer hardware have readily accommodated this increase by providing faster processors and increased memory, there are concerns that software engineering and management approaches have not kept pace.

This report presents the results of "Special Interest 2, Approach 1" of that study, which was chartered as follows:

> "Special Interest 2: How can unnecessary growth in complexity be curtailed? How can necessary growth in complexity be better engineered and managed? Are there effective strategies to make smart trades in selecting & rejecting requirements for inclusion in flight software to keep systems lean? When growth in complexity and software size are necessary, what are the successful strategies for effectively dealing with it?
>
> Approach 1:
>
> Architectures that are specifically designed to enable flight/ground capability trades are critical to moderating unnecessary growth in FSW complexity. The Mission Data System developed at JPL is an example of such an architecture. Furthermore, we propose to characterize the interaction between avionics design choices and the impact they have on FSW complexity. We propose to conduct interviews to explore this and other architectural approaches as a means of managing FSW complexity.

The interviews described above were conducted over the period from December 2007 to March 2008 at five locations: GSFC, MSFC, JSC, APL, and JPL. This report summarizes and categorizes the results of those surveys.

The following section describes the approach taken to create, conduct, and analyze the survey. The section after that presents a summarization of the results, by survey question. The next section presents an overall analysis and generalization of the results. The final section presents potentially actionable findings.

2 Approach

The work leading to this report was conducted in four phases:

1. A memo describing the survey in general terms, its objectives, and specific questions to be answered was written and distributed to participating centers.
2. Each participating center identified the individuals best positioned to answer the questions on behalf of the center.
3. The PI traveled to each center and conducted the survey via face-to-face interviews and roundtable discussions.
4. The raw notes from the interviews were analyzed and summarized, resulting in this report.

3 Summary of Results by Question

This section presents a summary of the results of the survey, organized by question. The survey memo included the following statements about “complexity” which were offered and accepted as starting points for this study.

- This study focuses on complexity in flight software, including the requirements that cause such complexity, with “risk” as the main concern about growth in size and complexity of flight software.
- Complexity is a measure of understandability, and lack of understandability leads to errors. A system that is more complex may be harder to specify, harder to design, harder to implement, harder to verify, harder to operate, risky to change, and/or harder to predict its behavior.
- Complexity affects not only human understandability but also “machine understandability”. For example, static analysis tools for simple languages such as C are stronger than tools for more complex languages such as C++.
- It’s important to distinguish between “essential complexity” and “incidental complexity”. Essential complexity arises from the problem domain and mission requirements, and can only be reduced by descoping. Incidental complexity arises from choices made about hardware, architecture, design and implementation, and can be reduced by making wise choices.
- Essential complexity can be moved but not removed (except by descoping). For example, a decision to keep flight software simple may simply move some complexity to operations, along with attendant costs and risks.
- Architecture is about managing complexity. Good architecture—for both software and hardware—provides helpful abstractions and patterns that promote understanding, solve general domain problems, and thereby reduce design defects.
- “Flight software” is loosely interpreted to mean software that is used in flight and whose failure can severely impair a mission or cause complete mission loss. By this definition, flight software includes obvious candidates such as attitude control and command handling and sequences and fault protection, but also includes a less obvious candidate such as parameters (MGS was lost, in part, due to an erroneous parameter update). This definition can include instrument software if defects in that software can cause science or exploration objectives to go unachieved.

In general each item discussed in the following subsections has a “sweet spot”. Too much or too little is not as good as the right amount. The items tend to participate in a program-wide trade-space because they take time or money to achieve, so some items will seem to conflict with others. Most items also have exceptions, as well, though this report omits such qualifiers (“usually”, “generally”, etc.) for the sake of brevity and readability.

3.1 General

1. *What are the sources of growth in complexity? What kinds of requirements increase complexity on flight software?*

Essential complexity:

- Increasing needs for autonomy.
- Increasingly demanding mission objectives.
- Increasingly sophisticated fault protection ("finish the mission" rather than "stop and call home").
- Increasing Nav/GNC capability.
- Increasing safety-oriented requirements.
- Fault protection with tight timing requirements.

Incidental complexity:

- Too many requirements.
- Requirements too proscriptive.
- Requirements with bad cost/benefit ratio.
- Inconsistent requirements.
- Requirements at wrong level.
- Use of standards with bad cost/benefit ratio.
- Inadequate or complicated avionics.
- Inadequate time spent designing/optimizing.
- Abstractions that get too deep, out of hand.
- Architecture rot, weak architecture.
- FP combinatorics beyond comprehension, beyond testable.
- Risky technology backed up with risky technology.

Other sources:

- General growth to fit available resources.

Discussion:

1. Interesting not a lot about complicated code, languages, etc.
2. Several people mentioned that code and functionality tends to grow to fit the avionics. This is a very significant effect if true. Worthy of more thought. Human nature? Consequence of less time designing? Enables less time designing?

2. *How can unnecessary growth in complexity be curtailed?*

- Good reviews to identify and excise incidental complexity and low cost/benefit essential complexity.
- Systems thinkers to make sure overall system is traded to achieve best whole-system results, e.g. avoid “simple” software that’s very expensive and complicated to operate.
- Modular software architecture to isolate concerns and allow composition of individually tested elements.
- Good architecture rooted in the fundamental mission objectives.
- Design fault protection early, otherwise it is added on piecemeal, distorting the architecture a little more with each addition.
- If heritage software is used, make sure its fundamental assumptions match the fundamentals of the new mission.
- Provide time and process for iteration on the requirements between all stakeholders to make sure the system has been distilled down to its essence.
- Use design patterns to capture sound solutions to recurring engineering needs, so less time spent reinventing wheel, freeing resources to work on more important things.
- Imbed with the contractor rather than trying to get everything into the requirements.

3) *How can necessary growth in complexity be better engineered and managed?*

- Plenty of reviews, with enough time to study the materials, with dialog (i.e. flow both up and down), and with appropriately educated and experienced reviewers. Inappropriate reviewers can have negative value.
- Requirements should not constrain the implementation space.
- Architecture.
- Simplify, simplify, simplify. Remove elements from the software design until you can’t find anything else to remove.
- Be aware of the various human cognitive biases [1], and try to not let them get in the way of rational decision-making. NOTE: The authors strongly recommend that the citation is fetched and read. It’s a great list of how we humans go wrong in our attempts at rational logical reasoning. Many of the points made in the interviews can be traced to an instance of the cognitive biases, suggesting that an awareness of them could help us in our engineering decision-making. Also take a look at cockpit crew decision-making during a crisis.

4) *Are there effective strategies to make smart trades in selecting & rejecting requirements for inclusion in flight software to keep systems lean?*

- Don’t just accept requirements. Take the time to have plenty of reviews, with enough time to study the materials, with dialog (i.e. flow both up and down), and with appropriately educated and experienced reviewers. Inappropriate reviewers can have negative value.
- Consider end-to-end costs, not just partitioned costs. Rational self-interest tends to cause each WBS element to locally optimize, sometimes by pushing costs to another WBS element.

- Build a flight software architecture that minimizes the incremental cost of adding functionality, or modifying functionality. Otherwise as new or different needs are discovered (which they always are), they tend to get implemented as ground capabilities or workarounds, though they "should have" been implemented on the flight side. Fault protection is especially vulnerable to this phenomenon.

5) *When growth in complexity and software size are necessary, what are the successful strategies for effectively dealing with it?*

- Spend up-front time getting the architecture right. The key properties the architecture must have are modularity (isolating pieces of the system from other pieces of the system), layering (to provide common ways to do common things), and abstractions (to provide simple ways to use to capabilities with complex implementations).

6) *What technical strategies do you advocate to reduce and/or manage flight software complexity?*

- Minimize non-deterministic behavior, to facilitate code analysis and testing. "If you can test it exhaustively, it's simple. Otherwise, it's complex"[2].
- Repeated mention of architecture, modularity, partitioning.
- Build out of individually verifiable elements.
- Watch out for complexity added in order to shift costs, e.g. a subsystem adding a full computer instead of an ASIC so that some of the subsystem development cost is shifted to the flight software WBS.
- Understand and document (and if necessary question and push back on) the assumptions behind the requirements.
- Make lifecycle trades on crew involvement vs. automation.
- Software reuse brings complexity because the software, and the assumptions behind it, must be understood, which can be as much or more work as writing it in the first place. Reuse can also drag in functionality you don't need, increasing risk and incidental complexity.

7) *What managerial strategies do you advocate to reduce and/or manage flight software complexity?*

- Hire good people. Don't put an inexperienced person in a key position on a new or difficult element.
- Assign ownership to each interface.
- Have strong architectural element to the team.
- Defend the architecture, do not let it rot in the name of short-term gains.
- Give due weight to long-term attributes, e.g. Maintainability, operability.
- Do architecture up front. The architect needs real authority to protect the architecture, otherwise short-term cost and schedule pressure will lead to decisions that corrupt the architecture, to the detriment of lifecycle costs. According to one definition, architecture is

the set of design decisions that must be made early in a project, and for which the consequences of a bad decision only appear much later.

- Build software simulators so the developers can test “on the hardware” early in the development lifecycle. Note you may be able to make this somewhat easier by using partitioning e.g. ARINC-653 to simplify the simulator without compromising its fidelity with respect to the software.
- Educate the management, most don’t know software and tend to harbor non-trivial misunderstandings. Those misunderstandings bias management decisions, to the detriment of the software and so the system.

3.2 Flight/Ground Capability Trades

1) *What flight/ground capability trades are critical to moderating unnecessary growth in FSW complexity?*

- Trades often happen before a key stakeholder (e.g. the implementing organization) gets involved, resulting in another round of trades when the IRDs and ICDs are negotiated. It would be better to have all stakeholders at the table earlier.
- Capabilities, features need to look nice from more than a Computer Science perspective. Need experienced practitioner inputs earlier.
- Careful balance of putting capability on the flight side (to reduce ops complexity and so costs), vs. ground side (to reduce development costs). Enable migration so can test on ground, migrate to flight once tried and true.
- Incidental complexity tends to feed on itself (complexity begets complexity) so it’s very important to minimize it.
- High communication coverage and rapid turnaround is expensive, but the more you have the simpler the flight software can be, because ground can handle more of the issues that come up. Which of course in general increases operations cost.
- The longer the active phase of the mission (a mapping mission for example) the less complex the flight software has to be, because any opportunities missed because of an outage can be retried later, so high uptime is not as essential as a short or one-shot (e.g. EDL, flyby) where it either works or the mission is a failure.

2) *What are the downsides, if any, to moving some complexity to the ground software and/or operations?*

- Moving functionality to ground generally increases operations cost and reduces functionality and science return. It generally requires more-timely communications and more bandwidth. More people in the loop.
- Tends to drive towards more interactive systems, which causes delays due to the humans in the loop and comm. Latency.
- Put functionality as close to the flight software as you can, so bandwidth restrictions and latency issues do not add incidental complexity to the system. This requires timely communications.

- If the functionality is done by humans on the ground, then it increases the chance of operator error.

3) *What trades are most affected by expected mission duration?*

- The longer the mission, the more strings and redundancy (and attendant complexity) is necessary.
- The longer the mission, the more cross-strapping, software patch capability, RAM failure workarounds, and other software mitigations for failures are necessary.
- If the "long" part is science acquisition, then less uptime is required for the longer mission because outages can be amortized across a longer period of time and data can be acquired at the next opportunity. Short missions require higher uptime and so more onboard rapid-response fault handling.
- The longer the mission, the more important is operability (often mitigated with increased automation) because its cost more dominates the full lifecycle cost.

4) *What trades are most affected by distance from Earth?*

- Fault protection.
- Onboard recovery and autonomy.
- The further away, the higher the latency and so the more fault responses that must be handled onboard to meet timeliness requirements.

5) *Are there examples where rational self-interest does not lead to the best solution for a mission (or a whole program)?*

- Vendors tend to try to use what they have (NRE already amortized) though that may not be best solution for the program. It may be needlessly difficult to interface with or operate, for example.
- If the management team is different for different phases (e.g. development versus operations) then there will be a tendency to e.g. reduce development costs, which can cause a disproportionate increase in operations costs.

3.3 Avionics Design Choices

1) *What avionics designs have complicated flight software or made it harder to get right, and what was the effect on flight software? (Consider things such as lack of memory protection; lack of double-buffering; events that are not latched; slow data bus; insufficient RAM; polling versus interrupt-driven; etc)*

- Voting is very complicated.
- Excess CPU capacity invites requirements creep.
- Distributed architectures are more complex than centeralized ones.
- Trying to fit into constrained resources is a major driver of complexity.
- Interfaces that require very rapid response.

- Interfaces that do not provide the option of interrupt-driven, forcing polling. You can always turn off the interrupts if you want polling, but if they're not there in the first place the bridge is burned.
- Interfaces that do not provide atomicity, that is to say several ports must be manipulated separately to get the desired result. Especially bad if timing tight (forcing critical sections) or no way to insure the operation actually worked.
- Hardware interfaces should be designed with the help of software people to ensure that they do not impose unnecessary complexity on the avionics software. In other words, as key stakeholders, software people deserve a seat at the table.
- Use of processors with weak tool chains (e.g. no good compilers available, or must be programmed in assembly language).
- Not double-buffering outputs, causing all sorts of response time issues.
- Hardware without enough test points, without enough visibility into its operations.
- Incongruent fault protection mechanisms in hardware, e.g. persistence timers or enable/disables that don't facilitate system fault protection.
- Critical or timing-sensitive interfaces across a bus puts bus characteristics into the analysis making it much more complicated.
- Registers that do something when read, so complicates read-back of register state, or makes it impossible. They should just return the last value written.

2) *What choices have you made in avionics design to reduce demands on flight software?*

- Distribute functionality to remote controllers to reduce real-time demands on the central processor.
- Use simple and well-understood bus e.g. 1553.
- Use physically separate processors for critical and non-critical applications.
- Train developers in lessons learned on previous programs/projects/missions.
- Prefer DMA over programmatic i/o for high-throughput devices to offload CPU.
- Isolate hardware specifics behind device drivers or other layering mechanism.
- Each hardware device should do one thing, and do it well.
- Put logic in FPGAs (or more generally, application-specific processors). However, FPGA logic is harder to review than software and so errors are often discovered later, harder to fix, and more expensive to fix.
- Industry is moving towards sophisticated devices e.g. start trackers that return attitude information that can be directly used by GN&C, rather than star maps that require significant processing before use. Complexity increases in short term when older technologies are kept as backups to the newer "simpler" approach.
- Use of commercial standards and hardware designs (Intellectual Property, Cores) rather than inventing from scratch.

3) *What technical or managerial strategies have you used, or advocate, to avoid such problems?*

- Pay lots of attention to hardware/software interfaces.
- When device is contracted out, make sure contract specifies effective mechanisms of determining appropriate interfaces.
- Co-locate technical stakeholders so informal hallway conversations help converge the device interfaces and behavior. Downside of this approach is more information tends to reside in peoples heads, rather than on paper, because the paper is not an essential communication mechanism (I think Apollo had this problem too, after Apollo I fire they found the written as-builts were not accurate).
- Manage overall avionics interrupt rate, because many devices each with “reasonable” interrupt rate can add up to unreasonable interrupt rate on the software. Use double-buffering, block transfers, high-water interrupts.

3.4 Software Architecture, Design, and Verification

1) *What architectural concepts do you advocate to better manage flight software complexity? What experiences or lessons motivated these architectural concepts?*

- Component-based architecture, with components as the element of reuse. Configuration manage the component specifications and API’s to manage costs of evolution, testing, etc.
- Model the behavior of key aspects of the system (state transitions, timing) to avoid finding out what doesn’t work after it’s built.
- Don’t dictate architecture to designers. Focus on the bottom line: delivery of an independently testable, verifiable product.
- Have one architectural leader.
- Establish and stick to patterns.
- Assign one developer to each module.
- Limit interactions between modules.

2) *How have these architectural concepts paid off (or are expected to pay off)?*

- All comments in this section said that all the approaches in the previous section have paid off as expected, be it time or money or risk or functionality – Ed.

3) *What choices have you made to improve separation-of-concerns? How have you modularized things to contain the impact of change to one module, or a few modules?*

- Components and connectors.
- Highly portable OS abstraction layer, proven portability to a number of operating systems so reusable components have stable OS API so component itself does not need to be ported to different platforms.

- Good software tends to be partitioned like the hardware: A module/component per hardware box.

4) *Multi-threaded code is harder to get right. What policies or guidelines, if any, have you established?*

- Careful management of interactions with data items.
- Awareness of possibility of race conditions.
- Blocking calls are risky.
- Keep it all as deterministic as possible so schedulability is known.
- Use process-oriented model, not threads, with IPC between processes (ARINC-653 is limiting case of this model).
- Limit the interaction mechanisms between threads.
- Use fixed priorities, not dynamic, for schedulability.
- Have time overrun detection on the threads/processes.

5) *What kinds of complexity have made software verification much more difficult, and what strategies have you used (or advocate) to address it?*

- Multiple threads interact in ways that greatly increase number of cases that must be considered, may not even be able to test all the cases.
- Exchanging information between multiple cyclic elements on separate clocks (always eventually leads to occasional two or zero packets per cycle) and can fall into many phasings, each which must be tested.
- Simulation and fault injection used to test interesting cases.
- Testing on multiple simulations, each from different stakeholder team interpreting the ICD proved valuable by identifying different assumptions made by the different teams.
- The more connections/interactions, the harder to verify.
- Global data accessed from multiple modules increases difficulty of verification.
- Squeezing into small footprint/timeprint results in system more difficult to verify.
- Time/space margin makes it easier to verify.
- If complex fault protection behavior implemented with limited core FP capability (e.g. just persistence counters) resulting system is difficult to verify.
- System should have built-in awareness of process dependencies so it can dynamically adapt the schedule as necessary without an offline schedulability analysis.
- Lack of a good simulator hinders verification.

6) *If mission requirements continue to "raise the bar" (become more complex), what changes do you foresee being needed to be able to deliver dependable flight software? (Think of managerial strategies as well as technical strategies.)*

- Need an architecture (or architectures) that manages the complexity. Minimizes incidental complexity.
- Need quality test facilities.
- Need more capable hardware platforms.
- Need software reuse: Solve recurring needs (e.g. bus interfaces, schedulers, …) so we can focus on mission specific capabilities.
- Need hardware reuse: Evolvable hardware architectures that don't disrupt the software with each step in evolution.
- Develop software product lines.
- Baseline requirements early and minimize their churn so we can spend more time on moving forward less churning.
- Develop processes that cause design to converge without change boards etc injecting disruptions.
- Build reusable tests.
- Code generation and generate unit tests at the same time. Bonus is makes the target language (C, C++, Java…) less of an issue.
- Build software simulator testbeds that run on the desktop so each developer can run lots of tests at all levels.
- Close the loop on fault protection (keep working towards mission objectives, don't just safe and call home).
- Sequence-oriented execution is not a good match for the fundamentally asynchronous physical reality.

4 Findings

If there could be only one finding, it would have to be this: the primary source of incidental complexity is failure to apply the right mix and measure of known engineering principles, and the solution is to properly apply those principles. That said, the formulation of the interview questions does not allow determination of the belief of the interviewees that the proper measure can be applied within traditional cost and schedule allocations. In other words, is it necessary to throw money at complexity, or can it be successfully managed with smarter allocation of typical project funding? There is significant evidence in the literature that costs increase exponentially with essential complexity and with size [4]. The more complex the project, the larger portion of the overall budget will have to be allocated to managing complexity.

It's important to note that something might be very complex to one person, but much less so to another. Something might be very complex to a person, but less so when that same person has more experience. Similarly, something might have a high complexity averaged over members of a community, and less complexity as the community gains experience. The point of this digression is that the methods to reduce complexity at any point in time (holding "people" constant) are not the same methods that one would use to mature a community, in order to reduce the "complexity constant" itself. The ultimate findings of this study should address both. Where does abstraction fit into this model? It takes good people and a mature community to develop the right abstractions, but they improve the efficiency of "constant people" people by hiding it so it doesn't have to be understood.

Recall that essential complexity entails a certain level of implementation complexity, i.e., there is a lower bound on the implementation complexity of a given essential complexity. Implementation adds its own complexity on top of that theoretical—but probably unrealizable—lower bound. That complexity is called incidental complexity. The responses as to how to limit the incidental complexity covered a wide range of methods, which are described in the body of the report.

The second finding is that fault protection requirements are a growing source of both essential and incidental complexity. The difficulties of implementing and testing fault protection logic suggest that some new thinking may be in order. In fact, this was the motivation for the NASA Planetary Spacecraft Fault Management Workshop that met in April 2008, as reported in Appendix F. An example of new thinking about fault protection appears in Appendix G, reinforcing the importance of architecture in minimizing incidental complexity.

The majority of the methods for limiting/managing complexity can be portioned into three groups: architecture, reuse, and management. These categories can be seen in the findings and recommendations described in the main report.

5 References

[1] List of Cognitive Biases, Available at http://en.wikipedia.org/wiki/List_of_cognitive_biases, Accessed February 26, 2008.

[2] H. Forsberg, “Certification Issues in Avionics”, Available at http://www.artes.uu.se/industry/041123/ARTES041123_pv.pdf, Accessed February 5, 2008.

[3] V. Basili, Qualitative software complexity models: A summary in tutorial on Models and methods for software Management and Engineering, IEEE Computer Society Press, Los Alamitos, California, 19080.

[4] COCOMO II model for software cost estimation. http://sunset.usc.edu/csse/research/COCOMOII/cocomo_main.html

Daniel Dvorak is a Principal Engineer in the Planning & Execution Systems section at the Jet Propulsion Laboratory, California Institute of Technology.

Kirk Reinholtz is a Principal Engineer on staff with the Flight Software and Data Systems Section at the Jet Propulsion Laboratory, California Institute of Technology.

Final Report

NASA Study on Flight Software Complexity

Appendix C — Flight Software Complexity Primer

Editor: Steve Williams,
Applied Physics Lab,
Johns Hopkins University

Contents

Executive Summary

This document lists some general areas where flight software, fault protection, and operations can become complex, with some specific examples in each area. The examples were provided by individuals at JPL, Goddard and APL. It's not the intention of this document to state that each example represents an instance of unnecessary complexity. Rather, the intent is to provide examples of the types of issues that cause complexity to grow, so that future missions are aware of these issues, and can perform their own trades to determine if the complexity is necessary or not.

Lest anyone think that this document just represents flight software blaming others for making software more complicated, it contains a section that discusses the impact of flight software on Mission Operations. The document lists several examples where flight software design and decisions not to implement flight software functionality have made life more difficult for Mission Operations.

The purpose of the complexity stories presented below is to raise awareness of how certain decisions can increase complexity, whether in software or testing or operations. The stories should not be construed as criticism of anyone involved in those missions. The key point is that seemingly reasonable decisions in one domain can have surprising consequences in another domain. This primer is a small attempt to educate so that we can keep those surprises to a minimum. Undoubtedly, much more can be written on this subject.

1 Science Requirements

Science requirements can have a large impact on flight software, flight hardware, ground software, and operations. It is important that these requirements be documented and justified.

1.1 Science Requirement — Examples

An unnecessarily demanding requirement on "science data completeness" (say, 99% where 90% would suffice) may force the implementation of excessively complex Fault Detection and Correction (FDAC) software to enable the spacecraft to "fly through" an anomaly occurring during science data collection. The science data completeness requirement may also lead to extremely complex hardware architectures, with hot back-ups and extensive cross-strapping. Of course, the FSW developed to utilize this extremely robust hardware architecture will be correspondingly complex. Note that mission duration can also be a driver for implementing a very complex hardware architecture (i.e., a 10-year mission has to provide more hardware redundancy than a 3-year mission).

By the same token, extremely high observing efficiency requirements will impose very complex scheduling functionality on ground system software so as to milk every possible minute of observing time out of an orbit. This in turn may require the development of equally complex FSW to carry out a series of highly-coupled precisely time-sequenced activities in order to realize the complex timeline produced by the ground's scheduling software. Note that the theoretical maximums for Low Earth Orbit (LEO) missions will be much lower than that attainable for other orbit geometries such as Lagrange Point orbits because of observing time loss arising from LEO once-per-orbit Earth occultation. As a result, there is more impetus to develop highly efficient/complex schedules for LEO missions to avoid wasting any potentially available observing time after those time swaths excluded by orbital geometry have been subtracted.

1.2 Science Requirement - Lessons Learned

Any requirement that specifies a figure of merit can have a large impact on software complexity because complexity is a nonlinear function of many interacting parameters. There's a huge difference in complexity between achieving 90% completeness versus 99% completeness. Every requirement should be well justified with a documented rationale, and that requirement should be challenged if it is found to be difficult to achieve. Good questions to ask are "why that particular figure of merit?", "why that particular number?", "what is the minimum acceptable?", "what happens if that requirement is not achieved?", etc.

2 Instrument and Sensor Accommodation

The spacecraft bus hardware and software generally accommodate any unique features of science instruments or sensors. This often adds to their complexity.

2.1 Instrument and Sensor Accommodation - Examples

A data interface to instruments on one mission was selected based on the fear of the instruments teams that handling CCSDS command and telemetry packets would be too much of a load on the relatively small processors in the instruments. In actuality, the extra load on the processors would have been minimal compared to the command and telemetry formats that were actually used. The spacecraft bus software had to add a large amount of new code to support these new data formats.

On a separate mission, all instruments but one accommodated the existing CCSDS command and telemetry interfaces. This instrument, because it was “off-the-shelf”, required the spacecraft to accommodate its interfaces. This involved custom hardware and a considerable amount of new software. It was simply assumed that these accommodations were better made on the spacecraft side of the interface, without any trade study to see if that was actually the case.

2.2 Instrument and Sensor Accommodations – Lessons Learned

Having the spacecraft bus accommodate the unique features of sensors and instruments is often the correct choice. However, these choices are usually made without adequate trades to determine if overall system complexity can be reduced by making changes to the instruments and sensors. Programs should make sure that these trades are performed, and not just assume that the spacecraft bus is where all of the accommodations should be made.

3 Inadequate Avionics Design

Inadequate hardware resources are probably the most common complaint of flight software developers.

3.1 Inadequate Avionics Design - Examples

Inadequate memory resources for FSW code can result in complex, multiple uses of the same code for different applications. For example, there may be elaborate front-end processing to shoe-horn a series of applications into a common algorithm implemented in a single module instead of implementing a few, less-complex customized modules. Because the applications are being force-fit into an inappropriate module, there may be inferior performance as well as unnecessary complexity.

Instrument flight software is often a victim of decisions to use an old hardware design in order to save money. Ironically, solving the problems inflicted on the FSW may cost more money than that saved by using the bargain-basement hardware. For example, the hardware selected may have a poor tool chain. In one case at GSFC, the FSW team had to write its own C compiler in assembly language. In another case, insufficient RAM was provided and to compensate for the hardware shortfall the FSW team had to design in a paging scheme. And of course, if the processor is slow (relative to computers that could have been utilized instead), designing FSW that is capable of meeting timing requirements on the real-time processing may be very difficult.

Use of an old hardware design on two missions to save money also has meant the use of old, unsupported tools and operating systems. This has left the FSW team with the responsibility of supporting these tools on their own, and without any recourse when problems are found. Older compilers tend to do less checking than newer ones, and problems that should have been flagged by the compiler were not.

The decision to “give back” RAM during initial hardware design left one program with virtually no reserve (<8%) at launch. This decision was made very early in the program, before FSW requirements and design were mature and before several large users of RAM were identified. The FSW team has to be conscious of this lack of RAM every time it entertains new changes, and it’s left the team vulnerable in the event they would need to add code to compensate for a hardware failure.

3.2 Inadequate Avionics Designs – Lessons Learned

In a cost-constrained environment, it is tempting to always want to re-use existing hardware from the last mission. This may in turn facilitate the re-use of flight software. However, this can come at a cost. Slow processors and not enough memory can make flight software more complex. Software development on processors only supported by less capable tools is inherently more risky than development on processors supported by better tools. Decisions to re-use older hardware or to limit the use of resources such as memory must take into account the negative consequences to flight software of doing so. Flight software, for its part, must design its software to be hardware independent, so that upgrades to hardware don’t require wholesale software changes.

4 Hardware Interfaces and FSW Complexity

Just as the overall avionics design can increase FSW complexity, the details of hardware interfaces and the number of different types of interfaces can greatly influence flight software complexity.

4.1 Hardware Interfaces and FSW Complexity — Examples

If an instrument has a moving component that has a significant amount of mass, moving the component can introduce significant attitude jitter. If the instrument/bus interface includes the capability to warn the bus when the mechanism will start rotating and when it will stop, the FSW's control law can utilize a feed-forward signal to compensate for the movement as it happens rather than responding to the movement after it happens. A simple feed-forward model used in this fashion can yield both a less complex overall FSW implementation as well as improved attitude control performance.

If an instrument repetitively generates new measurements at a high frequency, and overwrites the previous measurement on each cycle (no buffering), then that can levy difficult-to-achieve timing requirements on flight software. A simple change to the hardware interface, such as buffering, can significantly reduce timing demands.

4.2 Hardware Interfaces and FSW Complexity — Lessons Learned

The lesson learned here is fairly simple, namely, that software be made aware of and has the chance to review all hardware interfaces that it will need to support. Interfaces that are conceptually simple can still be costly and time consuming to implement and test. Such costs not only show up in the flight software, but also in simulator hardware and software, and in integration and test. Reducing the number of unique interface types is always a good idea.

5 Miscellaneous Hardware Issues

Flight software, fault protection and operations are all affected by either the need to protect against hardware faults, hardware that is inappropriate for the application, or hardware that adds complication to the overall system.

5.1 Miscellaneous Hardware Issues — Examples

The basic timing reference to the software on several missions is a hardware-generated one Pulse Per Second (PPS). The software includes logic to protect against the case where this hardware malfunctions in such a way as to produce multiple pulses per second. This step was taken without any trade or investigation to determine if this is a realistic way for this hardware to fail, and what the probability is of such a failure. Is the hardware likely to fail in such a way that multiple PPS outputs per second are produced, or is it likely to fail in such a way that the PPS output just disappears?

Many companies have a spacecraft bus design that includes a hardware-based fall-back known as an "emergency mode controller." It's a common strategy for supplying basic functional redundancy that works well in earth orbiters where one can expect the ground to intervene very quickly. For deep space missions with longer inter-contact times and higher levels of functionality needed to maintain basic health, it's not such a great strategy. During an audit of one spacecraft design, the interaction of their hardware-based emergency mode controller and the fault protection software was investigated. Due to race conditions between hardware and software, it was unknown how certain fault scenarios would play out.

One mission had catalyst bed temperature sensors whose A/D range did not bracket the entire cold-to-hot operating range of the cat-beds. In fact, it couldn't even measure down to what was considered the minimum allowed temperature. In the end it was impossible to monitor for the frozen condition, requiring the mission to change their ops approach and several on-board sequences.

One family of missions derived from a past mission in which there was insufficient power margin to allow simultaneous powering of both hardware strings. Therefore the design included a hardware interlock to prevent that condition. A recent mission using that architecture encountered a fault, resulting in a string swap. Because the interlock prevented powering on the backup string, the mission could *not* gain access to data that would have explained the anomaly. Weeks later the anomaly repeated on the new string and the system swapped back, but the old string did not retain useful data through a power cycle.

One mission made use of a new solar array design that was more efficient, but also more sensitive to sun angle. The fault protection design was made more complicated by having less margin for sun pointing performance.

A recent mission flew an exotic propulsion system that required a correspondingly complex power system. Several new and novel fault protection algorithms had to be added to the original project scope to detect new failure modes in the propulsion and power system combination.

5.2 Miscellaneous Hardware Issues — Lessons Learned

Where complexity is added to the system to deal with hardware faults, a study must be done to make sure that the faults are credible, and that the software being added to mitigate the faults is not so complex that the overall system becomes less reliable, and is actually protecting against a credible fault.

In several of the above examples, hardware was re-used from previous missions, but features of the hardware made it not well suited to the newer missions, resulting in complications for fault protection and operations. Missions must make sure that re-used components, either hardware or software, are appropriate for their application. Components are re-used in order to save costs, but if fault protection and operations become more complicated in order to accommodate these components, cost savings may not be realized, and the health of the mission can be compromised.

When components are added to the mission that improves capabilities in one area, this additional performance must be traded against the possible impacts on other areas. This is simply performing good system engineering.

6 Fear of Flight Software

A general distrust of flight software can sometimes result in a decision to duplicate certain flight software functionality elsewhere in the system, either in other software or in hardware. This can add to overall system and operational complexity.

6.1 Fear of Flight Software — Examples

Flash/EEPROM memory programming is conceptually a fairly simple activity, being no more complicated than many other things the flight software already does. Several missions have chosen to add an additional command implemented in software to directly control the flash memory programming voltage, even though this voltage is controlled automatically by the software as part of its flash memory programming. One program requires specific hardware enabling of the ability to write to EEPROM, in addition to a command to the software to do so. This adds both hardware and operational complexity.

On some missions that have a Guidance and Control (G&C) processor that performs closed-loop attitude control, software has been placed in the Command and Data Handling processor to perform open-loop control based on commands from the ground. This capability is used for Trajectory Control Maneuvers (TCMs), and represents a duplication of functionality already in the G&C processor.

One program stored a sequence of critical commands in hardware, even though the capability to issue these commands existed in software. The hardware implementation of the commands forced the sequence to be finalized much earlier than it otherwise would have been, and didn't allow any changes to the sequence post-launch.

If Mission Operations does not trust the FSW to do its job without hand-holding, you end up spending a lot of resources to develop and validate the onboard function but then still implement the same capability in the ground system along with a process that enables the ground to check the onboard version against the ground version. Clearly in such cases it's best to allocate the function to the ground, or look for a technology demonstration mission to develop confidence in the onboard implementation.

Similarly, relegating increased functionality to the ground can actually increase complexity onboard. In order for the ground to do a lot of hand-holding by having people carefully watch the spacecraft state on consoles, the FSW must generate all the telemetry points that will be down-linked to be displayed on the consoles. The FSW will probably also have to do additional processing of the telemetry points to generate the input that will go into event messages designed to draw attention to the things of particular interest to the console staff. For example, a whole series of largely artificial stages may be created to describe the transition from one control mode to another (say, slew to fine-pointing) to enable the Mission Operations team to follow the onboard activity on a step-by-step basis.

6.2 Fear of Flight Software — Lessons Learned

Systems Engineering and/or Mission Operations sometimes fear the software's implementation of specific functions. This fear is sometimes a function of past experiences of a member or members of these teams. This has resulted in a number of instances of an increase in system and software complexity, and an increase in operational complexity. Situations where functionality is duplicated elsewhere in the system because of a fear that the original implementation of the functionality isn't reliable need to be evaluated to see if such fear justifies the added complexity.

7 Design for Testability

A flight software architecture that makes it cumbersome and time-consuming to place the software in a state to support specific tests will likely result in fewer of these tests being run, increasing program risk.

7.1 Design for Testability — Examples

Many test cases are focused on specific issues and span a brief period of time, so in principle the whole test could be completed in a few minutes. However, for several spacecraft, to test out *anything* you essentially have to "launch" the spacecraft, go through de-tumble, acquisition, etcetera in order to get the flight system into the desired state for the test. The setup procedure may take 3 hours, and that gets repeated for every test! The result is that far less testing gets done, and defects that could have been caught before launch manifest during operations, when it's much harder to diagnose.

In a similar vein, on one mission there was no way to jam-load the state of the attitude estimator, so every test suffered through the 20-30 minutes it took for the estimator to converge. Again, the net result was that testing was more time-consuming and costly, all for want of a straightforward feature in flight software.

7.2 Design for Testability — Lessons Learned

The nature of the testing the software needs to support needs to be thought of early in the software development process, and the software must then be designed to easily support these tests. Testing is the most important means of raising confidence in a flight system and its operations, but testability is often ignored as a design consideration.

Testing is sometimes treated as an afterthought at reviews, especially those held early in a program. The focus is on requirements and design, with little thought given to testing. Sometimes, those teams who will be most involved in testing — such as acceptance testers, Integration and Test, and Mission Operations — are sparsely staffed at such early reviews. These teams need to be fully represented at these reviews, so that their testing needs can be incorporated in the flight software design.

It must be possible to set a consistent internal state in a straightforward way for the needs of a particular test, or at least have a few selected starting super-states that could serve the majority of tests. It must also be possible to make FSW "jump forward in time" to avoid wasting precious testbed time waiting for a future event.

8 The Effects of Flight Software on Mission Operations

Flight software has an enormous affect on Mission Operations. Decisions not to add flight software features, or not to fix flight software bugs, can make the operation of a spacecraft far more complex. Conversely, if the flight software is designed with operations in mind, it can make operations much simpler.

8.1 The Effects of Flight Software on Mission Operations - Examples

Appropriate addition of onboard autonomy can significantly reduce the complexity of operations. For example, a real-time onboard target acquisition function that has the capability to maneuver the spacecraft onto the science target relieves the ground of the responsibility for real-time ground-commanded attitude maneuvers. This can be a particularly stressful and potentially risky operational activity for LEO missions because of the absolute time nature of occultation patterns and communication outage cycles.

On a mission that included risky and complex Delta-V maneuvers on two spacecrafts right after launch, the FSW team designed the software that supported these maneuvers with ease of operation in mind. This resulted in a solution that ultimately required one to two orders of magnitude fewer commands than on other spacecraft. The Mission Operations team could then spend more of its time responding to the few spacecraft anomalies that occurred in this time, rather than having to spend a lot of time constructing complex command sequences.

As a converse to this, although autonomous “do-it-all” commands are good, mission operators need a complete set of low-level commands for troubleshooting and testing. As a case in point, one spacecraft provided an ‘autonomy’ command for imaging that required ~40 arguments, and it was the *only* command for pointing the camera and getting images. However, in a testbed environment where all the operator wanted to do was move the camera, he/she still had to supply all 40 arguments.

On one mission, it was proposed that that a small file system be created for the solid state recorder (SSR) to simplify data management, given the need to manage science data from multiple instruments. However, the recommendation was considered “too late” to get it into FSW, so the SSR was flown without a file system. The unfortunate result during operations was that science people were overwriting each other’s data on the SSR. The project held meetings of 10-15 people, including many scientists, for approximately a year to solve problem. The far-from-ideal solution was to give each instrument an allocation and have FSW count bits from each instrument and discard anything exceeding the allocation. Scientists *could* have spent their time more productively on science than on the allocations. The lesson here is that a modest investment in FSW would have paid off in simplifying operations and reducing total mission cost.

On another mission the phrase “an operational workaround exists” was often used to reject ideas for added functionality in FSW. Operations engineers identified many problems well before launch but were not listened to because the project was trying to save on cost. The result was that many different problems which *could* have been addressed in FSW were pushed onto flight operations. It’s hard to put a price tag on workarounds because they appear one-by-one, and each one in isolation is not a big issue, but they add up to a large operational burden that increases risk in two ways. First, each workaround has to be remembered and applied at the right time; forgetting one can be disastrous, and will be labeled “operator error”, though the “error” is more a symptom than a cause. Second, workarounds distract the operations team from vigilant monitoring of the spacecraft. Time spent on workarounds is less time spent on monitoring small forces, examining flight data, and generally looking at the big picture. One operator suggested that 20% of operator time be unallocated to give them time to look at big picture and do high-leverage risk reduction.

It was pointed out that as operational requirements kept increasing (in the form of workarounds), no money was added to the ops budget for staff or tool support. Every project is in crisis mode and they don’t have time or budget to fix operational problems at the source (in FSW), so they throw the problems over the wall to operations. Even if the project increased the ops budget (which they don’t), adding ops staff late in the game is not an answer because it takes ~2 years to learn the nuances of a spacecraft and its flight software behavior and operational procedures. The net result is that ops risk goes up with each workaround, and it’s hard to quantify that risk and compare it to money saved in development.

Here’s a story with a happy ending, but it provides another example of how a proposed de-scope in FSW would have had a huge impact on operability if it hadn’t been pointed out and reinstated. Specifically, the FSW team working on the command sequencer for a mission proposed to get rid of multiply, divide, and indirect addressing operators because they were behind schedule. An operator recognized that this decision would have a huge impact on science, because onboard command blocks would be virtually useless. In order to convince the project, she had to go to the FSW systems engineer and ask “how would you handle this science operations scenario without multiply and divide and indirect addressing?” The answer was that it was impossible, so the extra functionality was kept on the schedule. The lesson here is that operability considerations need to be part of the review process for FSW designs and proposed de-scopes.

Telemetry design is important to operations, and poor design can lead to lot of complexity and larger amounts of downlink traffic. For example, one spacecraft used a 30-year-old fixed channel design. If an operator wanted to know the status of 8 devices, it came down in 8 16-bit words instead of one 8-bit byte. FSW designers often bundle data in one way, but operations often needs it in different groupings. An operator cited another outrageous example from a spacecraft where, in safe-mode, the spacecraft did *not* include the command counter in downlink telemetry, so there was no way to know if commands were getting into the spacecraft! On missions that allow flexible construction of telemetry packets, it was pretty easy for operators to specify what they wanted and have it come down at a certain rate or upon change. The lesson here is that operators need a flexible way to specify desired telemetry because it is very hard to predict what data will be most important at different times. They need the ability to see any data at any time, not all the data all the time.

On one mission, low downlink bandwidth during hibernation and safe modes and the long lengths of power system telemetry packets made Mission Operations reluctant to include these packets in the downlink. This led to less than desired visibility into the power system at these times. The availability of a small amount of selectable telemetry in one of the spacecraft housekeeping packets allowed some of the most critical power system points to be telemetered.

In addition, a shorter power system packet has been added in a post-launch build. Closer communication between Mission Operations and flight software in the development phase concerning telemetry requirements in the various modes might have resulted in this functionality being available at launch.

8.2 The Effects of Flight Software on Mission Operations — Lessons Learned

Trades between adding flight software functionality and attempting to work around a lack of functionality in operations are difficult, but they must be made. It is tempting for programs with cost and schedule pressures to opt not to include the software functionality, but the short term software costs must be balanced against long term increased operations costs, and increased risks to the mission. The risks and costs of making software changes are generally understood, but the risks to spacecraft health and safety of attempting to operate the spacecraft without needed capability are difficult to quantify and can be overlooked.

Mission Operations teams may not be fully staffed early in the program, and thus may not recognize the flight software design decisions that will affect their operation of the spacecraft. Incorporating design features in the software to improve operability is much easier done, and is less costly, when these features are designed into the software from the start. This requires early Mission Operations involvement. Flight Software teams want to make their software easy to operate, but without direct inputs from Mission Operations, they are often guessing as to how to do so.

Final Report

NASA Study on Flight Software Complexity

Appendix D — Software Complexity[1]

Gerard Holzmann,
Jet Propulsion Laboratory,
California Institute of Technology

[1] An abbreviated version of the text that follows, cleared of NASA/JPL specific data, appeared in IEEE Computer, December 2007, pp. 102-104.

Contents

Executive Summary

In a 1984 book,[2] sociologist Charles Perrow wrote about the causes of failure in highly complex systems, concluding that they were virtually inevitable. Perrow argued that when seemingly unrelated parts of a larger system fail in some unforeseen combination, dependencies can become apparent that could not have been accounted for in the original design. In *safety critical* systems the potential impact of each separate failure is normally studied in detail and remedied by adding backups. But failure *combinations* are rarely studied exhaustively; there are just too many of them and most of them can be argued to have a very low probability of occurrence. A compelling example in Perrow's book is a description of the events leading up to the partial meltdown of the nuclear reactor at Three Mile Island in 1979. The reactor was carefully designed with multiple backups that should have ruled out what happened. Yet a small number of relatively minor failures in different parts of the system (an erroneously closed valve in one place and a stuck valve in another) conspired to defeat the protections and allowed the accident to happen. A risk assessment of the probability of the scenario that unfolded would probably have concluded that it had a vanishingly small chance of occurring.

[2] Charles Perrow, *Normal Accidents: Living with High Risk Technologies*, Princeton University Press, 1984.

1 Software Failures

What lessons can we draw from Perrow's analysis in the construction of complex safety critical *software* systems? In the Three Mile Island accident no software was involved. Today, we do not have to look very far to find highly complex software systems that perform critically important functions: think of the phone system, fly-by-wire airplanes, or manned spacecraft. To stick with spacecraft, the amount of control software that is needed to fly space missions today is rapidly approaching a million lines. For the new manned space program we should probably expect close to 10 Million lines of flight code, and one to two orders of magnitude more ground software. If we go by industry statistics, a really good (but expensive) development process can reduce the number of flaws in software to in the order of 0.1 residual defects per one thousand lines written. A "*residual* defect," for all clarity, is a defect that shows up *after* the code has been fully tested and delivered with an industry-standard rigorous testing process.[3] A system with a million lines of code, then, should be expected to experience at least 100 defects while in operation. Not all these defects will show up at the same time of course, and not all of them will be equally damaging.

As a rule of thumb, one sometimes assumes that the number of defects shrinks by an order of magnitude from one severity class to the next. This approximate relation seems to match data on residual software defects that has been collected for some of our larger unmanned space missions (e.g., the Cassini mission). Following this rule, 90% of all residual defects can then be expected to be benign and easily worked around. 90% of the remainder can be expected to be of medium severity, and only the final remainder would then be classified as major. If we start with 100 residual defects, for one million lines of source code, this then means that one or two of those 100 defects would fall in this final group. Clearly, if the linear relations continue to hold (which is of course only an assumption), an increase of the software base to ten Million lines of code would increase the number of potentially fatal defects to at least ten.

[3] The number of *residual* defects is not the same as the *total* number of defects in the code – it only counts the defects that actually show up. The larger class is often referred to as *latent* defects.

2 Bugs and Valves

Knowing that software components can fail does not tell us any more than that a valve can get stuck in a mechanical system. As in any other system, this is only the starting point in thought process that should lead to the design of a reliable system.

Reliability is, of course, a *system* property, and not a property that can be provided by any single system component, not even if that component is, say, the flight software.

We know that the effects of failures can be mitigated by adding fault protection and redundancy to a system. But this is where the findings from Perrow's analysis become especially interesting. By adding backups and fault protection, we inevitably also increase the size and complexity of the system, and we may unwittingly be adding new failure modes. We may be introducing unplanned couplings between otherwise independent system components. Backups are typically designed to handle *independent* component failures. In software systems, they can be used to mitigate the effects of individual software defects that could be mission-ending if they strike. But, what about the potential effect of *combinations* of what otherwise would be minor failures? Given the magnitude of the number of possible failure combinations, there simply is not enough time to address them all in a systematic software testing process. Just 10^2 residual defects, for instance, might occur in close to 10^4 different combinations (10^2 x (10^2 – 1)). This then opens up the door to a "Perrow class" accident, but this time in software. As before, the probability of any one specific combination of failures will be extremely low. But, as experience shows, those are precisely the types of things that lead to major accidents. In fact, almost all software failures that have been experienced in space missions can be reasoned back to unanticipated combinations of otherwise benign events. Quite possibly this issue, then, should be our biggest concern in the construction of ever more complex software systems. So what can we do about it?

Examples of this phenomenon are readily found. The loss of contact with the Mars Global Surveyor spacecraft is perhaps the most recent example that has all the elements of a Perrow-style failure.

First, an update of some flight parameters reached only one of the two computer strings, when the second string was unexpectedly down. This was a very minor problem, since both the updated parameters and the earlier versions were perfectly usable. A small correction to this problem was planned, but for one of the parameters (the soft-stop parameter for the solar arrays) the new value ended up being written to the wrong memory location. Normally this would also be a minor problem and easily detected. It so happened, though, that the new memory location happened to contain another key system parameter that now ended up being corrupted. That parameter, though, was rarely needed. What had gone wrong so far could easily have been caught in routine checks at any point later. Several months after these events, without the routine checks having been performed yet, the solar arrays were adjusted from their summer to their winter position – again a very minor routine operation that is done twice each year without any problems. In this case, though, the adjustment triggered a fault, which was caused by the incorrect value for the soft stop parameter. The fault triggered safing. This was the correct response, given that an off-nominal condition had been detected that should now be checked and corrected by ground intervention. Only a sequence of relatively minor problems had occurred so far.

The top two priorities for the spacecraft in safing mode were to be power-positive (i.e., to make sure that the batteries were charged) and to communicate with earth. It could not do both at the same time, given a perceived problem with the solar arrays (a conservative approach, given that the solar arrays had reach a hard-stop unexpectedly). Pointing the solar panels (presumed stuck) at the sun also pointed the batteries at the sun though – something that had not been anticipated, and was caused by the hidden coupling of safing priorities and the perceived failure mode of the solar panels. As a result, the batteries overheated, which the fault protection software then misinterpreted as a signal that they were overcharged. This is still not a major problem, until it combines with still another relatively small problem. Communicating with earth requires pointing the antennas at earth, but that required access to the one extra parameter that had been corrupted in the original update. Now the cycle is complete: a series of relatively small problems now lined up to cause a big problem that prevented the spacecraft *both* from communicating with earth *and* from charging its batteries. Within a matter of hours led to the loss of the spacecraft. Taking away any one of the smaller problems would have prevented the major problem. What makes this example extra interesting is that some of the dependencies were introduced by the fault protection system – which itself functioned precisely as designed. The part of the design that was meant to prevent failure helped to bring it about.

3 Big Failures Often Start Small

If major failures tend to use minor defects as stepping stones to move towards disaster, then one remedy will be very clear: we should try hard to take away as many of the smaller stepping stones as possible. Minor software defects typically have no chance of causing outright disaster when they happen in isolation, and are therefore not always taken very seriously. Perrow's insight is that if we can reduce the number of minor flaws we will also be reducing the chances for catastrophic failures in unpredictable scenarios.

It is tempting to compare this approach with the crime fighting strategy that was successfully used in New York and other large cities in the nineties: aggressively fighting small crime to eliminate the conditions that can lead to, or create the environment for, bigger crime to flourish. In software we may achieve the same effect by being extra thorough about even the smallest and seemingly benign software defects – making sure that they never get the chance to combine in unforeseeable ways to trigger larger problems.

We can reduce the number of minor defects in several ways. We can, for instance, adopt much stricter coding rules than are normally used. A good example is the set of rules that JPL is experimenting with, known as the "*Power of Ten Rules*" (IEEE Computer, June 2006, pp. 93-95). Recommended as part of the "Power of Ten" is the use of strong static source code analyzers (such as Grammatech's CodeSonar, or the analyzers from Coverity or KlocWork) on *every* build of the software, from the start of the development process. Another equally important strategy, they may seem too simple to be of much value, is to require developers to *always* compile *all* their code with *all* warnings enabled at the highest warning level available. (In some compilers this is called the 'pedantic' setting.). Safety critical code should pass such checks with *zero* warnings. It is important to note explicitly here that even warnings that can after thorough manual analysis be shown to be invalid will *not* be permitted. The rationale for this approach is that neither humans nor automated tools should be confused by safety critical code: it should be so clearly correct that even a tool can establish that. In almost all cases it is relatively simple to rewrite a code fragment to prevent even invalid compiler warnings. The clean-build principle is already a matter of routine for good software developers. For safety critical code it should be considered more than a preference though; it should be made one of the highest priorities in software development.

4 Decoupling

A second method to reduce the chances for Perrow-class software failures is to increase the amount of decoupling between software components: separating independent system functionality as much as possible. Much of this is already standard in any good software development process and meshes well with code structuring mechanisms based on modularity and information hiding. Many coding standards for safety critical software development (such as the MISRA-C rules used in the automotive industry) contain rules that require the programmer to limit the scope of declarations to the minimum possible, avoiding or even eliminating the use of global declarations, and strongly encouraging the use of static and constant declarations (using keywords such as `const` and `enum` in C).

As such these principles can be part of the stronger coding standards that we would like to see adopted NASA-wide for mission critical and safety critical code development.

One of the strongest types of decoupling can be obtained by executing independent functions on *physically* distinct processors, providing only very limited interfaces between them. High-end cars today already have large numbers of embedded processors on board, each performing functions that are carefully kept separate. The Mercedes-Benz S-class car, for instance, is said to contain 50 embedded controllers, jointly executing over 600,000 lines of code.[4] Many robotic spacecraft also use redundant computers – but they typically operate only in standby mode, executing the same software as the prime computer. (This holds to a large extent also for the current Shuttle computers.) When the controlling computer crashes due to a fault in either the hardware or the software, (one of) the backup computers takes over. This strategy, of course, can offer only limited protection against *software* faults. The amount of decoupling could be increased by having each computer control a different part of the spacecraft – thus limiting the opportunities for accidental cross coupling in the event of failures. When the hardware for one of the computers fails, the others can still take over its work load, so that protection against hardware failures is preserved.

[4] K. Grimm, *Software technology in an automotive company – major challenges*. Proc. Int. Conf. on Software Engineering, Portland, Oregon, 2003, IEEE, pp. 498-503.

5 Containment

Separating functionality across computers is also a good example of a *defect containment* strategy. If one computer fails, it affects only the functionality that was controlled by that one computer. Another good example of the use of defect containment in software is the use of memory protection, i.e., to guarantee that one thread of execution in a given computer cannot corrupt the address space of another. Memory protection is standard on most consumer systems, but not always used in embedded software applications. For safety and mission critical systems, the use of memory protection should be a firm requirement.

Finally, other forms of defect containment can be based on the principle that we have named "simplified hierarchical backup." In a simplified hierarchical backup design, each critical system component is implemented twice (by the same developer): the first implementation is the fully functional flight version – optimized and with all features that are desirable. The second component is a scaled down version of the first, with only the absolute minimal survival functionality – possibly not the most efficient version, and not with all features, but providing survival capabilities. The ratio in size between the first (prime) module and its backup (online) version should be roughly 10:1. This means that the online version of the code can be tested (and possibly even formally verified exhaustively) to a much higher standard of rigor than the prime version. Apart from this, the smaller online version should also by a statistical argument on residual defect rates after testing be expected to have at least ten times fewer residual defects than the prime version. Clearly, this approach to system backups can be extended from a single backup module to a series of smaller and smaller modules, each new level taking over as part of a fault protection strategy that increases the reliability of the remaining software with each new stop – to the level required to recover fully from the mishap, without risking complete system failure.

The principle of hierarchical backups is already relied upon, to some extent, in hardware designs. When the high-gain antenna fails, for instance, we switch to the low-gain antennas, which are somewhat more reliable. If the high-resolution imager fails, we switch to a lower-resolution version, etc., etc. It would take some extra steps to follow through on these principles in reliable software system designs. For safety-critical design this could well be a key part of a comprehensive strategy for defect containment that is currently missing. It should perhaps be noted carefully that this approach to software design is fundamentally different from standard approaches to N-version programming – which are typically envisioned to involved N independent programming teams, each pursuing a full design of a prime component. It has been shown that standard N-version programming does not really succeed in achieving design independence,[5] and also that it tends to increase the cost of software development N-fold – which may make that approach unaffordable.[6]

[5] J.C. Knight and N.G. Leveson, "An Experimental Evaluation of the Assumption of Independence in Multi-version Programming," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 1 (Jan. 1986), pp. 96-109.
[6] L. Sha. "Using Simplicity to Control Complexity," *IEEE Software*, July-August 2001, pp. 20-28.

6 Margin

Another common method for strengthening the defenses to seemingly minor defects is to provide more margin than is necessary for system operation, even under stress. The extra margin can remove the opportunity for close calls, where temporary overload conditions can provide one of the key stepping stones leading to failure. Margin does not just relate to performance, but also to the use of, for instance, memory. It can mean, for instance, that one does not place mission critical data in *adjacent* memory locations. Since small programming defects can cause overrun errors, we do well to provide safety margins around and between all critical areas of memory. These 'protection zones' are normally filled with an unlikely pattern, so that errant programs can be detected, without causing harm. The recent loss of contact with the Mars Global Surveyor spacecraft provides another excellent example of the need for these strategies. In this case, an update in one parameter (the soft-stop for a solar array) via a memory patch missed its target by one word, and ended up corrupting an unrelated, but critical earth-pointing parameter that happened to be located next to the first parameter in memory. Two minor problems now conspired to produce a major problem, and ultimately lead to the loss of the spacecraft. Had the key parameters been separated in memory, the spacecraft would still be in operation today.

The use of multiple layers of functionality (or less gloriously known as "work-arounds") is yet another way to provide redundancy in safety critical code. As one example, most spacecraft have a non-volatile memory system to store data. On newer spacecraft this is typically flash memory. Several independent means are typically provided to store and access critical data on these devices. A spacecraft can often also function without an external file system, in so-called "crippled mode," by building a shadow file system in main memory. The switch to the backup system can be automatic. We cannot predict where software defects may show up in flight, but we can often build in enough slack to maximize the chances that an immediately usable work-around is available no matter where the bugs hide.

7 Model-Based Engineering

Perhaps the main lesson we can draw from Perrow's discussion of complex systems is the need for scrutiny of even minor software defects, to reduce the chances that combinations of minor flaws can lead to larger failures. The strategies mentioned so far dealt mostly with defect prevention and defect containment. We've left the most commonly used strategy to last: defect *detection*. Defect detection in software development is usually understood to be a best effort at rigorous testing just before deployment. But, defects can be introduced in *all* phases of software design, not just in the final coding phase. We can improve the effectiveness of defect detection by not limiting it to the end of the process, but to practice it from the very beginning. Large scale software development typically starts with a requirements capture phase, which is followed by design, coding, and finally testing. In a more rigorous model-based engineering process, each phase in this process is based on the construction of *verifiable models* that capture the main decisions.

Engineering models, then, are not just stylized specifications that can be used to produce nice-looking graphs and charts. In a rigorous model-based engineering process, requirements are specified in a form that makes them suitable to use directly in the formal verification of design models, for instance with strong model-checking tools.[7] Properly formalized software requirements can also be used to generate comprehensive test-suites automatically, and even runtime monitors[8] that can be embedded as watchdogs in the final code, thus significantly strengthening also the final test phase. Test randomization techniques,[9] finally, can help to cover also the less likely execution scenarios that are so often missed in even thorough software testing efforts.

None of the steps we have mentioned for defect prevention, detection, and containment can be considered more important than any of the others. Even if all the engineering models used in the production of a safety critical software system are exhaustively verified, it is still wise to test also the resulting code as thoroughly as possible with standard techniques. There are simply too many steps in the chain to take anything for granted in critical software development.

[7] http://www.spinroot.com/
[8] http://www.runtime-verification.org/
[9] A. Groce, G. Holzmann, R. Joshi, Randomized differential testing as a prelude to formal verification, Proc. Int. Conf. on Software Engineering, Portland, Oregon, 2007, IEEE, pp. 621-631.

Final Report

NASA Study on Flight Software Complexity

Appendix E — On Test Strategies for Complex Software Systems[1]

Gerard Holzmann,
Jet Propulsion Laboratory,
California Institute of Technology

[1] This is part II of a two-part discussion. Part I, titled 'Conquering Complexity' contained a more general discussion of the nature of errors in complex systems. Here we consider the implications of the observations made to the definition of a test strategy for complex software systems.

Contents

1 Introduction

It is often thought that the failure of a complex system is often contributable to a single primary cause that was somehow missed in system testing. After a failure, we expect an investigation board to identify and describe this single *most likely* cause, and to provide recommendations on how to prevent its recurrence. As argued in our earlier report, these types of single-cause failures do happen, but they are more the exception than the rule. The majority of failures in complex systems are due to unforeseen *combinations* of, what otherwise would be, relatively benign flaws. What is unforeseen in this case is not the nature of each flaw but what the collective effect is of some arbitrary combination of these flaws. Given a system with E small flaws in a system, there are in general (close to) E^N possible combinations of N such flaws. Even for small values of E and N, the resulting number can be very large indeed – defeating any attempt to test all combinations exhaustively. It should also be carefully observed that the *probability* of each combination *decreases* rapidly with increasing N. This fact often gives testers and system designers a false sense of security. The problem is that the total *number* of possible error combinations *increases* rapidly with both E and N – giving a complex system an astounding number of choices to defeat all single-cause protections that have been provided. Especially for large values of E and N, each separate failure scenario will have an exceedingly low probability of occurrence, which means that it normally does not trigger significant scrutiny in system testing.

The *probability* of any one specific error combination in a complex system is almost always vanishingly small, but then so is the probability of occurrence of any one specific *correct* execution sequence. Arguments on test thoroughness that rely solely on a probabilistic argument should therefore be avoided, and if made they should be treated with strong suspicion in safety analyses. Major failures in complex system often have unlikely causes that in standard probabilistic risk assessments would be discarded with high probability.

2 Definition of a Complex System

A *complex* system has been defined succinctly as follows:

> A system is classified as *complex* if its design is unsuitable for the application of exhaustive simulation and test, and therefore its behavior cannot be verified by exhaustive testing. [2]

[2] Defence Standard 00-54, *Requirements for safety related electronic hardware in defence equipment*, UK Ministry of Defence, 1999.

3 Residual Defects in Complex Software

Statistically, for a *well-controlled* large-scale software development process, a fairly broadly accepted rule of thumb is to assume that there will be in the order of 0.1 to 1 *residual software defect*s per 1,000 lines of code written (counted after stripping comments and blank lines). Residual defects are defects that survive the review and test process and that reveal themselves only in mission operations. The size of the still larger class of *latent* defects (all defects that *could* reveal themselves) is generally unknown.

Since we cannot accurately predict where precisely the residual defects are (or else we would simply fix them), and almost by definition cannot exhaustively test for them, we should adapt our test strategy for dealing with them more effectively.

1. First, if we cannot predict where the residual defects hide, but can estimate what their likely numbers are, at least part of our test strategy should not be biased in any way towards specific sources of error. A good way to avoid bias in testing is to use a strategy that leverages **Test Randomization**.
2. Second, we should use the knowledge that some software defects will survive into mission operations, no matter how thorough our software design or test process is. Any system component can break, including any non-trivial software component. System reliability is fundamentally a *system* property, and cannot be the property of any one system component. Since residual software defects will occur with near certainty in complex software systems, there must be a strategy for dealing with them to prevent *system* failure. Strategies for dealing with residual software defects fall in the general category of **Fault Protection**. Traditionally, fault protection deals primarily with hardware faults, not with software faults. That should change.
3. Third, we should develop strategies to reduce the number of ways in which residual software defects can *combine* to produce complex error scenarios against which the system typically has no or only weak defenses. Strategies for dealing with this phenomenon fall in the general category of **Fault Containment**.

We will discuss the implications of our observations on these three concepts in more detail below.

4 Test Randomization

Test randomization was once dismissed as “probably the poorest methodology.”[3] More recently, though, this technique has gained prominence as a powerful strategy for testing complex software systems as well as smaller critical software components that defy formal proof. The basic technique is known under a number of different terms, e.g., as ``fuzz testing,’’[4] ``stochastic testing,’’[5] and ``probabilistic testing.’’ There is even an annual workshop devoted to the topic of test randomization.[6]

In Miller et al’s study from 1990, a total of 90 standard and very widely used standard applications were tested on six different versions of the Unix® operating system. Each application was tested with randomly generated input, leading to the remarkable result that approximately 30% of these applications could be made to crash or hang. The causes of the failures included out-of-bound array indices, null-pointer dereference errors, divide-by-zero errors, ignored return values of function calls, and unbounded loops. A repeat of the experiment in 1995 showed in one case that 47% of all applications that used dynamic memory allocation could be made to crash, due to the basic failure of applications to check the return value of calls to the `malloc` routine for an out-of-memory condition.

McKeeman[7] used the technique at DEC in 1998 to test the robustness of a new compiler, using randomly generated but syntactically valid inputs, with similarly impressive results. In 2006, the same basic technique was applied by us in JPL’s Laboratory for Reliable Software[8] to test the robustness of mission-critical flash file system components, again with very impressive results that compared well with even the rigor of fully formal logic model checking approaches.

Randomized testing is best used as a standalone, fully automated, procedure, with a test generator generating inputs for the system under test. The input is generally not completely random, since this would restrict the effectiveness of the test to mostly invalid inputs, but is constructed as a deliberate series of random variations of nominal execution scenarios. Variation may be produced by a random selection of a valid, but perhaps unusual, selection of input values to function calls (such as negative values or zero values where normally positive values are processed), random re-orderings of otherwise valid commands, random omissions in command sequences, or the insertions of seemingly nonsensical commands or function calls with out of range parameters or no longer valid parameter choices. A most effective method is also to include an explicit emulation of the hardware environment in which a software system or component is to execute, and to introduce random fault injection within that simulated environment. It should be observed that the test sequence itself, though generated as a random variation of what should be a valid execution sequence, is fully deterministic. If the test fails, the error scenario can be replayed from start to finish by re-executing the same input sequence under

[3] G.J. Myers, *The art of software testing*, New York, 1979.
[4] B.P. Miller, L. Fredriksen, and B. So, "An Empirical Study of the Reliability of UNIX Utilities", *Comm. ACM* Vol. 33, No. 12 (December 1990).
[5] J.A. Whittaker, "Stochastic software testing." *Annals of Software Engineering*, Vol. 4, 1997, pp 115-131.
[6] `http://www.mathematik.uni-ulm.de/sai/mayer/rt07/`
[7] W.M. McKeeman, Differential testing for software, Digital Technical Journal, 10:1, December 1998, pp 100-107.
[8] A. Groce, G.J. Holzmann, R. Joshi, Randomized differential testing as a prelude to formal verification, Proc. ICSE, 2007, pp. 621-631.

the same controlled conditions of the environment. In a multi-threaded system where we may want to investigate the sensitivity of the system to race conditions and potential data corruption, this means that generally we must control also the behavior of the process or thread scheduler as part of our controlled environment. (Subject to controlled random behavior variations like all other parts of the system under test.)

This randomized test procedure works best if the determination of whether or not a test succeeds can also be automated. A crash of the system is of course an unambiguous sign of a test failure, but it is not always that simple. One simple way to avoid the difficulty of having to formally specify an oracle that can distinguish successful executions from failing ones is to apply a technique that is known as *differential testing*. The oracle in this case is a trusted reference system that can accept and execute the same inputs as provided to the system under test. After each test, the resulting state of the oracle is compared with that of the system under test and any discrepancy can be reported as a fault in one of the two systems. Most likely, of course, the system under test will be at fault, but this is not necessarily the case. We have used this approach in the verification of flash file systems, with standard Unix file systems serving as the reference oracle. McKeeman, similarly, used a suite of trusted compilers to test a newly designed compiler. If no reference system is readily available, due to the specialized nature of the component being tested, one can be built. The reference system clearly would not be subject to the same design constraints as the actual implementation – it can be an idealized and highly abstracted or simplified version of the component. In the best case, this methodology is linked with the methodology for Fault Protection and Fault Containment that we will discuss next. In the system architecture we describe there, every critical software component is provided with a highly simplified backup version. The backup version is designed to have the same high-level behavior as the actual implementation, but provides only survival functionality. It is deliberately constructed to be at least one order of magnitude smaller in size than the primary component, which means that it can also be expected to be more rigorously verifiable. The differential testing strategy will reveal all differences between the rigorously verified component and the more complex flight article, which means that it can still serve its purpose.

5 Fault Protection and Fault Containment[9]

Non-critical software applications are often designed in a purely monolithic fashion. When the application crashes the only recourse one has is to restart it from scratch. This approach is clearly not adequate in the construction of safety- or mission-critical systems, where even a short interruption of functionality can be hazardous.

There are two primary strategies for achieving system reliability. The first is to use a design that achieves robustness through *simplicity*. A simple design is easier to understand, verify, operate, and maintain. The second strategy is to exploit *redundancy*. If the probability of failure of individual components is statistically independent, the chance of having both a prime and a backup component fail at the same time can be made very small. If all components have the same independent probability of failure p, the probability that all N components fail in an N-redundant system is p^N. Simplicity reduces the value of p, while redundancy increases the value of N. Trivially, for all values of $N \geq 1$ and $0 < p < 1$ both techniques can lower the probability of failure p^N.

Unfortunately, one of the basic premises used in the redundancy argument that we used above, the statistical independence of the failure probabilities of components, can be very hard to achieve for software. Well-known are the experiments performed in the eighties by Knight and Leveson with *N*-version programming techniques, which demonstrated that different programming teams tend to make the same types of design errors when working from a common set of (often flawed) design requirements.[10] Independently, Sha[11] also pointed out that a decision to apply *N*-version programming cannot be made independently of budget and schedule decisions. With a fixed budget, each of N independent development efforts will inevitably receive only *1/N*-th of the total project resources. If we compare the expected reliability of N development efforts, each pursued with *1/N*-th of the project resources, with one targeted effort that can consume all available resources, the tradeoffs become very different.

Redundancy in the way that has proven to work well with hardware systems, therefore, cannot be duplicated easily in software systems. A slightly different strategy could be used, though, to combine the principles of simplicity and redundancy in building software systems that are more resilient to failure.

Consider a standard software architecture consisting of software modules with well-defined interfaces. Each module performs a separate function. The modules are defined in such a way that information flow across module boundaries is minimized. We will assume here without loss of generality that the only way for modules to interact is through message passing over trusted channels. Modules execute (at least logically) on independent hardware, to secure that the crash of one module cannot affect other modules in any other way than across its module interface. A failed module may stop responding, or fail to comply with the interface protocols by sending

[9] This section of the document summarizes part of an earlier LaRS report on reliable software systems design, written in September 2006 as part of a NASA-funded study with the same title.
[10] J.C. Knight and N.G. Leveson, "An Experimental Evaluation of the Assumption of Independence in Multi-version Programming," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 1 (January 1986), pp. 96-109.
[11] L. Sha. "Using Simplicity to Control Complexity," *IEEE Software*, July-August 2001, pp. 20-28.

erroneous requests or responses. We will make a further assumption that module failures can be detected either through internal consistency checks within each module, or by peer modules that check the validity of messages that cross module boundaries.

Clearly we can only develop a fault containment strategy for component failures that are *detectable*. This means that not all conceivable types of errors can be defended against. Some errors, for instance, are only apparent after the fact. This means that consistency checks generally have to be more aggressive than may seem reasonable to the designer, but that is a different topic than we are focusing on here. We restrict our attention here to detectable defects for which a remedy is at least in principle possible.

In the proposed software architecture, each critical software component is provided with a simplified backup. During normal system operations, this backup module is idle. When a fault is detected, though, the faulty module is switched offline and replaced with the backup module. (Naturally, the backup module can have its own backup, and so on in a hierarchical chain, but we will not pursue this generalization here.) The backup, due to the fact that it is a simplified version of the primary module, may offer fewer services, or it may offer them less efficiently. The purpose of the backup, though, is to provide a survival and recovery option to a partially failed system. It is designed to provide the minimally necessary functionality that is required for the system as a whole to "stay alive."

It should be observed that in traditional software architectures a failing software component effectively serves as its own backup. Upon a failure the component is restarted (likely as part of a complete system restart) in the hope that the cause for the failure was transient. Similarly, in a dual-string system where both strings run the same software, the backup string takes over processing by running the identical software as the prime system. It will be possible to defend against a *larger* class of software defects if the backup module is *not identical* to the primary module and deliberately constructed to be *simpler* (meaning: more thoroughly testable, or even formally verifiable).

Note again that if the primary and backup modules are constructed with an *N*-version programming method we still do not necessarily gain additional reliability from this type of system structure. Some of the same design errors may be made in the construction of both modules, and if the two modules are of similar size and complexity, they likely contain a similar number of residual coding defects.

The backup modules in the approach we describe here are constructed as *simplified* versions of the primary modules. Specifically, these backup modules should be designed and built by the *same* developer that designs and builds the primary module. The primary module is build for *performance*; the backup module is built for *correctness*. The strategy can only succeed if the statistically expected number of residual defects in the backup modules is substantially lower than that of the primary modules.

If the primary component has a probability of failure p and the backup q, we should have $1 > p > q > 0$ (ignoring the boundary cases where we have either the easily avoidable certainty of failure or the unachievable case of absolute perfection). The combination of a critical component with backup has a new probability of failure (*p.q)* which, under the conditions stated, will be smaller than the individual failure probabilities *p* and *q*.

This basic mode of operation we have described here is frequently used in *hardware* designs for critical systems but so far does not appear to have been leveraged for the design of critical *software* components. A car, for instance, typically has a spare tire that is not nearly of the same quality as the regular tires – but it provides the survival capability that will allow the driver to take the car to a service station. A robotic spacecraft, similarly, normally has both high-gain and low-gain antennas. When the high-gain antenna becomes unusable, the more reliable low-gain antenna is used, be it at a reduced bit-rate. The low-gain antenna provides the survival capability that may allow full service to be restored through ground intervention. The approach also matches one of the key lessons learned reported for earlier complex systems developed by NASA, such as Skylab. (For instance, lesson learned[12] Nr 34: "*When designing redundancies into systems, consider the use of non-identical approaches for backup, alternate, and redundant items.*").

[12] http://klabs.org/richcontent/Misc_Content/AGC_And_History/Skylab/Skylab_Lessons.htm

6 The Role of Simplicity

Throughout this document, we have tacitly assumed that complexity in certain types of systems (like inter-planetary spacecraft) is unavoidable. This in itself is an important assumption. Many system designers today make this assumption, and will defend it strongly. It is perhaps important to point out here that this point of view was not shared by early designers of spacecraft, leading to NASA's first major successes such as the Mercury,[13] Gemini, and Apollo[14] missions, up to and including the first lunar landings in the late sixties. These missions clearly required what on the outset looked like a highly complex series of design decisions. Reading the documents from these early missions, though, and especially the lessons-learned sections, one is struck by the frequent emphasis on design simplicity and meticulous attention to detail, as the best method to achieve overall system reliability. As Phillips (the Apollo program director) phrased it in an Op-Ed piece in the New York Times in 1969:[15]

> "Indeed, managements often fail because they allow an organization to find complex answers by haphazard approaches rather than to seek simple solutions along well defined paths."

Complexity – though the focus of our task – should never be embraced as inevitable. The best way to assure system reliability is design simplicity and the ability to exhaustively verify critical design decisions.[16]

[13] F.J. Bailey Jr., "Review of lessons learned in the Mercury Program relative to spacecraft design and operations," AIAA Space Flight Testing Conference, Cocoa Beach, Florida, March 1963.
[14] George M. Low, "Apollo Spacecraft," 1969. (Manager of the Apollo Spacecraft Program, NASA, Houston).
[15] New York Times, 17 July 1969, p. 35, "Leaders find simple way is best," Sam C. Phillips.
[16] As Alfred North Whitehead (1861-1947) once famously phrased it: "Seek simplicity and distrust it."

Final Report

NASA Study on Flight Software Complexity

Appendix F — Managing Fault Protection Software Complexity for Deep Space Missions

Kevin Barltrop,
Jet Propulsion Laboratory,
California Institute of Technology

Contents

Summary

A recent NASA memorandum, “Issue: Growth in Flight Software (FSW) Complexity V5, NASA OCE”, was written describing the challenges faced by the growth in complexity and size of flight software. A response to that memorandum identified a special interest area for the particular challenges of fault protection software:

> *Fault protection logic accounts for a sizable portion of flight system software. Are there techniques which effectively bound the complexity of fault protection systems?*

The fault protection software special interest area calls out several activities to address this question:

1. *Encourage discussion of the issue*:

   Share JPL’s robotic mission fault protection experiences in a workshop specifically focused on fault protection, and include participation from all NASA FSW developing centers as well as representatives from industry and academia.

   Participate in NASA’s Fault Management Workshop [last April].

2. *Document the state of the practice*:

   Investigate and document approaches to fault protection used to date within NASA. What worked and what didn't?

3. *Make recommendations*:

   Make specific recommendations for techniques that bound the fault protection capabilities required for mission success.

   Investigate the feasibility of eliminating or at least minimizing fault protection software as a separate entity.

Activities for encouraging discussion of the issue have been completed through a series of on-site visits at NASA centers, through teleconferences with representatives from NASA centers and through participation and presentation at the Fault Management Workshop in New Orleans in April of 2008. This paper concludes the special interest investigation by documenting the state of the practice (Chapter 1) and by making recommendations (Chapter 2) to manage the complexity of future fault protection systems.

The examples and assessments here are based on conversations, on data collected during site visits, and on the surveys collected for the NASA Fault Management Workshop. The examples in the paper intentionally hide the names of institutions and projects in the end notes. It was written this way to promote focus on the techniques rather than on the institutions. Also, since the paper includes only information that other institutions were willing to disclose or that was already known to the author, it’s hard to avoid including more detail for examples from the author’s personal experience.

Some of the terms and phrases used here are not part of any accepted vernacular for fault protection. Since fault protection lacks a consistent lexicon, new vocabulary is introduced here to better structure the discussion.

Let us start by pointing out that not everyone even agrees as to what "too complex" means. Most would probably agree that things difficult to understand are complex. We'll refer to this complexity-of-understanding as *conceptual complexity*. Is ease of understanding the *best* criteria for "too complex?" After all, attitude estimators (such as Kalman filters) in attitude control systems are understood by few people, yet they avoid the controversy that surrounds fault protection algorithms. Similarly, science instruments routinely operate using physics and technology that only their respective experts understand.

Some things are classified as complex based not on understandability, but on *count*. For example, checking many thermal sensors is complex because it involves many individual thresholds, but the concept of checking temperature is itself a simple concept. By this definition, a brick wall could be classified as complex because there are many bricks, each with its own parameters such as its position and orientation, though conceptually a wall made of bricks is quite simple. We'll refer to this type of complexity as *enumerative complexity*.

These are two distinct kinds of complexity that occur in many areas other than fault protection, so what makes fault protection such a source of controversy? Even if the fault protection were no more complex than routine subsystem functions elsewhere in the system, it would still be a lightning rod for attention during the months before launch because, as a cross-subsystem function, it relies on flight system and test infrastructure maturity beyond that required by the individual functions of the system. That maturity arrives late in the schedule and consequently so does much of the fault protection testing. But of course, fault protection is actually quite complex, requiring the coordination of multiple subsystems, each with their own contribution to a complicated picture. So there are significant problems to be solved in both the complexity of the software and in the path to getting it ready for the mission.

1 State of the Practice — What worked and what didn't?

1.1 Background

To examine the state of the practice for fault protection software across NASA and its contractors we identify salient topics within the practice. Discussions at NASA centers and during the NASA Fault Management Workshop identified *technical assessment* and *process assessment* as useful categories for analysis of state of the practice.

For technical state of the practice we focus on *architecture*, as represented by the software framework, and on *application*, as represented in the fault handling scope.

For process state of the practice we look at the requirements approach, the verification and validation approach, and the documentation approach.

It also appears that institutions are pursuing increasingly demanding missions with new technologies that stretch and sometimes break their heritage fault protection capabilities. So we will briefly examine the progression of complexity in missions and the role of institutional priorities and technical fears in the fault protection process.

1.2 Technical State of the Practice

1.2.1 Software Frameworks

NASA centers and industry partners for deep space missions use software approaches that fall into conceptual and product line families. An institution's choice of architecture is driven by factors such as the complexity of its previous missions, its business philosophy for product reuse and improvement, and with whom it has partnered. Each software approach introduces a unique combination of problems, so not every institution is confronting the same set of difficulties today. However, organizations freshly entering technological territory already explored by others repeat mistakes of the trailblazers. At the heart of these mistakes is the temptation to accommodate more complex missions by augmenting an existing architecture rather than by reworking the architecture to fit the mission.

The evolution of fault protection approaches can be illustrated as a family tree, an illustration of which is shown in Appendix A. For purposes of this analysis, we organize the architectural approaches in terms of some key functional and organizational topics: fault monitoring, fault localization, fault recovery, and layering. These features stood out as recurring discriminators within the survey results and within discussions during the site visits to NASA centers. Inspection of the history of fault protection software frameworks and review of current practice leads to a picture of distinctive families that have evolved within the industry. These families are often shared by multiple institutions, though some institutions also make use of multiple families. Opinions regarding the benefits and costs for these families are varied and enthusiastic.

Although we'll examine specific arguments for and against approaches in each area, it's worth noting now that the biggest problem throughout industry is the management of interactions between fault protection algorithms. Historically, the interaction problem of earlier, simpler missions was more manageable due to the limited number of fault conditions that could arise and consequently the limited number of fault algorithms in play. Over time, both mission complexity and the software capability to handle more faults have increased, leading to an explosion in interaction opportunities within systems. The slope towards crossing that complexity line is slippery, with the difficulty of applying yesterday's design solution to today's mission seen clearly only from the bottom of a dark and gloomy slide.

Now on to the architecture…

1.2.1.1 Fault Monitoring Architecture

Fault monitoring refers to the function of detecting fault symptoms or errors. A fault monitor acts as a type of estimator, filtering down data into a few qualitative bins (such as good/bad/unknown), using thresholds or other checks. To filter out noise and ignorable transients, fault monitoring often applies smoothing and/or persistence counting. The surveyed fault monitoring frameworks can be characterized according to several recurring features: deployment, statistic generation, false alarm control, and behavior specification.

1.2.1.1.1 Deployment

This refers to the choice of location for the monitoring software functions. It can be local, central, or more frequently a combination of local and central. In local deployments, fault monitoring is distributed throughout the flight software with monitoring functions for a subsystem resident with that subsystem's software. When subsystem teams develop their own fault detection algorithms, local deployment usually results. Making a monitor resident with the subsystem algorithms permits convenient interfacing to local response algorithms that manage subsystem resources, and in some architectures it ensures the use of consistent criteria for both triggering local algorithms and for notifying corresponding system level algorithms.[i] Unfortunately, many local deployments of fault monitoring rely on individually hand-coded functions without a standard framework. Lack of standard interfaces and services, such as statistics generation and state reporting, make testing and operations more difficult. However, some institutions do use well-defined local monitoring frameworks that define base software classes with standard services and conventions.[ii]

Central deployments of fault monitoring inspect data that pass through central telemetry conduits, for example, the telemetry manager task.[iii] Central deployments provide a standard framework with consistent operations and test interfaces and services. However, central deployments must often co-exist with legacy subsystem software containing locally deployed fault monitoring (frequently attitude control software). Coordinating local and central deployments usually falls to sending "good/bad" tokens rather than raw data from the local to the central software. In cases where there is no local subsystem framework for monitoring services, and where only tokens are available to the centralized fault monitoring, visibility into the state of fault monitoring is greatly reduced.

1.2.1.1.2 Statistics

Visibility into the state of fault monitoring is extremely important for efficient troubleshooting. A major form of visibility comes through the generation of running telemetry statistics. Some systems generate no statistics at all, probably making false alarm margin assessment and debugging very difficult.[iv] A chief benefit of on-board data statistics generation, as opposed to ground based statistics, is that it can cover every sample point for a data channel without the cost of recording and then transmitting each raw sample point as telemetry to the ground.

A common statistic, know sometimes as a "watermark," is employed by several frameworks[v] to capture the extreme values of telemetry for a given period of time. Knowledge of these extreme values allows a project team to understand their margins between observed flight values and fault detection thresholds. In some frameworks the watermarking is applied to the values exactly as they are checked against detection thresholds. Therefore the watermark data gives a direct reading of how close values are to exceeding their thresholds or by how far they have already exceeded them. In other implementations the watermarking may or may not be applied to the specific data that is checked against thresholds, sometimes because the watermarked data is subject to additional processing steps before it is compared against thresholds. For those cases additional ground reconstruction is needed to really understand the margin story and in some situations accurate reconstruction is not even possible. These differences in approach show that not all statistics generation solutions are of equivalent value and that project teams should be very precise about what data is actually fed into the statistics telemetry.

At least one framework family includes others statistics such as the mean, mode, and standard deviation.[vi] Since most statistics include system performance parameters covered by mission requirements (usually as n-sigma distribution requirements), confirming requirements compliance is made easier by the availability of the additional statistics.

1.2.1.1.3 False Alarm Control

The next architectural feature of interest for fault monitoring is the means of false alarm control. This comes primarily through the mechanisms for controlling when a monitor is active and how it handles expected transients. Monitors must operate in systems that pass through configuration and software mode changes. These changes often introduce anomalous data that violates threshold or state checks, leading to opportunities for false alarms. All systems employ some combination of mechanisms to avoid these false alarms during transitions. Approaches for this fall into two key strategies: manual control and automatic control.

For *manual control*, an external function (for example a fault response or a ground command) must explicitly change the monitoring behavior through a command or parameter change at the appropriate time.[vii] For example, a spacecraft may depend on command sequencing to disable a component's fault monitor for several minutes following that component's power-on.

For *automatic control*, the monitor is supplied with sufficient state data or filtering to automatically suppress false alarms. For example, monitors that inhibit health checks of hardware when the hardware has been commanded OFF and then continue the inhibition until after some defined warm-up period, will automatically avoid typical start-up false alarms.[viii] A simpler automatic approach is to ride through transient conditions (expected or otherwise) using a generous persistence threshold.[ix]

Proponents of the automatic control approach claim that some kind of monitor behavior control is needed to accommodate known transient conditions, and that implementing it in a standard way within the infrastructure is the most robust approach to avoid false alarms. Furthermore, employing mechanisms that explicitly recognize the underlying physics of monitoring (e.g., warm-up times, ignore data for hardware commanded off) avoids the parameter tuning games that over-optimize for either false-alarm suppression or fault detection. Critics of the automatic approach point out that automatic behavior control adds to the expense of the flight software, and that the cost of controlling the monitor behavior can be moved to the development of the test and operations products.

1.2.1.1.4 Behavior Specification

Our last architectural feature of interest for fault monitoring is the means of behavior specification. This refers to the means by which the basic fault monitoring behavior is described. Two families are represented in the surveyed software. *The table-based approach* relies on a collection of parameters applied within a set of standard threshold-checking algorithms.[x] For example, each entry in the table may specify the data to be checked, the thresholds to check against, and the persistence filtering limits.

Sometimes, the data available for monitoring is not useful in its raw form and additional preprocessing must be applied.[xi] Table-based approaches sometimes add a hook to first call such a preprocessing function. Because the underlying framework for the table-based approach defines only fixed reused logic for checking data, these extra preprocessing functions can and do become a “black box” catch-all home for additional logic not accommodated by the framework. Consequently, what may appear in a summary chart to be a simple threshold check actually includes hidden logic that obscures important details of the monitoring. In at least one institution, software data structure and enumerative range limits in the framework drove the team to use the preprocessing functions as a packing/unpacking mechanism to perform multiple checks at once, making it even more difficult to predict the monitor behavior.[xii]

Proponents of the table-based approach point out that adding a simple threshold check entry to a table is easier than installing new or patching existing software algorithms. There are some examples where an operations team was able to make such a change very quickly in response to a pressing need.[xiii] Critics of this approach assert that allowing such changes to be handled as a simple operational procedure rather than as a software change increases the probability of making an operational error. They also claim that confusion results from using tables without intuitive metaphors for the checks performed.

The other major behavior specification family, *case-by-case logic description*, relies on diagrams and/or pseudo-code to describe the behavior. For example, flow charts of one form or another specify the logic flow, preprocessing, and checking of data, thereby providing the flexibility to design whatever filtering or binning strategy best maps to the underlying data-checking objectives.[xiv] When implemented on top of a framework that provides services for recurring functions, it provides standardization as well.

Proponents of the logic description approach claim that describing each monitor explicitly using a flexible specification allows the software to better reflect the underlying fault state estimation problem. This avoids getting boxed into stock threshold-based detection solutions that don’t fit the problem. Critics of the logic description approach claim that explicit specification is expensive and that because the threshold detection approach can work for most cases, it is cost effective to use it.

1.2.1.1.5 Other Features

The previously described characteristics (deployment, statistics, behavior control, and behavior specification) occur in most of the examined fault monitoring approaches. Some solutions, however, exhibit additional special features, usually as a means to tie internal aspects of fault monitoring to intuitive concepts. These concepts include "warm-up time" allocations, "opinions" on data quality, "symptom" identifiers, enabling/disabling, etc…[xv]

1.2.1.2 Fault Localization Architecture

Fault localization (also called fault isolation) refers to the function of determining where a fault has occurred. For basic *hardware* faults, the location can be inferred directly from the fault symptoms. However, accurate identification of *system* faults may require combining symptoms from multiple areas. The fault localization architectures can be characterized according to deployment and mapping technique.

1.2.1.2.1 Deployment

Like fault monitoring, the deployment of fault localization can be classified as either a local or central approach. The local approach requires gathering together of relevant symptom indications within some local subsystem software. The central localization approach collects together indications from all of the subsystems into one software task to generate a conclusion about the location of the underlying fault. The results of the fault localization are used to select the fault response.

1.2.1.2.2 Mapping

Fault localization architectures can also be divided into the table versus logic specification families. For the simpler *table-driven approaches* a many-to-one mapping table accommodates the case of multiple symptoms indicating the same fault location.[xvi] However, this table driven approach does not accommodate cases where a symptom may indicate one underlying fault when accompanied by symptom "X," but an entirely different fault when accompanied by symptom "Y."

The *logic specification approach* provides the flexibility to use logical combinations of fault indications, and furthermore allows them to be expressed in intuitive terms.[xvii] The same ease of updates versus risk of operator error arguments given for the fault monitoring framework apply to the fault localization framework.

1.2.1.3 Fault Response Architecture

Fault response refers to the function of altering the configuration and behavior of the spacecraft in response to a detected fault condition. Fault response architectures can be characterized according to their approaches for deployment, thread control, interaction management, behavior selection, command execution, and responsiveness to system state.

1.2.1.3.1 Deployment

Like fault monitoring and localization, fault response deployment can be local, central, or more commonly, a combined approach. All architectures probably include at least one class of *local response* within subsystem software, though it is not always explicitly called out as such. That is, any time software executes different algorithm paths in response to observed data, the system is performing a form of local response. For example, screening out anomalous sensor readings as part of a control loop would be a local response. Some architectures implement local responses specifically as a means to "ride through" anomalies and inhibit escalation of minor local problem into a bigger system problem.[xviii] Other architectures allow subsystem software to actually perform local configuration changes or repair actions for hardware, according to its layering architecture (See next section on Layering Architecture).[xix] All architectures surveyed had at least some level of *central response* deployment, usually referred to as a system fault response and often involving a variant of "safing." These responses have the capability to make configuration changes across multiple subsystems. The research uncovered no particular fundamental problems associated with this aspect of deployment.

Deployment can also be characterized as dedicated or shared. The dedicated deployment uses a framework reserved explicitly for the handling of faults. A dedicated deployment would typically have a component described as a fault manager or fault engine.[xx] Shared deployments use the same mechanisms for fault protection that are used controlling general mission activity behavior. A shared deployment would typically use activity sequences with logic built-in to handle fault situations.[xxi] Another architecture uses a state control and goal-based software algorithm approach that drives mission activities and fault handling behaviors.[xxii] One mission took a mixed approach combining some software logic with calls to the same sequence engine used by mission activities.[xxiii]

1.2.1.3.2 Threads

The complexity of behavior for a given response architecture is heavily influenced by its approach to thread execution. [This does not necessarily mean *task* threads as provided by the operating system, as some systems include engines that define their own internal threads (e.g., sequence engines)] The most common approach for thread execution is in-parallel execution of multiple responses. For distributed deployments involving multiple layers, this can result in the tandem execution of multiple local *and* central responses. Most central deployments surveyed do provide a framework to also execute multiple system level responses in parallel.

No surveyed architecture was limited to purely serial execution, but at least one framework family limits in-parallel execution to a form of interleaving, where a response yields execution to another only at designated interruption points where it is safe to do so.[xxiv] One mission flying that framework chose to avoid that capability in the design, effectively giving a serial-only implementation to avoid the possibility of response collisions.[xxv]

The benefit of the in-parallel approach is that it allows a system to clock out configuration changes very quickly. For example, if three reaction wheels need to be powered on and initialized, an in-parallel design can spawn the three responses (one for each wheel) simultaneously, whereas a serial approach would require calling the responses one after another. The Achilles heel of parallel execution is that without a robust interaction management approach a system encounters unpredictable and occasionally fatal interactions and collisions between responses. Engineers must spend considerable effort finding these interactions and then tweaking timing and enable/disable settings to avoid them. At best a project can hope to wring out the most likely interactions, but without explicit interaction mechanisms in place, some problem scenarios remain.

In theory, serial execution of responses has the benefit of simplifying behavior and making it more predictable. It greatly reduces the problem of interactions between responses and eliminates the problem of collisions between responses. The weakness of this approach is its inability to quickly complete multiple actions at once and does not allow a system to interrupt a lower priority response with a higher one. In some missions, the system design may allow responses to be so short that the penalty of holding off a higher priority one is acceptable; but that will not be possible for all cases.[xxvi] As noted earlier, no system in use appears to limit behavior to strictly serial execution, but there are systems that only allow parallel execution under certain conditions.

1.2.1.3.3 Interactions

Interaction management is strongly tied to the serial/parallel situation. Test runs for parallel execution systems with no formal interaction management mechanisms show chaotic, hard-to-predict behavior. This situation is evident in scenarios where parallel threads try to simultaneously drive system states to different values (e.g., one trying to turn a component off, while another trying to turn it on). In such cases at least one response fails to achieve its intermediate objectives, and likely fails to achieve its final objectives. Worse, the end states resulting from multiple threads at work can actually be incompatible, leading, for example, to a situation where the system is configured to perform a function using hardware that some other response has powered off.

Approaches to interaction management fall into four main camps: case-by-case enable/disable manipulation, timing tuning, resource contention management, and control allocation. *Case-by-case enable/disable manipulation* is an implementation strategy rather than an architectural approach. Practitioners working with a system lacking built-in safeguards realize that interactions occur as they add more responses, and find that particular cases can be resolved by disabling competing algorithms. The choice of what to enable and disable is often driven by the need to de-conflict the specific set of cases that have been discovered and lacks the structure of an overarching strategy.[xxvii]

Timing tuning depends upon finding particular timing situations that result in bad interactions. The timelines for these scenarios can sometimes be manipulated enough through command spacing to deconflict the interaction. Unfortunately, this often solves the problem for a very particular set of initial conditions with no guarantee that the fix will work for other initial conditions.[xxviii] It can required several iterations sometimes to adjust the timing to solve for a range of identified conditions.

Resource contention management involves at least two key features: an identification of resources that competing (and possibly prioritized) software functions may want to control or change; and a fall-back strategy for functions whose resource needs can not be met. If one response wants a component powered off and another wants the component on, then the component is given to the higher priority function, and the lower priority function must go to a fall-back action. In some of the architectures, resource contention management is built into the framework.[xxix] At least one project augmented its framework with an ad-hoc implementation of resource management.[xxx]

Control allocation is the practice of designating one function at a time as capable of controlling a system state. The simplest approach is to have a static allocation where manipulation of a state must always be performed via that controlling function.[xxxi] State-based system design capitalizes on this approach, for example, by analyzing the design in terms of the controllable states, and then by dedicating software functions to control those states.[xxxii]

A system's layering architecture often organizes control of states at different levels. For example, the component layer might control the physical power state of a hardware item, while a system layer controls the desired power state.[xxxiii]

In some architectures, particularly those with critical event protection, some form of inter-lock logic allows transfer of control between multiple software functions according to a set of hand-off rules. For example, frameworks with a fault management function and a critical sequence function allow fault management to pause critical sequences, thereby asserting control of the spacecraft states, and then later relinquishing control back to the critical sequence. Timeouts and other mechanisms are often in place to avoid deadlocks and to constrain the hand-offs to certain safe points. [xxxiv] Other approaches use sophisticated sequence (or Macro) engines with access to system data to implement both critical activity execution and fault response activity with one mechanism.[xxxv]

In one architecture, interactions between in-parallel responses were managed by having each response explicitly check the states of other responses.[xxxvi] This approach made it possible to predict behavior on a case-by-case basis, but did not give an intuitive sense of the overall behavior.

Framework solutions for interaction management come from a deliberate effort to deal with the underlying types of conflicts that occur. Although they make the framework itself larger and more complex, they reduce the resulting behavioral complexity by forcing the system to obey cross-cutting rules and conventions for the control of states and resources. Once a reusable framework has been vetted, later implementations using the framework have easier testing and troubleshooting. Interaction management is certainly the most broken area of the fault protection software practice, and institutions would be wise to look very closely at this issue for future work.

1.2.1.3.4 Behavior Selection

Behavior selection refers to the function of selecting which actions to perform as a result of a fault-detection. Strategies in use include simple table look-up, in-line response software logic, and sequence/macro logic. Architectures often use a mix of these strategies. The simple *table look-up approach* uses an incoming ID corresponding to some fault detection that is mapped to the ID of a result response algorithm.[xxxvii] *In-line software logic approaches* include state machines and flow chart type designs.[xxxviii] *Sequence/macro logic* is used for systems that have command sequencing capability that supports conditional execution.[xxxix] A typical combination strategy is to use a simple table-look up to select a response algorithm responsible for handling certain classes of faults, but then to have in-line logic within the response (software or sequence/macro) to refine the choice of actions further.[xl]

Critics of the table look-up only approach point out that it leaves little room for interaction management. However, it's possible that additional table data to parameterize resource use by responses could supply a table-based interaction management solution. Since all deep space applications appear to augment the table approach with either manual or automatic interaction management, it's safe to say that table look-up alone does not provide a good solution.

1.2.1.3.5 Pacing

Behavior pacing refers to the function of allowing appropriate time for the effects of responses to be realized before moving on to additional (usually more severe) actions. It's closely related to the problem of interaction management, except the effects of interest are the side-effect or redundant fault detections that come about through the physics of the fault rather than through command collisions or incompatible commanded states from in-parallel responses. Systems with poor pacing race through multiple configurations in response to side effects without giving the applicable fix sufficient time to take effect. Pacing strategies fall into two key families: *manipulation of fault detection* or *direct control of response progression*.

Behavior pacing through *manipulation of fault detection* usually relies on either temporarily disabling fault detection within the responses, by tuning persistence values in such a way as to stagger the timing of triggering new responses, or by creating local responses to reduce escalation of anomaly effects. Using responses to *temporarily disable* and re-enable fault detections depends greatly on avoiding interaction problems that might leave fault monitoring in an inappropriate end state.[xli] At least one mission made several algorithm timing changes after testing to avoid just that situation.[xlii] Any system that lacks good interaction management must be very cautious about manually piloting the enable/disable state of monitors.

Persistence tuning imposes a responsibility on the persistence limits to support both accurate estimation of fault states and timing of triggering responses.[xliii] Unfortunately, combining these two concerns can lead to persistent limit values over-optimized for one of those responsibilities, or well-selected for neither.

Local responses that inhibit escalation of fault effects generally provide good results so long as they are tuned to reflect the tolerance of the spacecraft subsystem algorithms to outages rather than to explicitly manipulate the timing of the responses.[xliv] Otherwise, the combining of concerns results in an optimization problem similar to that of persistence tuning.

Behavior pacing through *direct control of response progression* approaches the issue by working through the response design rather than through fault monitoring. The most common approach to controlling pacing is to include delays between commands in a response. This provides some very basic timing control, but must be augmented with other approaches when the wait times are too long to satisfy recovery time constraints. Another approach uses state machine designs with explicit checks within the response logic to wait for a desired state to be achieved and/or for some period of time to pass.[xlv] A third approach uses a table that specifies, for each configuration change, how long the system must wait before allowing a new response effecting that same configuration area to execute.[xlvi]

Pacing control improves the predictability of the system by making it less likely that the system will "over respond." For simple systems that have few configuration options, such as monolithic whole side swap approaches, pacing is less of a concern. But any system with significant autonomous cross-strapping must face the pacing problem.

1.2.1.3.6 Command Execution

Command execution methods fall into two categories, and as typical, often combine two methods. The most basic *sequence-based* command execution approach uses predefined command sequences with fixed delay times between commands.[xlvii] More sophisticated sequence capabilities allow conditional logic within the sequences, blurring the distinction between sequencing and flight code.[xlviii] The alternative, *in-line command* execution, is used in architectures where response algorithms are implemented with flight software code rather than just sequences. Rather than invoking canned command sequences, the algorithms make individual function calls to execute spacecraft commands.[xlix]

The benefit of using command sequences is that the fault protection software can leverage off of an existing infrastructure with a simple interface to clock out commands. This benefit is not realized, of course, if the fault protection software uses its own dedicated sequencing framework instead of reusing the flight operations sequencing framework.[l] Making use of command sequences also has the benefit of ensuring that possible combinations of commands for a response are all defined before-hand and can be tested as stand-alone products without the overhead of the response algorithms. At least one project did rapid shot-gun testing of sequence combinations separate from the response algorithms to prove that the sequences were compatible with the end states created by other sequences.[li]

A benefit of in-line commands is that the system can specify command arguments on-the-fly. The specification for the command can then be done once using run-time settable arguments rather than several times at pre-run-time as needed for predefined sequences. The in-line command interface approach does, however, require a more complex framework to handle the variation in command parameters that may exist. One survey respondent used a combination approach, dividing up the commands into small sequences that each configure a very basic item, but then calls each of these sequences with in-line code. In theory, a sequence-based interface in which each sequence is limited to a single command would be nearly equivalent to an in-line approach. Reducing the number of sequences by combining single-command sequences would then reduce the possible command combinations at the expense of some flexibility. It is inevitable that in-parallel sequenced-based command execution without framework provisions for interaction and pacing management will produce unpredictable and difficult to test behavior.

1.2.1.3.7 Responsiveness

System state responsiveness refers to the capability of a fault response to react to the current state of the spacecraft. Systems that respond to a condition that is no longer present tend to give inappropriate behavior. As a simple example, it would be inappropriate for a fault protection system to power cycle a hardware component that the ground has commanded OFF between the time of the fault detection and the time of the fault response execution. It would be better to just leave the hardware item power OFF as commanded by the ground. Solutions to this class of problem usually involve additional checks in the fault responses to make sure that triggering fault states are still present and that current software modes and states of the system are compatible with the response’s intent.

Proponents of these additional checks claim that although this requires more software logic, it does give much simpler behavior for the system. Critics point out that it requires an increase in software that must be debugged and validated.

1.2.1.4 Layering Architectures

The layering architecture refers to the organization and flow of control of fault handling between different software algorithms. Layer architecture can be described in terms of layer categories, and in terms of the explicitness of the organization and flow control.

One of the surveyed layered architectures defines three layers, with categories such as "component level," "performance level," and "system level."[lii] Another architecture defines categories such as: "local," "system/hardware," "system/performance."[liii] Yet another defines only a single level.[liv] The layering approach defines properties for each category and for relationships between categories. There is often a hierarchical relationship between layers where a lower level layer will attempt to resolve a problem, relinquishing control to a higher level layer if the problem cannot be so resolved. Another approach is the unlayered model, which dispenses with hierarchy, instead focusing on point-to-point relationships between algorithms.[lv]

Proponents of the layered approach claim that hierarchies map well onto a way of thinking already familiar to people. It allows one to review and analyze behavior at well-demarcated levels of detail (i.e., discussing all component level behavior versus system level). Critics say that the layered approach results in the force-fitting of behavior to artificial categories that drive systems through problematic configurations paths (for example, unnecessary swapping of hardware). Critics of the unlayered approach point out that allowing the system to go beyond a few well-defined paths produces a range of behavior that can't be verified and validated using traditional methods.

1.2.2 Survey Data on Software Architectures

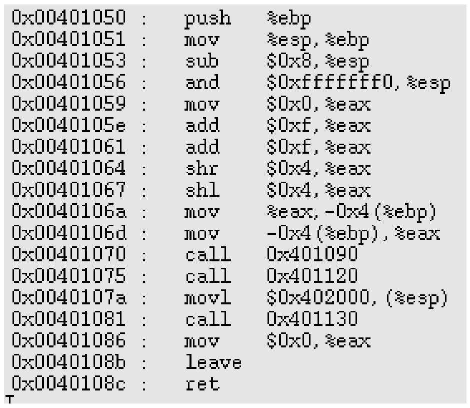
Appendices B through D tabulate the families of fault monitoring, detection, response, and layering, respectively, for deep space missions. The discussion above informs on the pros and cons for each of these applications.

1.2.3 Consistency of Terms and Concepts

A recurring observation during the course of the workshop, site visits, and teleconferences was the lack of consistency in fault protection terminology and its use in the software frameworks. Most institutions recognized fault protection concepts such as detection, isolation, and response, but used different terms to describe them, or in many cases, used generic labels that did not speak to the specific concepts of fault protection (for example, "macro"). For example, a given software architecture may support functions compatible with the notion of fault recovery, but the actual source code developed by engineers generically monitors for telemetry conditions (good or bad) and runs real-time sequences without ever including software identifiers that reflect the idea of a fault or system response.[lvi] Explicitly including such identifiers and constructs (i.e., *abstractions*) applicable to fault protection improves the understandability of the software.

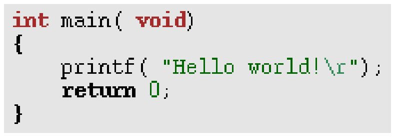
The surveyed architectures vary in the number of formal abstractions provided by the software frameworks. Most architectures support concepts such as response, monitor, tier, threshold, persistence, safe mode, enable/disable, sequence, min/max/watermark, hierarchy (system/subsystem/component). Other architectures provide extensions such as watch mode, recovery time, warm-up/settling, interruption priority, resource competition, mission mode, activity resumption, opinion, fault, and symptom. Although the developers and immediate users consider abstractions useful for communication among themselves and to tie the software implementation directly to intuitive behaviors and concepts, by-standers and reviewers often require training to understand what lies behind an abstraction.

Although this may seem like a minor point, reliable review and understanding of software depends heavily on using clear and intuitive references. For example, consider the example assembly code snippet below:



```
0x00401050 :   push   %ebp
0x00401051 :   mov    %esp,%ebp
0x00401053 :   sub    $0x8,%esp
0x00401056 :   and    $0xfffffff0,%esp
0x00401059 :   mov    $0x0,%eax
0x0040105e :   add    $0xf,%eax
0x00401061 :   add    $0xf,%eax
0x00401064 :   shr    $0x4,%eax
0x00401067 :   shl    $0x4,%eax
0x0040106a :   mov    %eax,-0x4(%ebp)
0x0040106d :   mov    -0x4(%ebp),%eax
0x00401070 :   call   0x401090
0x00401075 :   call   0x401120
0x0040107a :   movl   $0x402000,(%esp)
0x00401081 :   call   0x401130
0x00401086 :   mov    $0x0,%eax
0x0040108b :   leave
0x0040108c :   ret
```

The intent of this code is probably unclear to most readers. This equivalent “C” version, however,



```
int main( void)
{
    printf( "Hello world!\r");
    return 0;
}
```

is understandable to many people. A key difference between the two versions of the code is that the “C” version has abstractions that map to concepts (like print) that are familiar to people. The same incentive for comprehensibility exists to express the autonomy design of systems in familiar, everyday concepts. A difficulty in using abstractions, however, is that understanding what lies behind an abstraction requires training and experience. Even the “hello world” example can leave some specific questions unanswered, such as “What happens when the \r character is printed?” [Answer: It depends. On a Windows machine it will result in continued output at the start of the current line. On a UNIX machine it will result in continued output at the start of the next line].

It’s not enough just to decide to use abstractions in the design and development. A project must also decide where to use them. For example, while the flight software installed on a spacecraft is binary machine code with no useful fault protection abstractions, the original source code usually does contain labels and identifiers associated with fault protection. Furthermore, one can choose to locate the abstractions somewhere other than in the source code. For example, behavior diagramming tools can be used to automatically generate source code, which in turn produces the binary image. The necessary location for abstractions is within those design artifacts directly created and reviewed by humans, be it "C" code, a database, or a design diagram.

So, abstractions allow us to quickly communicate a high level view of a complex concept and codify enforced properties that are applicable to the abstraction, but they also leave us open to ambiguity about details until we look under the hood. This is the way people have always communicate complex ideas. It's not perfect, but without these abstractions we spend so much time on details that we "lose the forest for the trees." The key to promoting clear understanding is to pursue consistency of those abstractions and work hard to map them to the concepts that people already understand.

1.2.4 Scope of Fault Handling

Our survey shows that the scope of fault handling varies largely, depending upon the inclusion of critical events, the degree to which systems are expected to fend for themselves without ground intervention, and the project and institutional risk philosophies. Coverage can usually be characterized according to fail safe versus fail operational, single fault tolerant versus selected redundancy, organization of fault containment, and failure possibility versus probability. Fault recovery is also often described informally in terms of "temperament." For example, the expression "Big hammer approach" was recognizable to many as a strategy for swapping out all hardware at once even for a very local fault.

1.2.4.1 Critical Events

A number of missions include what are commonly referred to as critical events. Critical events are those events whose success is absolutely necessary to the success of the mission. For example, the success of a lander mission depends entirely (but not exclusively) on completing a successful entry, descent, and landing (EDL). Critical events come in at least two flavors. The *time-critical event* consists of an activity that must be successfully completed during a small window of time with no opportunity for the ground to conduct a second attempt. For example, EDL cannot be performed after the spacecraft has flown by the landing target. The *irreversible-critical event* is one in which there may not be a small time window constraint, but for which catastrophic failure of the attempt still results in premature end of mission. For example, repressurizing a propulsion system may be done during one of many windows, but catastrophic failure during the attempt would end the mission.

The handling of faults during critical events falls into two major approaches: Autonomously recover and resume the activity; or ignore faults that would require autonomous recovery and resumption. Implementing *autonomous recovery and resumptio*n brings with it the interaction problem, in some cases of the kind that pits the fault protection response algorithms against the activity execution algorithm. Any mission that chooses the autonomous resumption path must be prepared to take on the interaction problem. *Ignoring faults* is a self evident tactic premised on the assumption that the short durations of critical activities make it unlikely that a previously healthy spacecraft would suddenly fail.

Critics of the autonomous recovery approach point out that the interaction problem is too expensive to justify for such a small window of time during the mission. In fact, debugging interactions is likely the biggest cost upper on projects with autonomous recovery of critical events. Proponents of the autonomous recovery approach say that much of the expense comes from the choice of design solutions and not from the concept of autonomous recovery itself. Critics of ignoring faults say that the “short duration” argument does not address the fact that critical activities often include significant configuration changes and shocks within the system, providing credible opportunity for previously well behaved hardware to suddenly develop a fatal tic or two.

1.2.4.2 Limited Ground Contact

Deep space missions as a rule have limited ground contact opportunities compared to earth orbiters. Institutions taking on deep space missions for the first time quickly encounter the cultural differences between earth orbit and deep space missions. Nearly constant monitoring and negligible light-times for communications allows earth orbiting missions to allocate more spacecraft protection responsibility to the ground. Earth orbit scenarios assume that fault handling timelines are long enough to permit the ground to become part of the chain of response. Deep space missions, however, have stretched timelines, with days or even weeks between ground contacts that cannot accommodate the ground as part of the chain of response.

A frequent implementation that reflects this issue is the choice to autonomously disable fault handling once a spacecraft has entered its so called “safe mode.”[lvii] In earth orbiters, the mission can expect quick intervention (within a day if not hours) by the operations team to recover from the anomaly and restore fault protection. In deep space missions, a flight system may have to sit in this unprotected configuration for days if not weeks. Institutions new to deep-space exploration must carefully review the assumptions underlying the tried and true safe and disable approach.

1.2.4.3 Risk Posture

The risk posture of a project team’s *home institutions* is perhaps the most subjective of factors influencing the scope of fault handling. As well as the question of high versus low risk, it’s often a matter of opinion about what technical solutions are risky or safe. Part of this comes from the fact that practitioners themselves vary in their experience and resources, so what one person would consider risky, another considers business as usual. The implication of this is that the technical approach and expertise of the organization is an extremely important component of the risk level inherent in a mission. If a company has no experience with critical events recovery, for example, then it may be more risky for them to attempt it than it is to hope that no fault will occur.

Program risk posture is that handed down through the project contract in response to the cost and risk assumptions of the sponsoring program. For example, the Discovery Program missions are intended to be lower cost, and therefore more accepting of risk compare to, say, a flagship outer planets mission.

Unfortunately, the risk posture tends to change as a project moves from formulation phase through implementation and operation. A proposal team may be willing to sign up for higher risk in order to reduce cost, but the development and operations teams often get cold feet, pursuing more expensive designs and procedures that reduce risk. Prevailing political environments can also influence the project posture. For example, following the Mars Observer loss, the once acceptable *faster, better, cheaper* philosophy gave way to a more conservative posture that was retroactively applied to in-progress missions.[lviii]

1.2.4.4 Fault Handling Strategy

Fault handling strategy refers to the depth and breadth of protection for functions, resources, and activities of a spacecraft. The fault handling strategy approach is strongly tied to the project risk posture and to the presence of any critical mission activities. Classification of fault handling strategy can be expressed in terms of fail safe versus fail operational, single fault tolerance versus selected redundancy, organization of fault containment, and failure possibility versus probability.

1.2.4.4.1 Fail Safe versus Operational

A project's policy for when to fail safe versus when to fail operational is one of the most important it can establish. The *fail safe* strategy dictates that the spacecraft preserve the health and safety of mission critical assets, but not the performance or completion of mission activities. When a fault occurs, fault protection cancels any pending or in-progress mission activities, completes whatever repair actions it has available, and then leaves the spacecraft in safe mode. This is workable for projects with no mission critical activities and a fact of life for projects that lack the funding and schedule to pursue the fail operational strategy. The phrase "health and safety" is usually interpreted to mean the immediate critical functionality of engineering systems as well as the long term health and safety of other mission critical resources such as science instruments.

In the *fail operational* strategy, the spacecraft protects the completion of the ground specified activities in addition to core health and safety. It requires a more sophisticated framework to carry off successfully and is difficult to thoroughly test.[lix] Protection of mission critical events, such as launch initializations, orbit insertions, and landings, is the most common application of this approach. Historically, fail operational fault responses have rarely been invoked during the actual critical activity, so it has usually been an insurance policy [When has it been used?] rather than a demonstrated savior of missions.

Another variant is the *best effort fail operational* approach. The best effort approach restores failed functions without interrupting mission activities, falling back to fail safe only if some other system performance fault occurs.[lx] It is most commonly used during non-critical periods for systems already possessing the fail operational capability. These systems leverage off of the fail operational infrastructure to reduce the variety of fault protection modes and to reduce the impact of faults during non-critical event periods of the mission.

1.2.4.4.2 Single Fault Tolerance versus Selected Redundancy

Single fault tolerance refers to the practice of supplying sufficient redundancy (block or functional) so that a single fault anywhere in the system will not result in the loss of the mission. *Selected (or selective) redundancy* refers to the practice of supplying redundancy (block or functional) to increase system robustness in selected areas on a case-by-case basis. *Block redundancy* refers to the use of identical redundant strings of hardware as backup, while *functional redundancy* refers to the use of alternate tactics to compensate for the loss of functions or performance.

The choice of method to emphasize derives largely from the risk posture and cost of the project. A rule of thumb at one institution is that long duration (> 10 years) or flagship missions[lxi] should adopt a single fault tolerance policy while others should adopt the selected redundancy policy. According to the reliability group of one organization, the rationale for the single fault tolerance policy is less about parts reliability and more about human error. It comes from a belief that human error will trump hardware reliability estimates and introduce failures regardless of what a parts-based reliability estimate may show. It also assumes that there is a high probability that the human error would not be repeated in the primary and backup hardware. Since it is not possible to predict *where* the human errors will occur, the theory is that supplying redundancy for all critical components will cover all bases, and for any special cases where redundancy cannot be applied, the mission plays the odds that the human error will not occur there.

Deciding *where to add* redundancy for a single fault tolerant system is considerably easier than that of systems with selected redundancy. A single fault tolerant system has redundancy for all critical components, except for a few exempted areas of non-credible failures (such as bus structure).[lxii] A selected redundancy approach requires somewhat subjective decision-making for where to spend resources (mass, dollars) for redundancy.[lxiii] Engineers usually choose based on an approximate cost-benefit assessment with data provided by an analysis such as a Probabilistic Reliability Analysis (PRA).

The *application* of single fault tolerance is considerably more expensive than selected redundancy. Aside from the cost (dollar, mass, etc…) of the additional hardware, there is great complexity in wiring and writing software to autonomously use cross-strapped hardware. The Fault Containment Organization section of this paper will continue that discussion.

1.2.4.4.2.1 Single Fault Tolerance

Unfortunately, lack of consistent interpretation of the phrase “single fault tolerance” has led to confusion for the scope of fault protection on at least several occasions. The problem starts with the word “fault.” Most institutions use “fault” to refer to an event during which a system fails to meet its functional or performance requirements. Consequently, “single fault tolerance” would mean that the system must tolerate (where tolerate means that it meets minimum mission requirements) one such event, but not necessarily more. Furthermore, events such as a processor reboot from a high energy particle hit, a command error, or a software exception would be considered to be a fault.

At least one NASA center,[lxiv] however, defines “fault” as the “underlying cause of a [hardware] failure or anomaly.” The center uses “error” or “symptom” as terms most equivalent to the other usage of “fault.” For the “underlying cause” definition of fault, “single fault tolerance” would mean that the system must tolerate one underlying hardware failure, though not necessarily more.

There is significant difference in the intent of single fault tolerance for these two definitions. The event based interpretation has led to systems that respond to false alarms, environmental disturbances, and command errors by executing a fault response and then disabling most or nearly all further fault protection. For the institutions that take this approach, they have, technically speaking, satisfied the single fault tolerance requirement per the event-based definition. A typical design of this type has a fall-back emergency configuration that supports basic operating functions with no further capability to use redundant supporting hardware, or to reset single-string hardware.[lxv]

The organization that uses fault to mean underlying failure cause, however, considers any event not tied to an underlying failure to be an error, not a fault, and therefore not one of the single faults mentioned in the policy. Per the more common industry usage, this approach would probably more accurately be described as "single failure tolerance." Recent consequences of this disconnect over the meaning of "single fault tolerance" have led to several cases where the contractor proposed a design that they believed to be "single fault tolerant" but was later found not to meet the intent of the requirement.[lxvi]

1.2.4.4.2.2 Selected Redundancy

The chief difficulty of the selected redundancy approach is deciding where to apply redundancy. Reliability models of a system can identify particularly vulnerable areas of the system, but since selected redundancy is used more often on less expensive missions, the cost of adding redundancy is not always compatible with the project cost cap.

The accuracy of the reliability estimates is also suspect. A great many assumptions and models contribute to the outcome of those estimates. Modeling by analogy, for example, can lead to inaccurate results when there is a lack of understanding of the differences between an assembly being analyzed and the analogous assembly used as a reference. So, decisions based on this kind of analysis should focus on no less than orders-of-magnitude differences.

1.2.4.4.3 Fault Containment Organization

Choosing the hardware to make redundant is only half the battle. Identifying how redundant hardware will be organized into selectable groups, and creating and testing the fault protection logic to navigate the system through those possible configurations are expensive tasks. These selectable groups of hardware are referred to as "fault containment regions."

In another example of confusion over terminology, there appear to be at least three different types of fault containment regions, though projects don't rigorously distinguish between them in their documentation. The first type refers to *containment of physical damage*. The can show up in the form of a requirement that any electrical damage of a component not propagate across an interface to cause electrical damage in another component. The second type of fault containment region refers to *containment of function loss*. For example, it may be that damage does not physically propagate, but the loss of a component makes it impossible to use other components. The set of components tied together in that functional dependency would make up a functional fault containment region. The third type refers to the *operational containment* that the fault protection software can actually apply. For example, in order to reduce software complexity, several functional fault containment regions may be treated by the software as a single combined region of fault containment. Unfortunately, it is often unclear which of these forms of fault containment is under discussion during requirements and design development. Confusion over the scope of fault containment described in project requirements would be alleviated by making explicit the type of containment.

We have no specific data on whether survey respondents have taken this approach, but one strategy that has been suggested would be for a system to have *many functional* fault containment regions that are manually verified in combinations on the ground, but only a *few operational* fault containment regions in the initial load of flight software. Once a component has failed in flight, the mission can upgrade the software to partition the components *functional* fault containment region into its own *operational* fault containment region. Taking this approach would greatly reduce the number of autonomously selectable configurations that must be tested, but of course would require provisions to make these as-need upgrades during the mission.

1.2.4.4.4 Possibility versus Probability

Credit for this phrase goes to a contractor who participated in the April NASA workshop.[lxvii] Many companies use reliability analysis as an estimate of the probability of in-flight failure and pursue robustness improvements in those areas of higher failure probability. The assumption, of course, is that parts reliability is a sufficient indicator of system reliability.

One institution, based on personal experience, rejects that assumption, believing that human error defies modeling and is a major unpredictable contributor to failure. As examples, several recently lost missions (Mars Polar Lander, Mars Climate Orbiter, and Mars Odyssey) resulted from parameter or software or design flaws introduced by human error. Though of course, they weren't really a matter of failing components either. However, the recent waveguide switch anomaly on MRO is considered to be a parts failure case, one in which common wisdom was that this failure was non-credible and therefore allowed on the exemption list. One speaker at the NASA Fault Management Workshop made a special point of emphasizing that the real causes of failure are almost always human error, and consequently, our solutions must focus on the role of human error.

1.2.4.4.5 Protection Strategy

Protection strategy is a rather subtle aspect of the fault handling strategy issue and can be thought of as a matter of adopting a *function preservation* rather than *box–fixing* mind-set, even for the case of component level fault protection. The box-fixing approach focuses on fault responses concerned with restoring a box to a nominal operating state. The function preservation approach looks at the broader context of protecting the functions supported by the box. A function preservation outlook is more likely to supply component level responses with actions that protect higher level functions as well. For example, the box-fixing approach for a thruster failure might be to swap in a redundant thruster branch. The function preservation approach considers the entire function of thrusting and would also address questions such as electrical power requirements, thermal configuration requirements, etc… that go along with the thruster branch swap.

Most systems include some elements of function preservation in their fault response designs, but discussions during site visits suggested that there wasn't an explicit recognition of this point of view, and that function perseveration features in current designs come about unevenly and without systematic analysis. In one case, even though the function preservation strategy was recognized during the mission development, the design failed to ensure that it was applied, resulting in a temporarily inconsistent configuration of the catalyst bed heaters and thrusters.[lxviii]

1.2.5 Conclusions on Technical State of the Practice

The main lesson from the survey of the technical state of the practice is that practitioners build on past work without much critical examination of the assumptions and principles that underlie the task. Without that critical examination and understanding, design solutions poorly fit to new applications perpetuate, needlessly increasing the cost and risk of missions. All institutions moving into deep space missions have encountered this problem, with similar consequences.

1.3 Process State of the Practice

1.3.1 Requirements Approach

As is true for most spacecraft requirements, fault protection requirements are usually copied and tailored from a previous mission. The work of requirements development tends to follow one of several approaches across institutions: fault protection system engineering done as a part of mission assurance; fault protection done as an additional duty of systems engineering (most common approach); or fault protection as a dedicated role. Several institutions mentioned during the NASA workshop that they were now considering the dedicate role approach, following in the footsteps of other organizations. Having a dedicated fault protection engineer may improve architecture and design consistency, but there was no specific data to support that conclusion.

Historically, the high level fault protection requirements tend to focus on key interfaces and fault handling strategy, with little attention to cross-cutting features that would make systems more comprehensible, testable, and operable. Consequently, the designs resulting from those requirements are not very comprehensible, testable, or operable. Before choosing design solutions and scope of fault handling strategy, projects should consider global requirements that span the fault algorithms and govern how they operate and interact.

1.3.2 Verification and Validation Approach

It's during the verification and validation effort that a project's troubles really become apparent. Basic verification is usually completed, albeit late, which inevitably delays the start of any flight-like scenario testing. Although fault protection design activities and reviews inspire heated debate, the faulted scenario test program seems to regularly inspire the classic stages of response to tragedy: denial, anger, bargaining, depression, and acceptance. This situation is common to deep space missions with their high levels of autonomy. However, Earth orbit missions with fail safe protection and low autonomy report that this is not a common problem for them.

The deep space testing tragedy begins with the fault protection team first denying that the schedule is in jeopardy. As it becomes clear that the as-planned test program can not be completed, the team members become angry that their progress is impeded by problems that appear to be out of their control. During bargaining, they try to reduce the scope or backslide on requirements and justify the addition of more staff to get through the testing. Depression follows when they find themselves in what a NASA workshop participant described as the "fault protection death spiral," a seemingly endless cycle of discovery and software fixes with no leveling off in the rate of discovery. Eventually the team accepts that the system is not tested to the team's comfort level, and that it is better to make sure that the system can pass the existing tests than to create new ones that may uncover new problems.

We'll now look more closely at the V&V effort along three lines: V&V subject, V&V level, and test automation.

1.3.2.1 V&V Subject

For the discussion here the *V&V subjects* of interest are the framework software and the implementation software. The *framework software* consists of the core reusable services and structures upon which the fault protection software will be built. Frameworks evolve over time as institutions take opportunities to make upgrades. The *implementation software* consists of the application-specific objects and functions built upon the framework. The implementation provides the tailoring for specific missions and fault handling strategy. One organization has successfully reused implementation software as well as framework software because of their reuse of flight hardware and subsystem designs.[lxix]

1.3.2.1.1 Framework V&V

Framework V&V focuses on the global, cross-cutting mechanisms that support functions such as resource contention, pacing, global interaction rules, state control, information exchange, etc. It can be performed early, well before fault handling strategy is defined, and even before the project begins if the software is inherited or developed under a research activity. Because the framework can be tested early and separately from the project implementation, a good framework can be a powerful investment paying off across multiple projects, more so when it makes implementation easier. From the survey it appears that institutions with reusable frameworks do succeed in reducing certain types of problems over time. For example, one project using a third-generation fault management architecture[lxx] reported that uncovering and deconflicting algorithm interactions was not a memorable problem at the end of their test program. Instead they were focused on performance recovery and meeting tough critical activity timelines.[lxxi] Similarly one company's framework has enjoyed heavy reused across projects and believes that it has done much to reduce their costs.[lxxii]

1.3.2.1.2 Implementation V&V

Implementation V&V focuses on the application-specific fault protection algorithms that implement the required fault handling strategy. The ease of completing this activity and the kinds of problems encountered depend upon the architecture and the fault handling strategy scope. If the good behavior of the system relies heavily on the quality of the implementation instead of upon the infrastructure, this stage of testing ends in a death spiral campaign to uncover basic design implementation flaws. If a robust set of proven mechanisms are in place to enforce good behavior then the testing ends in exploring stress cases such as multi-fault scenarios.

1.3.2.1.3 V&V Level

V&V level refers to the level of system integration present in the software being verified and validated. Although fault tree analysis is the tool of choice for coverage verification, testing is the primary tool for fault protection V&V. Test levels progress from unit testing on stand-alone functions, through integration testing on fault protection functions operating with other flight software, through scenario testing where flight-like test situations are explored.

1.3.2.1.4 Unit Testing

Unit testing verifies that the software satisfies the mission requirements and accurately implements the specified design. A typical approach is to hand-write scripts that verify compliance with each requirement in a V&V matrix. The tests are often performed within an environment that allows the operator to manipulate the inputs to a function without the overhead of the entire flight-like environment. No particular problems have been reported with respect to the practice of unit testing.

1.3.2.1.5 Integration Testing

Integration testing verifies that the fault protection algorithms have been properly integrated with the rest of the flight software. Key points to check include confirmation that the inputs to the algorithms are supplied correctly and that the assumptions in the algorithms are consistent with the as-built (or as simulated) system. The type of interface for passing inputs to fault protection algorithms greatly affects the ease of testing. Architectures that rely on individual hand-coding of inputs to the algorithms require considerable time to check that each input is correctly prepared or even populated.[lxxiii] Architectures that use as-built sources of data, such as central deployments tied into an existing telemetry stream, require very little effort to verify inputs[lxxiv] (though they still require confirmation of assumptions).

Integration tests require a more flight-like environment than unit tests. Testing of fault monitors relies on fault injection simulation. Several participants of the NASA workshop reported that the development of the fault injection capability was a very expensive component of the fault protection task. One project reported that integration test preparation and execution required more effort than any other test activity.[lxxv] For architectures where sequences were part of the fault response architecture, verifying that the sequences resulted in the correct end states was reported as time consuming.

1.3.2.1.6 Scenario Testing

Scenario testing consists of exercising the fault protection software within flight-like mission scenarios. Since flight-like tests require a high degree of fidelity, they can be quite expensive to implement. The venues for this testing range from software-only simulations on a workstation, to the actual integrated flight system driven by simulations. We found a variety of strategies described or mentioned for designing scenario tests. Typical scenario selection techniques include focusing on vulnerable critical events, on points in time where spacecraft states change, on attaining algorithm path coverage, state, and/or data coverage, on using Monte Carlo methods, or on exploring the boundaries of performance envelopes. Scenario testing is often used as an expensive way to debug algorithm interactions.

1.3.2.2 Test Automation

During the site visits, participants observed that debugging test scripts was often as or more complicated than debugging the flight software itself. The traditional approach for creating these is to hand-write individual test cases with a list of requirements or a scenario design as a guide. However, some institutions have been exploring the use of test automation.

A number of recent missions have turned to automated testing as a means to improve test coverage. Automated testing requires more complexity in the test infrastructure, and the ease of applying automated techniques depends on how systematic the fault protection software design is. For example, if fault monitors all use a consistent, well-documented interface, then unit test scripts can be automatically generated from a database describing the interface. So, a little forethought in the design approach can greatly enable automated testing.

1.3.2.2.1 Motivations

The decision to pursue automated testing falls into at least three key camps. In the *validation desperation camp* are those projects that see no other means to validate the design other than by running large numbers of faulted scenario cases. The outcome of this situation has inevitably been the death spiral situation. It's made worse if the project lacks a good triage system to sort through the test results. The *verification efficiency camp* uses automated testing as a way to make verification more efficient. They exploit the standardization in their frameworks to automatically generate the test cases needed to verify the design requirements. The *unprejudiced validation camp* uses automated testing as a way to find interesting cases that an expert, but biased, test designer might not consider.

1.3.2.2.2 Techniques

Recent automated techniques described in the surveys and discussions include n-wise parameter variation, black box behavior assessment, Monte Carlo, model checking, traversing of fault scenarios, traversing of fault algorithm paths, and randomized search strategies. All of these techniques improved the test coverage of systems, and their practitioners say they have been valuable in uncovering implementation and design flaws.

The *n-wise parameter variation* technique derives from the observation that a large percentage of interaction problems result from combinations of just a few parameters. One project divided fault protection inputs into three categories (indices, guards, and states) according to how they affected the behavior of the function and varied one parameter from each of these categories for each test case, walking through all combinations of these categories. Another research project has demonstrated searches using pair-wise or three-wise variations of input parameters.[lxxvi]

Black box behavior assessment is a fault monitor testing technique to vary inputs according to various canned profiles and determine their effect on the output of the monitoring function. By generating steps, ramps, sinusoids, random noise, and other profiles, the test system records which of the profiles the monitor can detect, which the test engineer than compares against requirements.[lxxvii]

Monte Carlo techniques randomly generate inputs to a system according to defined distribution functions (Gaussian, uniformly random, etc…). These seem to be useful only for monitor testing where noise profiles are applied to the checked data.

Model Checking is a technique for determining whether a model of a system satisfies a set of constraints or properties. The system model can be derived from design specifications and/or from the executable software for the system. Model checking tools operate by driving the model through its reachable states, looking for cases where the constraints or properties are violated.[lxxviii,lxxix]

Traversing Fault Scenarios involves methodically exploring mission fault scenarios, varying initial conditions, and the types and timing of fault injections. The vector of faults and timing often target known tricky areas from past experience, and random failure of key components, especially those critical to S/C health and safety.[lxxx]

Traversing Fault Algorithm Paths involves methodically exploring the possible paths of fault protection algorithms, making sure that each branch of a conditional has been exercised. This confirms that those paths can be accessed and provide data to confirm that spacecraft health and safety or other requirements are met.

Randomized Searches use genetic or related algorithms to explore scenarios. They modify randomly selected parameters for each scenario and choose which parameters to carry forward for the next generation of tests according to defined optimization criteria, with the goal of finding flaws more quickly than by human selection or uniform searching. The technique can be applied to testing or as a search mechanism for model checking.[lxxxi,lxxxii]

1.3.3 Documentation Approach

1.3.3.1 Requirements Views

Industry seems to be moving from the document-based requirements approach to a database approach, notably through the DOORS requirements application. Requirements are linked level-to-level, and sometimes even down to design specification. Although there has been some research on integrated modeling and requirements approaches, this does not appear to be in practice for fault protection at this time.

1.3.3.2 Architecture Views

Mode and functional block diagrams dominate. Most do not use formal modeling that ties into any analysis tools, so the views are usually hand-crafted for the benefit of human reviewers.

1.3.3.3 Design Implementation Views

Flow charts, tables, sequence diagrams, state machine and mode diagrams, state effects diagrams, and other UML-like products are in use.

1.3.3.3.1 Flow Charts

Flow charts are probably the most common diagram. As a primary design spec approach, flow charts suffer from a completeness problem in that designers must identify all of the relevant branch points without necessarily understanding all of the state changes (which imply branches points) that are going on. A typical consequence is that a branch point omits a relevant check or that a branch point is missing entirely. These are usually discovered during testing when the uncheck condition occurs and the system behaves in a way that is undesirable.

1.3.3.3.2 Sequence Diagrams

Sequence diagrams are used to flush out particular scenarios for fault protection. Although they are good for defining simple examples of fault recovery, particularly for the purposes of external review, they lack specifications for branch points and do not usually provide a depiction of the behavior complete enough to support software implementation.[lxxxiii]

1.3.3.3.3 State Machine and Mode Diagrams

State Machine and Mode Diagrams can provide a complete description of the behavior, but the level of detail varies. At least one institution has used automatic generation (auto-coding) of flight code from state machine diagrams for a series of missions.[lxxxiv] They used a combination of commercial products (also used in the automobile industry) and in-house additions. On other occasions the diagrams served merely as a detailed design spec to be implemented by the software team.[lxxxv] A lesson learned from that auto-coding experience was that the level of detail needed for the code generate was excessive for general review purposes. It would have been better create less detailed, more abstract versions for external review purposes.[lxxxvi]

1.3.3.3.4 State Effects Diagrams

State effects diagrams have been used in proof-of-concept applications, but not for full-up flight project.[lxxxvii] These provide complete documentation of the states, their interrelationships, and constraints. However, the many-to-many nature of the view makes it difficult to draw cross-cutting or high level conclusions about the system displayed. This is similar to the level-of-detail lesson learned for the State Machine Diagrams.

1.4 Progression of Complexity

1.4.1 Mission Requirements

The top level requirements for deep space missions have largely remained the same for 30 years (ref 1): operate with limited ground contact, protect fragile systems (low margins), protect critical activities, accommodate other mission constraints, and maintain single fault tolerance. Near-earth missions, by contrast, often lack one or more of these requirements.

Philosophies on risk management differ between and even within institutions. One practices philosophy towards fault protection is that that it is better to solve the problem of creating fault protection software, than it is to ignore failure scenarios that would end the mission. Underlying this philosophy is the belief that engineers have the knowledge, techniques, and experience to succeed in developing that software. A contrasting philosophy is that the risk of introducing a mission-ending software bug due to software complexity exceeds the risk of actual hardware failure. Underlying this philosophy is skepticism that engineers have the ability to debug the software sufficiently and within schedule and cost.

Despite the constancy of the top level requirements, it is generally agreed that fault protection has become more complex! If the top level requirements have not changed much, why has complexity increased? At the very least, the character of missions has changed, technologies have changed, acceptable level of risk changes during the project lifecycle, reliability analyses for more complex systems are often ‘late’ in the project lifecycle, and new players in this field introduce new approaches and learning curves. These factors often contribute to the growth of essential complexity in new missions. In turn, because both the software approach and the lateness of development activities tend to amplify complexity, these factors contribute to the growth of incidental complexity in new missions.

Essential complexity is primarily a systems rather than software question. It is driven for example by what activities must be perform, by what functions are within those activities, by what information must be processed, and by requirements with respect to precision and speed. Essential complexity will continue to grow so long as we continue to demand that fault protection be applied to increasingly complex activities.

Incidental complexity arises during the process of creating the software to accommodate the essential complexity. In a perfect world the resulting software would have the minimal amount of complexity needed to satisfy the essential complexity. In practice software development acts as a sort of multiplier for the essential complexity. Its effect is made worse by failure to do appropriate hardware/software trade studies.

What factors contribute to the growth of incidental complexity? We see that choice of software architecture, skill and experience of software team, utility of development tools, opportunity and resources to make good fixes versus quick fixes, and avionics choices that limit software capabilities. Incidental complexity will grow along with growth in essential complexity if we continue to do business the same way.

1.4.2 Institutional Fears

Each institution enshrines a set of fears from past lessons learned into practices. Here are some example fears:

- Fault protection software is more likely to end a mission than actual hardware failure.
- It’s too risky to allow unsupervised firing of thrusters.
- Don’t power on backup hardware until you plan to use it or calibrate it.
- Toggle controls are bad.
- Tables are safer than custom code.

Often the lessons learned are superficial and address a symptom rather than the underlying cause or principle. The consequence is superficial practices that may work for the current set of applications, but are inappropriate for different applications. There doesn’t seem to be a system in place to “retire” fears, to cross-check that they are still applicable, or to make sure the correct underlying lesson has been identified. Consequently decisions made to transfer complexity between the flight and ground system and to avoid improved frameworks that make implementation more robust are based on what may eventually become superstition.

2 Recommendations — What next?

2.1 Summary

Making recommendations on a topic as controversial as fault protection is not easy. In this section we'll look at steps that are practical as well as useful. The reality is that even *within* institutions there is lack of consensus, so following through on these recommendations will pose a serious challenge to the entire industry. The earlier state-of-the-practice discussions went into great detail on the pros and cons of various techniques, so we leave it to individual projects and institutions to use that information as an aid for choosing their technical and process approaches. Here, however, we make recommendations that speak to the fundamental programmatic and decision-making processes in the fault protection arena.

One special topic item explicitly called out in this task is the investigation of reducing fault protection software as a separate entity. We'll examine some history and the current practice for that as well.

2.2 Fault Protection Recommendations

2.2.1 Finding #1: Standardize

There is confusion in the terminology and even in the basic concepts related to fault protection. The industry lacks good reference material for which to assess the suitability of fault protection approaches. This fosters inconsistent scoping of fault protection and confusion over what the scope actually is.

To remedy this it is recommended that NASA publish a NASA Fault Protection Handbook or Standards Document that provides a recognized lexicon for fault protection and a set of principles and features that characterize software architectures used for fault protection. For existing and past software architectures, it should provide a catalog of recurring design patterns with assessments of their relevance and adherence to the identified principles and features. A good outcome would be a well-received text that practitioners willingly describe as the "Bible of Fault Protection."

2.2.2 Finding #2: Improve Reviews

The proposal review process does not assess in a consistent manner the risk entailed by a mismatch between mission requirements and the proposed fault protection approach. This leads to late discovery of problems.

To remedy this it is recommended that each mission proposal include an explicit assessment of the match between mission scope and fault protection architecture. Penalize proposals or require follow-up for cases where proposed architecture would be insufficient to support fault handling strategy scope. At least one recent mission recognized the fault handling strategy scope problem, but did not appreciate the difficult of expanding fault handling strategy using the existing architecture. The handbook or standards document can be used as a reference to aid in the assessment and provide some consistency. A good outcome would be specific believe that a project's architecture to support fault protection is up to the job.

2.2.3 Finding #3: Educate

Fault protection and autonomy receive little attention within university curricula, even within engineering programs. This hinders the development of a consistent fault protection culture needed to foster progress and the ready exchange of ideas.

To remedy this it is recommended that NASA sponsor or facilitate the addition of fault protection and autonomy courses within university programs, such as a Controls program. Example: University of Michigan could add a “Fault Protection and Autonomy Course.” Other specialties such as (circuit design) have developed specialized cultures that allow for continuing improvement. Autonomy and fault protection should do the same. A good outcome would be clarity and understanding in discussions of fault protection among different institutions and less need to recruit new practitioners from outside of the field.

2.2.4 Finding #4: Define Policy

Projects often set fault protection scope policy (such as single point failure policies) with less critical review from NASA than other requirements, despite broad implications on cost and schedule. This leads to inconsistent risk postures for similarly classes of missions across organizations and expensive deferment of decisions due to lack of consensus within projects.

To remedy this it is recommended that NASA include policies for fault handling strategy and activity resumption as part of the Program requirements. This will improve the alignment of the fault protection capability with the priorities that NASA has for the respective missions. Some strawman examples are shown below:

> *Example Policy: The project shall apply redundancy for mission critical areas in which estimated reliability is less than half of the system norm and for which the cost of redundancy does not exceed X% of planned mission cost.*
>
> *Example Policy: The project shall adopt the following priorities for fault management during critical activities: (1) Continue execution of the scheduled activity; (2) Isolate faults to the box level; (3) Restore critical system functionality and resume critical activities.*

A good outcome would be clear agreement on the scope of fault protection and a good fit of the fault protection development effort within the project cost and schedule.

2.2.5 Finding #5: Assess Frameworks

There is a lack of consensus on the relative risks and merits of *increasing software framework complexity* that promote simpler behavior and implementation versus *using simple frameworks* that promote complex software implementations with difficult to predict behavior. For example, assembly code provides a simple framework to support complicated software implementations with boundless behavioral complexity, while a database language provides a complex framework to support simple applications with very constrained behaviors. For an application developer with access to a suitable framework, the former approach is less expensive.

To remedy this it is recommended that each project assess this trade off, consulting industry experts familiar with software architecture in a context broader than just embedded flight system software development. The assessment must include the testability of the end product and what architectural provisions would improve the bottom line cost and schedule, not just of an individual project, but across projects. Good outcomes would be a development and V&V effort not overwhelmed by the implementation complexity, and the completion of early validation through appropriate frameworks.

2.2.6 Finding #6: Include Architectural Improvements as Part of Contracts

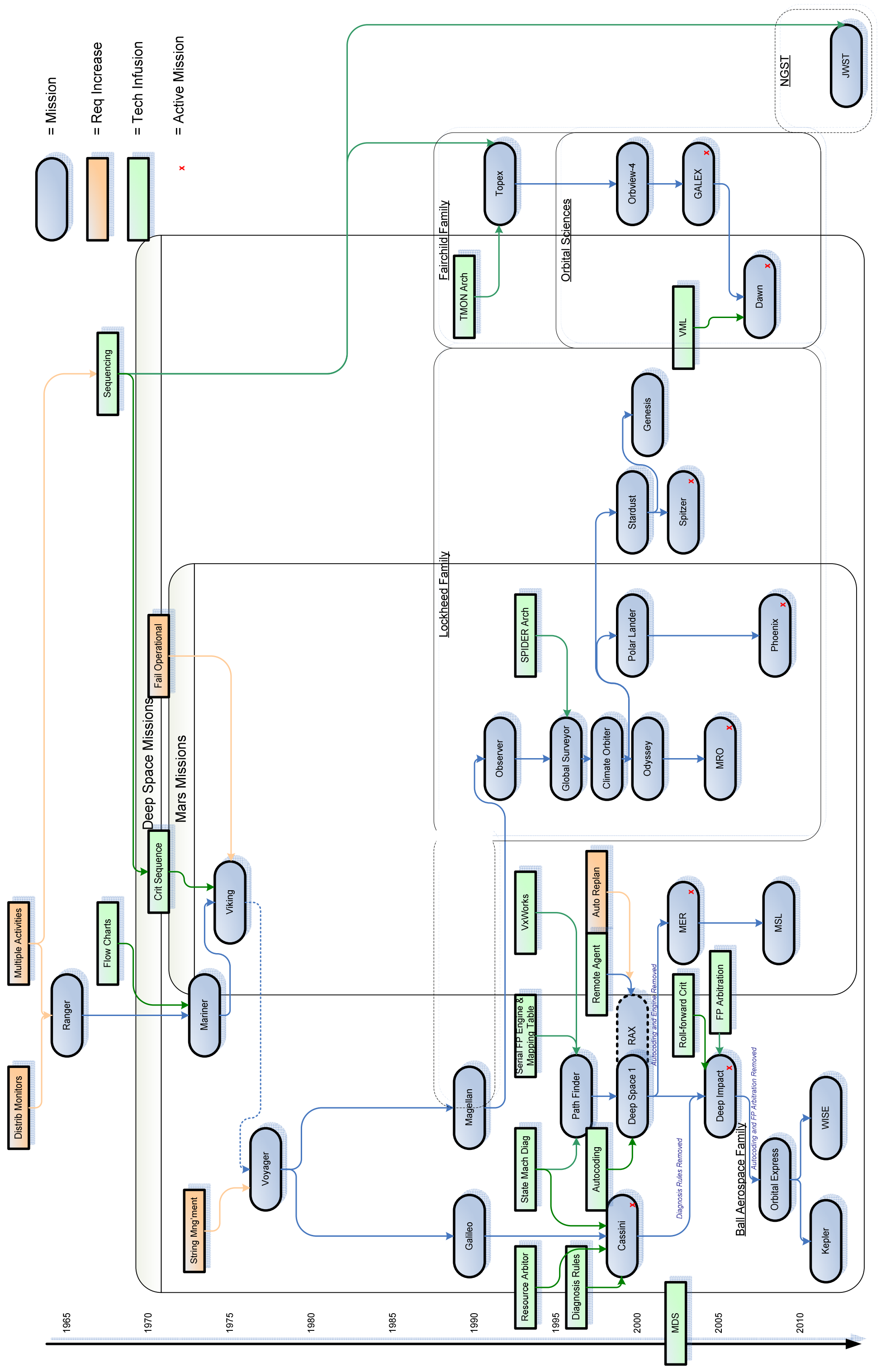
NASA contractors include financial gain as a key motive in their pursuit of work from NASA. Unless their heritage software fails to meet mission requirements they will regard pressure to change their architectures as a “make better” situation unlikely to improve their financial bottom line. If NASA wants to improve the state of the practice, it should either define project requirements that the existing architectures cannot meet, in which case the contractor would come back with an associated upgrade cost to meet the requirements, or NASA should include software improvement as a specific component of the contract.

2.3 Fault Protection Software as a Separate Entity

Although this topic is a line item in the fault protection software complexity task, it has been addressed in a separate sub-task and is not repeated here.

3 Fault Protection Family Tree

How did fault protection evolve?



4 Fault Monitoring Architectures

	Practitioner:	Lockheed	Goddard	Orbital		APL	JPL					Ball Aerospace
	Family:	"Spider" + VML	TMON			Rule-Based	Parallel State Machines	Smart Sequences	Local Software Logic	Reusable Fault Protection Framework		
	Missions:	MRO / Phoenix / MSL / MO		Dawn	GALEX	New Horizons / Messenger	Cassini AACS	Cassini CDS	MER	Pathfinder / DS-1	Deep Impact	Kepler / WISE / Orbital Express
Fault Monitoring	Deployment	Local	Central	Central	Central	Central	Local	Central	Local	Local	Local	Local
	Statistics Generation	TBD	Central=Yes Local=No	Central=Yes Local=No	Central=Yes Local=No	TBD		TBD				TBD
	Behavior Specification	Table	Table w/ Preprocess	Table w/ Preprocess	Table w/ Preprocess	Table of Logic	Custom LogicCode	Custom LogicCode	Custom LogicCode	Custom LogicCode	Custom LogicCode	Custom LogicCode
Fault Diagnosis	Combining Detections	Local	Central	Central	Central	Central	Central	Central	Local	Local	Local	Local
	Mapping to Response									Central	Central	Central

5 Fault Response Architectures

	Practitioner:	Lockheed	Goddard	Orbital		APL	JPL					Ball Aerospace
	Family:	"Spider" + VML	TMON			Rule-Based	Parallel State Machines	Smart Sequences	Local Software Logic	Reusable Fault Protection Framework		
	Missions:	MRO / Phoenix / MPL / MO		Dawn	GALEX	New Horizons / Messenger	Cassini AACS	Cassini CDS	MER	Pathfinder /DS-1	Deep Impact	Kepler / WISE / Orbital Express
Fault Response	Deployment	Local	Central	Central	Central	Central	Central	Central	Local	Central	Central	Central
	Thread Control	TBD	Parallel	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
	Behavior Selection	Table-Defined Tiers	Table-Defined Tiers	Table-Defined Tiers	Table-Defined Tiers	Macro Logic	State Machines withTiers	Table-Defined Tiers	TBD	State Machines withTiers	State Machines withTiers	TBD
	Behavior Pacing	TBD	Monitor Persistence	Monitor Persistence	Monitor Persistence	TBD	Response Delay Logic	TBD	TBD	TBD	Response Delay Logic	TBD
	Primary Means of Command Execution	Sequenced	Sequenced	Sequenced	Sequenced	Sequenced (Macros)	In-line	Sequenced	In-line	In-line	Sequenced	Sequenced
	Responsiveness to System State	Via Sequence Syntax	N/A	N/A	N/A	Via Rule syntax	Via Response Code	Via Sequence Syntax	Via Response Code	Via Response Code	Via Response Code	TBD

6 Notes

[i] Cassini AACS and Deep Impact
[ii] Path Finder and Cassini families.
[iii] TMON family
[iv] Goddard Earth Orbiters
[v] TMON, Cassini, Path Finder families
[vi] TMON family
[vii] TMON family
[viii] Cassini and Path Finder families
[ix] Ball Aerospace family and Lockheed SPIDER family
[x] SPIDER and TMON families
[xi] TMON family
[xii] Dawn mission
[xiii] Anecdote from GRACE mission
[xiv] Cassini and Pathfinder families
[xv] Cassini and Dawn missions
[xvi] Pathfinder family
[xvii] Cassini and Macro families
[xviii] Cassini, Deep Impact, SPIDER
[xix] SPIDER
[xx] Cassini mission, Pathfinder family, TMON familty
[xxi] Macro family
[xxii] MDS
[xxiii] Deep Impact
[xxiv] Pathfinder family
[xxv] Deep Impact
[xxvi] Deep Impact
[xxvii] TMON and Macro families
[xxviii] Dawn
[xxix] Cassini AACS and MER families
[xxx] Deep Impact
[xxxi] SPIDER, Cassini AACS, and Deep Impact
[xxxii] MDS
[xxxiii] SPIDER, Cassini, and Deep Impact
[xxxiv] SPIDER, Cassini, and Deep Impact
[xxxv] Macro family and Dawn
[xxxvi] Cassini AACS
[xxxvii] Pathfinder, Macro and TMON families
[xxxviii] Cassini AACS and Pathfinder family
[xxxix] Cassini CDS and SPIDER
[xl] Pathfinder family
[xli] TMON and Macro families
[xlii] Dawn mission
[xliii] SPIDER, TMON, and Macro families
[xliv] Cassini AACS and Deep Impact missions
[xlv] Cassini AACS and Pathfinder family
[xlvi] Deep Impact
[xlvii] Deep Impact mission and TMON family
[xlviii] SPIDER (via VML) and Macro families
[xlix] Cassini AACS, MER, DS-1, Pathfinder missions
[l] Orbital family with VML for activities and RTS for fault protection

[li] Deep Impact
[lii] SPIDER
[liii] Cassini AACS and Deep Impact mission
[liv] Dawn
[lv] MDS
[lvi] TMON and Macro families
[lvii] Orbital, Ball Aerospace, and NGST emergency mode controller design
[lviii] Deep Impact
[lix] Cassini, SPIDER, Deep Impact, New Horizons
[lx] Cassini AACS, Deep Impact missions, Dawn
[lxi] NASA, "NASA's Mission Classes for Solar System Exploration," http://solarsystem.nasa.gov/missions/future4.cfm.
[lxii] Cassini, Deep Impact, Dawn, New Horizons, Mars Orbiters
[lxiii] MER, DS-1, Pathfinder, MSL
[lxiv] JPL
[lxv] Orbital, Ball Aerospace, NGST emergency mode
[lxvi] Dawn and SIM
[lxvii] Orbital / Rustick
[lxviii] Deep Impact
[lxix] SPIDER family
[lxx] Pathfinder family
[lxxi] Deep Impact fault protection had difficulty finding the right recipe to converge the attitude estimator well enough to allow accurate AutoNav performance.
[lxxii] Lockheed's SPIDER
[lxxiii] Deep Impact
[lxxiv] TMON family
[lxxv] Path Finder family
[lxxvi] Friedman, "Diagnosis Combining and Design Knowledge"
[lxxvii] Deep Impact mission
[lxxviii] Barltrop, "Model Checking Investigations for Fault Protection System Validation," International Conference on Space Mission Challenges for Information Technology 2003.
[lxxix] Feather, "Model-Checking for Validation of a Fault Protection System," Sixth IEEE International Symposium on High Assurance Systems Engineering, 2001.
[lxxx] Barltrop, "Automated Generation and Assessment of Autonomous Systems Test Cases," IEEEAC paper #1228, Version 2, Updated October 22, 2007.
[lxxxi] JPL Nemesis project.
[lxxxii] Arslan, "A Genetic Algorithm for Multiple Fault Model Test Generation for Combinatorial VLSI Circuits," GALESIA '97.
[lxxxiii] SPIDER, JPL SIM project
[lxxxiv] Deep Space-1 and Deep Impact missions
[lxxxv] Cassini AACS
[lxxxvi] Deep Impact mission
[lxxxvii] Next Generation DSN demonstration

Final Report

NASA Study on Flight Software Complexity

Appendix G — GN&C Fault Protection Fundamentals[1]

Robert D. Rasmussen,
Jet Propulsion Laboratory,
California Institute of Technology

[1] This paper was published in the proceedings of the 31st Annual AAS Guidance and Control Conference, February 1-6, 2008, Breckenridge, Colorado.

Contents

Executive Summary

Addressing fault tolerance for spacecraft Guidance, Navigation, and Control has never been easy. Even under normal conditions, these systems confront a remarkable blend of complex issues across many disciplines, with primary implications for most essential system functions. Moreover, GN&C must deal with the peculiarities of spacecraft configurations, disturbances, environment, and other physical mission-unique constraints that are seldom under its full control, all while promising consistently high performance.

Adding faults in all their insidious variety to this already intricate mix creates a truly daunting challenge. Appropriate tactical recovery must be ensured without compromise to mission or spacecraft integrity, even during energetic activities or under imminent critical deadlines. If that were not enough, the consequences of a seemingly prudent move can have profoundly negative long-term consequences, if chosen unwisely, so there is often a major strategic component to GN&C fault tolerance, as well. Therefore, it is not surprising that fault protection for GN&C has an enduring reputation as one of the more complex and troublesome aspects of spacecraft design — one that will only be compounded by the escalating ambitions of impending space missions.

Despite these difficulties, experience has suggested methods of attack that promise good results when followed consistently and implemented rigorously. Upon close scrutiny, it is strikingly clear that these methods have roots in the same fundamental concepts and principles that have successfully guided normal GN&C development. Yet it is disappointing to note that the actual manifestation of these ideas in deployed systems is rarely transparent. The cost of this obfuscation has been unwarranted growth in complexity, poorly understood behavior, incomplete coverage, brittle design, and loss of confidence.

The objective of this paper is to shed some light on the fundamentals of fault tolerant design for GN&C. The common heritage of ideas behind both faulted and normal operation is explored, as is the increasingly indistinct line between these realms in complex missions. Techniques in common practice are then evaluated in this light to suggest a better direction for future efforts.

ACKNOWLEDGEMENTS

This work was performed at the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration. I thank Matthew Bennett, Mark Brown, Dan Dvorak, Lorraine Fesq, Kenny Meyer, and Dave Nichols for their helpful comments.

1 Introduction

Volumes have been written about how to do GN&C, when everything is more or less right. It is an incredibly rich topic steeped in physics, math, engineering discipline, and decades of practice. Rather less though has been written about how to make GN&C work, even when things go seriously wrong. That's understandable. Success is hard enough without adding the complication of faults. Yet even without faults, it takes arbitrarily more effort to guarantee all is just right than to tolerate imperfection. Faults or not then, making things work, even when things aren't right, is really the essence of engineering.

Conventional engineering accounts for normal production variations, environment uncertainty, wear, and other meanders within the engineering design space. In GN&C, these are things like misalignments, measurement and disturbance noise, and small modeling simplifications. One typically characterizes such variations through sensitivity analyses of various sorts around selected design points, striving to make a system that serves its intended function and performs correctly within these expected tolerances. This aspect of conventional design can be thought of as variation tolerance.

Fault tolerance is similar, but with a larger scope. Variations handled through fault tolerance are generally more extreme, precipitous, or dangerous than normal variations; and because they involve departures from intended functionality and correct performance, tolerating them tends to involve higher or broader design accommodations than do normal variations. Even so, the basic issues are the same: finding ways to assess and contain the effects of variation, such that functionality and performance are preserved.

Given this kinship, one might expect to see a close family resemblance when comparing the design practices of conventional variation tolerance with those of fault tolerance. Regrettably, it often isn't there. Conventional GN&C brings a mature understanding of dynamics and statistical modeling, measurement and estimation, control, planning, optimization, and other design elements — in each case grounded in solid theoretical foundations. But fault tolerant GN&C has a different story to tell. Many lessons have been learned over many years and many projects (usually something like, "Don't do that again"), but progress has been slow. Theoretical grounding for fault tolerance, as generally practiced, has significant ground to make up in comparison to its conventional cousin. This is the central theme explored here. Fundamental issues of fault tolerant GN&C design are considered, with the goal of reiterating basic guidelines commonly understood by those working in this field. The larger objective, however, is to underscore the conceptual bonds between conventional GN&C functions and GN&C fault tolerance with the hope that this suggests a direction for further growth in the latter.

This is of particular interest now, because in many ways we have reached the end of an era, where it might be said that customary methods have been carried to their logical extreme. In fact, by some assessments, standard fault tolerant design is in crisis, as the same litany of problems recurs on project after project (late delivery, fragile behavior, poor operability, incomplete testing, and so on) — and this is *before* considering the implications of new mission types that will push even harder. Solutions are not likely to come merely by polishing existing methods, starting them earlier, integrating them better, or wrapping them within tighter processes. The roots of the problem are deeper than this, so a different path is required.

There are many sides to this puzzle. Identifying hazards and evaluating risks are clearly important, as are redundancy and cross-strapping trades, sensor selection and placement, verification processes and methods, preventative maintenance, and other related topics. However, for the sake of brevity and focus, the emphasis here is on the fault protection[2] elements themselves — the integrated control functions, active in a live system, that make the difference between failure and success, when things go wrong.

[2] The term "fault protection", as used here, is essentially equivalent to "fault management", "safety and health management", and similar terms in common use. It can be taken broadly or not, to encompass a variety of topics. A middle-of-the-road view is adopted here to focus on online capability, while broadening consideration to implications on general system operation.

2 The Fault Protection Problem

The colloquial view of fault protection is simple. Detection of fault X triggers corrective action Y (e.g., a redundancy swap). Then the system just tidies up a little and reports home. Why isn't fault protection always this easy? Well, from a cursory view of architectures deployed against faults in space systems, one might get the mistaken impression that this is indeed all there is to it. Except where active redundancy is used (common in launch vehicles, for example), typical fault protection architectures consist of a monitoring system for detecting when something is not right, linked to a response system to isolate the fault and either retreat to a safe mode or restore the lost function (sometimes both). Topics abound when discussion turns to fault tolerance in general, but *monitors* and *responses* are still the dominant theme in actual implementation, leaving the general impression that fault protection is mainly about base reflexes, like pulling your hand away from a hot stove. Even the few supporting functions typically follow this pattern. For example, responses are often aided by a redundancy management system to facilitate selecting backups. In addition, a logging system records key details, such as which monitors tripped at what time, how severely, what response was triggered, etc. Of course, there are also the obligatory flags to individually turn monitors and responses on or off. The monitor-response theme though is ever present, describing, more or less, most systems that have ever flown in space.

Another common aspect of fault protection implementation is that this collection of monitor-response functions usually appears as a distinct appendage to normal functionality: monitoring functions generally eavesdrop on existing data streams; response systems commonly usurp existing command-sequencing functions; redundancy is often masked from normal functions through virtualization of some sort — largely the separate developments of a fault protection team. As fault protection experience has accumulated, the importance of early integration into the larger system has become apparent. Nonetheless, with few exceptions, the fault protection task and its associated architecture have tended to remain largely detached.

In this conventional picture, most fault protection architecture (as distinct from fault protection design or implementation) resides primarily in the monitor-response machinery. This may seem like a shortsighted assertion, but major architectural features peculiar to fault protection or embodying fault protection principles are generally scarce, outside the monitor-response structure.[3] For example, the resumption of disrupted critical activities, although usually considered a substantial part of the fault protection engineers' task, is nonetheless implemented separately in most systems from the monitor-response architecture; and it generally requires custom design (e.g., command sequences or state machines) with at most trivial specific architectural support identifiably dedicated to matters of fault recovery. Similarly, when the time comes for fault protection verification (regularly a time of revelation, it seems), focus tends, at least initially, to descend from a global view back to monitors and responses in the "touch all paths" spirit of testing, with the hope that this is somehow adequate. For such reasons, anything outside the monitor-response system is routinely treated as something other than fault protection architecture, at least from an implementation point of view. The handful of exceptions usually

[3] Even "rule-based" architectures generally offer little more than generic programming support to the monitor-response paradigm, with nothing intrinsic to say about fault protection, the issues surrounding it, or the principles supporting it.

consists of simple retry mechanisms, data error masking, filtering to tolerate temporary data outages, and other highly localized measures of that sort — all mostly ad hoc, thus with little supporting architecture.

Given this unadorned picture of fault protection as something with boundaries, distinct from the rest of the system, it has become a common management exercise then to count the number of fault monitors and responses to get some idea of the scope of “the fault protection problem”. Invariably this number is assessed as “too high” — designers trying too hard, too much redundancy, too many verification tests, and so on — and *that*, it is often said, is why fault protection is complex. Neglected in such simplistic accountings though are the abundant ways in which real fault protection departs from the simplistic orthodoxy suggested by monitor-response architectures. Common issues are listed in Figure G.1.

- Many scattered symptoms may appear concurrently from one root cause, often masking the real culprit.
- Some faults may be hard to distinguish from normal operation, yet still cause long-term trouble.
- What appears to be wrong during one mission phase may be expected or acceptable in another.
- There may be several possible explanations for observed difficulties, each with different ramifications or requiring competing responses.
- An apparent fault in one area may in fact be due to a sensor fault in another area.
- A fault may not manifest itself when it occurs, emerging only much later, possibly during a subsequent unrelated emergency.
- An apparent problem may not have a previously identified cause or may not express itself in an anticipated manner.
- False alarms may provoke responses that are as disruptive as a genuine fault, so there is pressure to compromise on safety.
- Misbehavior may be a consequence of design or modeling errors, including failure to address certain system-level interactions.
- Environmental or operational extremes may be pushing a system beyond its design range.
- Operator errors may be the primary cause of a fault, creating a conflict between doing what is directed and doing what is right.
- Overt action may be needed to diagnose a problem before corrective action can be decided.
- Complex incremental tactics may be needed to identify and isolate faults, or recover safe operation.
- Faults may create an urgent hazard to safety or critical operations that must also be handled at the same time.
- Concurrent response actions may be necessary, with potentially shifting priorities and conflicting demands on the system.
- Many more ways may be needed to restore operation after a fault than the typically small number of ways needed to perform the same operation normally.
- Functional backups, when full redundancy is not available, may involve degraded modes or characteristics of operation with broad implications on subsequent activities.
- Fault containment may not be effective, resulting in secondary faults that must also be handled.
- Isolation mechanisms may be insufficient to eliminate all adverse effects, even when redundancy is present.
- Other functions (often by design) may be masking symptoms or interfering with recovery.
- Quirks or omissions in underlying functionality may conspire against the accomplishment of otherwise reasonable actions.
- Late breaking developments in other system and software areas can invalidate fault protection models or tests, disrupting or invalidating V&V efforts.
- Actions must be chosen to preserve resources and maintain mission opportunities, even under severe deadlines or other constraints.
- Control authority needed to ensure safety and restore operation may be compromised.
- Timely responses may require substantial anticipation in order to ready potentially required assets and alternative tactics.
- The long-term consequences of short-term expediencies may be dire.
- Essential activities may need to be resumed, even though time or capability has been lost.
- Even when a mission is lost, it is generally important to attempt reporting the problem to operators — often a daunting task on its own.
- Extended autonomous operation under adverse or degraded conditions may be needed, yet still requiring a large portion of the system’s functionality at some level.

Figure G.1. Issues Routinely Complicating Fault Protection Design.

Sorting one's way through this maze is not easy, even with few failure modes or limited redundancy. However, with little beyond the basic apparatus of monitor-response architectures for support, fault protection designs tend to require a lot of clever improvisation. In the resulting confusion of ad hoc solutions, fault protection systems can become overly complicated, difficult to understand or analyze, capable of unforeseen emergent behaviors (usually for the worse), impossible to test thoroughly, and so brittle that the suggestion of even trivial change is enough to raise alarms.

2.1 Loss of Architectural Integrity

These problems are all signs of lost architectural integrity: the absence of conceptual grounding and regular patterns of design that embody and enforce fundamental principles of the discipline. In the worst of circumstances, where these issues are not under control, fault protection may even be detrimental to the reliability of the system *in defiance of its very purpose*.

This shouldn't happen. Anyone familiar with thoughtful, well-run fault protection development appreciates the profound implications of this effort on the quality and robustness of the overall design, even when everything is normal. It is rare to find anyone more intimately familiar with the system *as a system* than fault protection engineers. Their job by necessity requires them to understand the functionality of the whole system, the objectives it must fulfill, the scenarios in which it operates, its quirks of behavior, the intricacies of interaction among its parts and with its environment, the constraints within which it will perform correctly, the management of its resources, the ways in which it will be operated, and the design of the software that controls it. The sheer effort of assembling this picture into a coherent whole, when done properly, is undoubtedly a great contribution to systems engineering. Fault protection designers then take this a step further by extending the boundaries of system understanding into regions outside the design space, making changes that improve design robustness even under normal conditions. Fault protection, *done right*, greatly improves the safety and reliability of a system, whether or not a fault ever occurs.

The difficulty we face, then, is not one of finding ways to substantially reduce the scope of the fault protection task, for the merit of this activity is generally appreciated, even when its complexities are not. Rather, what we need is a way to make sure the right things happen on every project. Unfortunately though, having accepted that fault protection is hard, emphasis in the search for solutions has tended to shift generically to software engineering, primarily viewing the need as one of improved specifications, more reliable design methods and tools, tighter configuration management, better verification, and other measures aimed at process rather than architecture. There is surely no questioning the merit of such an enterprise, since concerns over unmanaged software complexity and consequent reductions in reliability are valid (fault protection being only one of many contributors). The message here, however, is that better software engineering is not enough — and in fact, doesn't even touch the topic of fault protection itself.

To preserve architectural integrity, which in the final analysis is a precondition for all other measures, there must first be Architecture. Not just any architecture, but Architecture that is up to the challenges of fault tolerance in all its dimensions. Otherwise, recurring appeal to ideas that have predictably fallen short will guarantee that each new system struggles with the dangerous irony of unsafe fault protection.

2.2 Fault Protection as a Principled Control Function

So, what do fault protection woes have to do with GN&C? From one point of view, GN&C merely compounds the problem. Due to its unusually many and diverse interactions with other system elements, GN&C tends to be involved in half or more of all the fault protection implemented on space vehicles. Issues of problematic system-level interactions, which plague many designs, increasingly find GN&C in their midst. Meanwhile, supplying robust GN&C capability hasn't been getting any easier, as ever more complex missions arise, calling for better performance and more autonomy in new uncertain environments. It is clear that good fault protection will endure as a vital ingredient in meeting these needs, so any solution to the fault protection problem as a whole will naturally strengthen GN&C as a key constituent.

The essential question here though is whether the reverse is also true. That is, are there principles to be learned from GN&C that might shed light on the fault protection problem? To be clear, the notion is not to address GN&C fault protection alone, with the condescending notion that all other fault protection problems fall by extension. **Rather, the idea is to view fault protection as fundamentally a control problem, the principles of which have been happily refined through GN&C and similar system control disciplines.** In this approach lies the benefit of structuring fault protection architecture through the same well-established patterns that have aided GN&C, but which go beyond present common practice in fault protection. With these principled patterns in place it also becomes quite clear that fault protection belongs as an integral and harmonious part of a collective approach to system control and operation, not as an appendage. This insight is essential for any attempt to establish an architectural foundation for further advances in autonomy — fault protection or otherwise. Yet this is by no means a new idea. Thus, there is a certain irony that this potential has yet to be broadly exploited, especially in GN&C fault protection.

Fundamental, stable patterns and sound principles are at the heart of good architecture. The foundations of these patterns and principles must be deeply understood so that departures from them can be recognized, if we are to deploy systems with architectural integrity. Let's take a quick look at those guiding GN&C.

3 Core Concepts

The conceptual roots of modern GN&C lie deep in systems theory[4], which concerns itself with the nature of behavior and interactions in complex systems. It is very much focused on the properties and behavior of the whole, recognizing that any description of the system at lower levels through a reductionist approach is necessarily incomplete. This acknowledgment gives rise to the idea of *emergence*. In fact, a system without emergent behavior is only trivially a proper system by usual definitions.

Naturally, this raises questions regarding the extent to which such behaviors are to be engineered, rather than merely endured. Unfortunately, many system designs leave key issues of emergence to chance. Engineers who have found themselves in the midst of a systems integration task that is as much an exercise of discovery as of verification will recognize the sinking feeling brought on by this realization. That sentiment certainly echoes the theme here. Hence, to the extent systems theory has anything to say about the topic, it is worth a brief reiteration of the basic ideas and terminology, despite their familiarity. This indulgence will be repaid shortly, as the relationship of these ideas to the topic of GN&C fault protection becomes clear.

3.1 Basic Ideas and Terminology

3.1.1 State and Behavior

Systems theory begins with the essential notion of a system as a dynamical entity, something that changes over time. All changes are reflected in characteristics referred to as *states* (things like attitude, temperature, operating mode, and so on). States describe what can change, but not what changes are possible: the principal interest of systems theory. The latter are captured in *behavior*s, the rules, constraints, or other “laws” that determine which histories of system state over time are possible. Descriptions of behavior are often referred to as *models*. State and behavior are complete descriptions of the dynamic aspects of a system — no exceptions.

3.1.2 Hierarchy and Scope

From systems theory we also get our basic notions of system decomposition and *subsystems*, *interfaces*, and *hierarchies*. All references to “system” here include subsystems anywhere in a hierarchy. A key concept in this view is that most interesting systems, and essentially all subsystems, are *open*, being subject to external influence (i.e., not *closed*). Absence of any obvious closure raises the question of system *scope*: deciding what is properly inside a system and what is not. Given the openness of systems, issues of emergent system behavior do not resolve themselves as the circumference of consideration is widened, so there really is no correct answer to question of scope. One must choose.

[4] Historically, systems theory appeared as a generalization and consolidation of earlier ideas from dynamics, control, filtering, communications, and so on. This term is taken loosely here to correspond with notions from cybernetics, control theory, signal processing, and others. Precise alignment with any of these domains of thought is not essential to the central points of this paper.

3.1.3 Objectives

When a system is directed toward some purpose, it is said to have an *objective*. Other recognizable words describing this vital notion from our systems engineering vocabulary are function, task, target, role, responsibility, aim, goal, intention, constraint, plan, etc. Like behaviors, objectives constrain the history of state, but rather than constraining what changes of state are possible over time, they constrain what changes are acceptable. Saying a system has acceptable behavior is equivalent to saying it meets its objective. A system that *fails*, therefore, is necessarily one that violates its objective, and conversely.

3.1.4 Control

The deliberate exercise of influence on an open system to achieve an objective is called *control*. In composite systems, it may be possible to exercise control on a subsystem only indirectly via chains of influence. Even if directly controllable, a subsystem may be disturbed by other such chains of influence. Engineered systems generally have some identifiable structure in their composition that tips the balance in favor of meeting objectives. The hierarchy of *functional decomposition* is immediately recognizable in this picture. Controlling a system generally requires managing influence through many, if not most, of its chains of influence. Manipulating these chains of influence directly is called *open loop* control. This is complicated considerably by random influences, and by a web of interactions that is far more tangled in typical systems than any top-down functional decomposition alone would ever indicate. Addressing this generally calls for additional structure.

3.1.5 Control Loops

The fundamental response to such complications is the introduction of negotiated, closed loop control. Closing loops is done by augmenting a controlled system with a *control system*, which applies *knowledge* of the state and behavior of the system under control to determine appropriate influences on it, so that it will meet its objective. This pairing is called a *closed loop* system. By regularly adjusting to observations of the controlled system as it is, the control system provides variation tolerance, as defined earlier, making the closed loop system more predictable than the behavior of the underlying system being controlled. Moreover, since a closed loop system is influenced by a direct statement of its objective, the apparent complexity of the original system from an external point of view is substantially hidden. (That's important, because closed loop systems are still open systems.) *Negotiation* arises from the normalization of direction to closed loop systems, whereby all external entities wanting to influence a controlled system element can do so through statements to its control system expressed in the common language of objectives (i.e., constraints on state). Since objectives express intent, they may be compared, in order to detect conflicting intents, and combined, in order to meet shared intent. The normalized behaviors of closed loop systems make prediction, planning, and coordination easier to do with confidence and efficiency, and hence easier to automate, if desired. A system operating on these principles is said to be *goal-based*.[i]

3.1.6 Cognizance

Through its knowledge of state, behavior, and objectives, a control system is in some meaningful way cognizant of the system it controls. Such knowledge may not appear overtly in a design, but it is present in the sense that, were state or objective to change, the control system would respond to it, and were behavior to change, control system plans or design would have to be revisited. In its cognizance role a closed loop control system solves the major problems of *how* to achieve an objective, while hiding these details from external entities. However, it leaves behind the vital problems of understanding *what* objectives mean exactly, *what* objectives are actually plausible, and *what* to do when objectives fail.

The key to addressing these concerns is *transparency*. **When we put a system together, it should be clear which basic control concepts are being applied and whether basic principles in their use are being followed.** The premise here is that these concepts and principles are in broad, productive use within normal GN&C, so making them transparent in GN&C fault protection should accrue similar benefits.

3.2 Making These Ideas Transparent

A control system needn't be too sophisticated before basic ideas of state, behavior, and objectives become more transparent. Even in simple early space systems, where control was exercised though manipulation of sensor biases, control systems were operating at a commensurate level of cognizance. For example, an objective presented to an attitude control system might have been to achieve a particular sun sensor output voltage (corresponding to some angle). Nonetheless, accomplishing this required knowledge of the voltage, through measurement, and use of an association between voltage, attitude, and thruster derived torque, which amounted to knowledge of behavior (if only in the designers' minds).

In more sophisticated systems, it is common to see models and the state we care about appear explicitly, as for instance gravitational dynamics in orbit determination. Likewise, objectives may be more clearly articulated, as in target-relative pointing. The merits of such approaches are well established, having arisen during the early years of systems theory development. Therefore, the elementary notions outlined here have deep and broadly understood implications that hardly bear reciting, especially in the context of GN&C. Somehow though, these ideas still seem to lose focus in many designs, especially in fault protection (and system management in general), so it is important to reiterate them. Here are some of the basic ideas.

3.2.1 Transparency of Objectives

Given the definition of behavior, an objective on a closed loop system is nothing more or less than a model of desired behavior for the system under control. That is, as described above, there are certain histories of the state of the system under control that are acceptable (i.e., satisfy the intent of the issuer), and the objective of the control system is to achieve one of them. An objective to point a camera, for example, can be achieved by any meander among the orientations that align the boresight within some tolerance of the target direction. It is far better to give a pointing control system such pointing objectives than to have to issue commands to thrusters, gimbals, and so on to make the same thing happen.

One sees then that closed loop control does not eliminate the need to know the behavior of a system. Rather, it simplifies the invocation of desired behavior by permitting direct commanding of it in the form of objectives — at least when objectives are transparent. In reality, the way objectives are commonly defined can leave certain aspects of behavior undefined, implicit, or conditional. Little omissions (as above where rotation around the boresight is ignored) may make someone unhappy, but far more serious infractions are possible. Failure to appreciate this can lead to dangerous vulnerabilities.

There are reciprocating aspects to this. First, an objective is not transparent if a control system produces behavior other than that expressed in its objective, while claiming otherwise. This much is clear. In addition though, when specifying an objective, anything short of expressing full intent is also not transparent. For example, if one intends a latch valve to be open for the duration of a propulsion activity, then a directive to open the valve is not the real objective. A valve control system could certainly act on such a directive, but not knowing the intent, it would have no subsequent basis for rejecting competing directives, determining whether the actual intent of the directive had been met, or taking any corrective action if it had not.

Real transparency therefore amounts to a sort of contract between the issuer of objectives and the control system achieving them, where objectives are expressed in a way that makes success or failure mutually obvious to both parties. Unfortunately, this is frequently overlooked. Delivered behavior can be different from what is expected, and the difference won't necessarily be apparent. Closed loop behaviors that are quirky, prone to surprises, hard to model, unreliable, or otherwise opaque about their capabilities are of questionable benefit. In fact, this is a primary concern behind reservations over increasing autonomy in space systems (including fault protection), which essentially amounts to permitting greater closed loop control on the vehicle. Operators are more comfortable dealing with the arcane details and peculiarities of a transparent open loop system, than with the broken promises of an opaque closed loop system.

Resolving this dilemma does not require abandoning direction by objectives. It simply requires more careful attention to making objectives say what they mean, and mean what they say. Indeed, substantial progress has been made in this regard, having been a principal motivation behind the gradual modularization and abstraction of capabilities within GN&C systems over the years, enabled by greater computational power. For example, many present day attitude control systems respond directly to profiled attitude objectives that bound motion with minimal transients. They require no compensation for biases or other errors — unbiased knowledge of attitude having been derived by applying knowledge of the sensor misalignments and biases. Similarly, influence on attitude can be exercised by setting secondary objectives on torque, which are met in thruster control systems by commanding thrusters on and off appropriately, adjusting for specific impulse and transients, mapping around failed thrusters, and so on.

Even where intent is clear though, systems still generally fall short in making overt connections between setting objectives and checking for their success. For example, pointing-profile objectives rarely include an overt criterion for how well this must be done, while fault monitors looking for excessive control errors rarely have any idea of their relevance to current objectives. Similar problems occur for tasks where the only indication that there might be an objective is the fault monitor (e.g., "too much" thruster firing). In still other cases, direction is given to systems with no explicit objective at all regarding the states under control, placing objectives instead on the state of the control system itself (e.g., "cruise mode").

In all such cases, responsibility for meeting objectives has been divided, and connections between objectives and their success criteria, where they exist, are generally implicit — hidden assumptions that are vulnerable to abuse. Intent is opaque, so predictable coordination issues arise. Consider what might happen, for instance, if transients at objective transitions became problematic. If gadgets, such as persistence filters, are added in the monitoring system to ride out such events, the system becomes vulnerable to real problems that occur during transitions and is overly desensitized to errors generally. On the other hand, if monitors are given access to control system internals (such as deadbands, modes, or gains) and other information they may need to be more discriminating, the result is a proliferation of ad hoc interfaces, an Achilles heel for any design. Thus, separating monitors from control cannot be defended on the grounds that it is simpler. Hidden assumptions, opaque intent, potential inconsistencies, fault coverage gaps, coordination problems, tangled interfaces, and so on are not simpler. They are merely symptoms of the fault protection problem.

There are two parts then to making objectives truly transparent: 1) be explicit about the existence and full meaning (success versus failure) of every objective, and 2) give full responsibility for managing objectives, including their failure, to the control system responsible for achieving them in the first place.

3.2.2 Transparency of Models

Models play an essential role in acquiring state knowledge by providing expectations against which evidence of state can be compared. Assuming a model is right, any departure from expectations is an indication that a knowledge correction of some sort is in order. Adjustments are typically small to accommodate measurement and disturbance noise, but may need to be large if the only way to align observations with a model is to hypothesize a discrete change of state, such as a fault. When models are viewed in this way, it becomes clear that there is no transparent way to separate models of normal and abnormal behavior, and hence to divide normal state determination from fault diagnosis.

The division of responsibilities described above frequently arises from the careless conflation of the two distinct sorts of error one sees in a control system. Large control errors indicate a problem in meeting objectives, whereas large expectation errors suggest a problem in state knowledge. The first calls for a fault response, while the latter calls for a correction in knowledge, such as the diagnosis of a fault. The former is *not* a fault diagnosis, and the latter is *not* an objective failure. However, when both types of excessive error are treated the same, they start looking different from everything else and will then tend to be split off from the rest of the implementation. Thus, the division of responsibility for objectives in control functions described above has an analog in the division of responsibility for knowledge in estimation functions. In both cases, models of behavior have been broken and scattered, resulting in a loss of model transparency.

Similar problems of transparency arise in the connections among controlled systems. In order for control systems to do their jobs well, it helps to close loops within subordinate functions as well, making them transparent in all the respects described here. However, since such dependent behaviors affect what control systems can do, models of these dependencies should be reflected in the behaviors control systems pledge in turn to others. Unless this is accomplished in an open and disciplined manner that allows integrity to be preserved and verified, subtle errors can creep into implementations that are hard to spot.

Consider, for example, the pressurization or venting of a propulsion system. Doing this properly depends on adequate settling of pressure transients, but the relationship between this objective and the valve actions taken to accomplish it is frequently implied only through command sequence timing. Even in systems with conditional timing, this dependency is often still implicit, the functional connection having been made elsewhere (e.g., in an uplink sequencing system), but beyond the awareness of the control system responsible for its accomplishment. Nothing in the normal way the propulsion activity is done carries any information about the effects of altered timing or actions in the sequence Therefore, any interruption that puts fault protection in control, perhaps for totally unrelated reasons, suddenly exposes the missing connections. What must a fault response do then to safely secure propulsion capability, once the interruption is handled? Resuming the sequence is problematic, since the assumed timing no longer applies; and restarting may not work, because initial conditions are different. Making the objective of the activity transparent would at least provide a basis for recognizing an unsafe condition, but gaining insight and concocting a reasonable response has to be done with no help from normal procedures. The result yet again is an ad hoc growth of scattered models, functions, and interfaces, which may not always get things right.

Adding to such problems is the typical absence of good architectural mechanisms to negotiate potentially competing objectives from multiple sources. Neglecting to articulate and negotiate potential conflicts, such as wanting to minimize thruster firing while achieving good antenna pointing, both involving objectives on attitude, will result in disappointing someone — usually the one without the explicitly stated objective. Looking around many implementations, it is common to see liberal manipulation of modes, enables and disables, priority or threshold tunings, locks or serialization of responses, and other mechanisms for accomplishing such coordination. However, these are all indirect, low-level, devices for managing software programs, not explicit, high-level architectural features for coordinating system objectives.

Another common mistake arising from neglect of this principle is the dispersal of essential behavioral information among complex parameter sets. No doubt buried there, for both designers and operators to sort out, are important relationships among objectives and behaviors. However, not only can their implications to actual behavior be opaque, but there is also typically no structure provided to ensure consistency of objectives with represented behaviors, consistency of the parameters among themselves, or consistency of the implied behaviors with actual behavior. Various mitigating processes can be tossed over this problem, but the haphazard treatment that created the problem in the first place leaves quite a mess under the carpet. The behaviors that emerge can be quite unpleasant.

Finally, there is the problem of shifting system capabilities. Such issues become particularly severe for fault tolerance. While most variation tolerance can be accomplished within a set range of objectives, and is thereby hidden from the rest of the system, fault tolerance (as with system management in general) frequently requires the rest of the system to adjust to changes in the available objectives. This may be temporary, until operation within normal operational bounds is restored, or it may be prolonged, if the system has irreversibly lost capability. Either way though, the system after a fault is not the one for which normal operation was prepared, and there are many more combinations of such abnormal cases than there are of normal cases. This means that for every potential alteration of available objectives in one place, there must be corresponding accommodations among all issuers of these objectives. These accommodations may result in further changes of behavior, and so on, rippling through the system until the effect is contained.

There are different ways to manage such propagating effects. A common one is to try to contain them with margin. If less is expected than a system is actually capable of delivering under better circumstances, then there is no need to announce reduced capability. In this approach, systems suffer reduced capability all the time in order to be able to tolerate it some of the time. This is common, especially for critical activities. Another approach is to pass along information about capability changes, not to the elements directly impacted by it, but to a central authority that reduces or withdraws objectives across the entire system in order to increase tolerance to the reduced capabilities. This is part of the ubiquitous “safing” approach. Building back to something useful — a task typically turned over to operators — may require design changes. A more forgiving and flexible approach is to establish a network of information exchange within the system that lets affected elements adjust as necessary, within a flexible range they have been designed in advance to accommodate. This becomes essential in systems operating in more dynamic contexts, but is also broadly useful in reducing operations complexity. In reality, all of these approaches appear in modern systems to some degree, each having merit depending on the nature of the risks involved.

Properly chosen and implemented, these methods work well. The systems in real danger are those where propagation effects have not been adequately considered or characterized, or where an assessment of this coverage is obfuscated by a design that fails to make them transparent. Unfortunately, as with negotiation mechanisms, most deployed systems show little overt architectural support for this principle. Instead, one typically finds minimal mechanistic support (e.g., command rejection, killing or rolling back sequences, and the like), accompanied by a plethora of ad hoc mitigation. Without the supporting structure of a principled architecture looking after this basic notion of closed loop control, the job gets a lot harder.

For transparency of models then, one must 1) expose and consolidate models, including the way they relate to the achievement of objectives, and 2) be explicit about how the rest of the system depends on, becomes aware of, and accommodates new information about what objectives are plausible.

3.2.3 Transparency of Knowledge

The dependence of control decisions on state knowledge suggests a logical division of closed loop control functions into two distinct parts, one being to maintain required state knowledge, and the other to make decisions about the appropriate control action. The acquisition of state knowledge is commonly referred to as *state determination* or *estimation*, but other activities, such as calibration and fault diagnosis, also fall under this definition. A benefit of this division is that system knowledge appears overtly at the interface between the two parts, making it easier to share across other control functions, easier to avoid divergence of opinion across the system, and easier to understand the reason behind control decisions. That is, state knowledge and its usage are made transparent.

It is essential to note that the state referred to here is not the state of the control system. Rather, it is the control system’s knowledge of the state of the system it controls. In fact, it can be argued persuasively that aside from this knowledge and its knowledge of its own objective, a control system should have no other state of its own. Practical issues work against that ideal, but it is something to keep in mind as a control system comes together. Every bit of state in a control system beyond the basic need for knowledge is a source of incidental complexity that must be managed carefully, and eliminated, if possible.

The provision of state knowledge is not nearly as elementary as typical implementations would lead one to believe. For instance, since state spans time, knowledge of it should also span time in a manner consistent with behavior models. However, in most implementations this is accomplished instead merely by interpolation or extrapolation, and then often implicitly and carelessly. Extrapolation, for instance, often amounts to simply using old data, assuming it will be used or replaced soon enough. Indeed, for most state information, and the evidence used to derive it, little if any thought is typically given to this.

In this light it is clear that events (measurements, detections…) are not state knowledge. They are merely evidence, which should contribute to state knowledge only through the application of behavior models. Even a simple measurement, invariably used only after some delay (small as that may be), requires a model-mediated extrapolation from event to knowledge in order to presume its usability over time. That supposition is a simple model of behavior, regarding both the changeability of the states being measured and latency in the control system’s use of the measurement. Such models are usually implicit in standard practice, but models they are nonetheless, since they describe how knowledge of state is to be carried forward in time. Therefore, they are in potential disagreement with other models, and could be flat out wrong, as recurrent problems with stale measurement data indicate. Avoiding this requires transparency.

Besides corruption from models that disagree, state knowledge may also appear in multiple versions within a system, and in fact often does (e.g., from different measurement sources). This is problematic because of the potential for control decisions working at cross-purposes due to different views of system state. There is more to this problem than just conflicting state knowledge. Separate opinions can never be as good as a shared opinion gained by considering all information sources together. Thus, there being no point in making matters worse than necessary, it is useful in any system to strive for a single source of truth for each item of state knowledge. That includes exercising the discipline not to copy and store state information around the system, but rather, to always return to its source on each use, if feasible.

Another key aspect of state knowledge is that it is always imperfect or incomplete, there being no direct, continuous way to access the actual state itself. Such knowledge can be determined only indirectly through limited models and the collection and interpretation of ambiguous or noisy evidence from the system. Thus, while a system is truly in a unique state at any instant, a control system’s knowledge of this state must necessarily allow for a range of possibilities, which are more or less credible depending on their ability to explain available evidence. Moreover, among these possibilities must generally be an allowance for behavior outside of understood possibilities. That is, a control system should be able to recognize when the system it controls is in a state for which confident expectations of behavior are apparently lacking. By representing such modeling issues in state knowledge, appropriately cautious control decisions are possible.

When confronted with multiple possibilities, control decisions become more complicated, but to edit the possibilities before presenting a selected one for control action amounts to making precisely this sort of decision anyway. This is one of many insidious ways that abound in systems to confuse the boundary between state determination and control, and hence the location of state knowledge (more on this in the next subsection). In doing so, transparency is compromised and the system is a step further on the slippery slope to lost architectural integrity. It is far better to acknowledge knowledge uncertainty, represent it honestly, and make control decisions accordingly with proper consideration of the risks involved.

Part of representing knowledge uncertainty honestly is also acknowledging that this uncertainty degrades further with time unless refreshed. This too should be overtly dealt with. For example, beyond some point in time, knowledge may need to be declared entirely invalid unless updated with new evidence. To keep the line between knowledge and control clean, this must be the responsibility of the producer of the data, not its consumer. Following this principle eliminates the problem of uninitialized or stale data.

Yet another concern is the choice of state representations, transparent choices being easiest to use correctly. With this in mind, certain obvious violations come to mind, such as the absence of explicitly defined units or frames of reference for physical values. However, given the topic here, it is useful to mention some of the recurring offenders in fault protection. We can begin with persistence counters, the most basic of which tally consecutive instances of an error. If such a counter is necessary, then presumably lone errors are insignificant — a statistical possibility under normal conditions perhaps. If that's the idea behind persistence, then it is apparently a measure of likelihood that the system is in an abnormal state. That's actually a good start, since uncertainty in state knowledge has entered the picture, as it ought. Moreover, before the persistence threshold is reached, one often finds systems rightfully discarding suspect data, indicating that the diagnosis of a potential fault has already been made, if not overtly. Yet, upon asking how a persistence threshold is selected, it quickly becomes clear in many systems that any of a variety of criteria might apply: allowance for an occasional operational fluke; the level of system error tolerance; a delay to control precedence or timing of fault responses; an empirical value that avoids false alarms — often multiple reasons — depending on the motive for its last adjustment. Thus, poor overloaded persistence actually represents almost anything but likelihood, as commonly used, and its threshold has as much to do with control decisions as with making a fault diagnosis. As a result, persistence thresholds join the ranks of a great many parameters in typical fault protection systems as tuning knobs, not directly relatable to states, models, or objectives, and consequently beyond help from any of these fundamentals.

Error monitors in general tend to have analogous problems when errors are not interpreted through models or correlated and reconciled with other evidence. When such a diagnostic layer is absent, and responses are triggered directly by monitor events, it becomes hard to put one's finger on what exactly a system believes it is responding to. This is an invitation for all kinds of interesting emergent behavior.

Similar issues occur when poor models of behavior are used. As described above, for example, timers in general have little if anything to say about the nature or conditionality of behavior. One particular troublemaker is the common command-loss timer. Do such timers represent objectives, state knowledge, measurements, models, or what? From the gyrations operators go through to manage them, and the inevitable mistakes made due to their typically opaque, raw character, it is clear that no one quite knows how such appliances fit into the control framework, or how they relate to other elements of the system.

There are many other problematic ways to represent state knowledge in a system. Others are described in the next subsection on control. A final one worth mentioning here though, is the disable flag for a fault monitor. If one disables a monitor, is this because the fault is no longer credible, or that the monitor is believed to be dangerously incorrect, or that its false trips have become a nuisance, or that the fault it detects is no longer considered a threat, or that triggering the fault under present circumstances would conflict with another activity, or what? The first few possibilities clearly refer to modeling issues; so it is prudent to have a way (arguably not this one) to temporarily preempt an incorrect model. The latter ones though amount to nothing less than direction for the control system to promulgate incorrect state knowledge. That's not transparency, it's cover-up — yet not unheard of in some fault protection systems.

For transparency of knowledge then, it is essential to 1) make knowledge explicit, 2) represent it clearly with honest representation of its timeliness and uncertainty, and 3) strive for a single source of truth.

3.2.4 Transparency of Control

In principle, given knowledge of controlled system behavior, knowledge of objectives (both imposed and imposable), and knowledge of the states of the controlled systems and those influencing it, no other information is required to choose control actions from among those that are plausible. There may be additional criteria for choosing options with the range of possibilities admitted by this principle (e.g., optimization), but in most cases that is a refinement somewhat beside the point here. Other exceptions involve cases where a system finds itself in a quandary among many poor choices, making it necessary to appeal to some meta-level policy that doesn't quite fit into the present schema. This gets a brief mention below, but otherwise the assertion above applies for most purposes.

Basing control decisions on information other than behavior, objectives, and states can actually be detrimental to a system. Presumably the only motive for using other data would be either that some other "real" objective besides the imposed one is known, or that state and behavior knowledge has not had the benefit of other available evidence or expertise. However, these conditions violate the principles of transparent objectives, models, and knowledge. Using alternative objectives subverts the very notion of control. Similarly, using extra data effectively puts the reconciliation of multiple information sources into the control function, creating a second, different version of system knowledge. As diversity of authority and opinion proliferates through such practice, the integrity of control decisions becomes increasingly suspect. Transparency of control consequently requires adherence to the principle that control decisions depend *only* on the canonical three items, for which some consensus has been established.

Unfortunately, violating this principle turns out to be an easy mistake, even with the most innocent of intentions. For example, having noted only that a previous error remains, a control function that tries an alternative action has violated the principle, effectively having made an independent determination of the state of the system. The proper approach would be for estimation processes to monitor control actions, making appropriate modifications to state knowledge to account for the observed results. Control actions would then be chosen differently on the next attempt, given the new state knowledge. For similar reasons raw measurements should be off-limits to control functions, certain arcane issues of stability aside. Control modes are also problematic, as described above under transparent objectives, and likewise for disable flags on fault responses, when misused as control mechanisms (like disabling fault protection during critical activities), and not just stoppers on modeling or implementation glitches. There are many other examples.

The flip side of using extraneous data is to ignore the state data you have. All kinds of commonly used mechanisms share this problem, such as time-based sequences that make no appeal to state knowledge, relying instead on fault monitors to catch dangerous missteps. Also problematic are responses with no basis at all, other than poor behavior. For instance, if errors are large but nothing else seems broken, resetting the offending algorithm makes things different for sure, and with luck might actually accomplish something. However, there is no basis for such a response from the point of view of control principles. (Reset and similar actions at a meta-level of control are different, because they involve the health of the control system itself.) A more likely candidate for puzzling problems of this sort is a modeling error, at which point the system should either shift to a more defensive posture or try to correct the model. Admittedly, the latter is a tall order, out of scope for many systems, but the first line of defense at that point, having validated models, has already been breached. To avoid getting into this situation one must honor the principle of control transparency. Non-specific responses signal an opaque design that demands scrutiny.

A useful byproduct of control transparency is that much of the complexity one often sees in control, when viewed properly, turns out to be a state or behavior knowledge problem. That’s no consolation to the providers of this knowledge, though it does mean that knowledge is likely to become easier to understand as transparency is improved. Other benefits appear on the control side, too, such as a more straightforward decomposition of activities, and consequent simplifications in planning.

Control transparency, is consequently fairly easy to achieve, simply by 1) avoiding any basis for decision besides knowledge of objective, behavior, and state.

4 Further Implications to Fault Protection

Fault protection is clearly a control function, overseeing the state of a system, and trying to achieve objectives of safety and success. However, as control systems, many fault protection implementations impose a decidedly non-conforming model of control over the systems in their charge. Having submerged basic principles of control, it is of little surprise that fault protection remains problematic in many systems. In exploring the basic notions of system control, it is alleged here that a number of these fault protection issues result from failure to apply control principles transparently, especially in comparison to normal GN&C design. In keeping with that theme, the following additional guidelines are suggested as ways to bring this discipline, broadly appreciated within the GN&C community, into GN&C fault protection.

Do not separate fault protection from normal operation of the same functions. Because faulty behavior is just a subset of overall behavior, normal and failed operation are intertwined in many ways: measurements of normal states are also affected by health states; robustness of control functions beyond normal tolerances aids fault tolerance; incremental tactics of normal algorithms play a central role in bootstrapping capability after a fault, etc. Thus, diagnosing the failure of a device is just part of estimating the overall state of the device; retrying a command is just part of controlling the device, given its state and the desired objective, and so on. The main difference seems only to be that decisions along paths of abnormal behavior are more sweeping and, if we are fortunate, invoked less often than those along paths of normal behavior.

One often hears explanations of fault protection complexity as being related to its interactions with so many other spacecraft functions. This way of looking at the issue is problematic, since it views normal operation and fault protection as separate functions. In fact, fault protection is often perceived as above normal operation — a sort of supervisory level function that watches over and asserts control when normal functions misbehave. In most cases, this is inappropriate, artificially and unnecessarily creating broad functional relationships over and above what are already there by virtue of the physical relationships within the system under control. By excluding fault protection from the control function for each system element, these control functions are unable to fully assume their cognizance role. This unnecessarily divides functionality, adding interfaces and complicating or confounding interactions all around the system.

Another way to appreciate the value of uniting management of normal and abnormal behaviors is to consider the ever-diminishing distinction between them in more complex systems. This is well illustrated in reactive, in situ systems, such as rovers, where novel situations are routinely encountered and revision of plans is frequently necessary, as initial attempts are thwarted. In the pursuit of viable alternatives, dynamic considerations of risk, priority, resource use, and potential for lost options are shared under both normal and abnormal conditions. Since putting two parallel management systems in place for such systems would make little sense, one sees instead the inevitable joining of fault protection with normal operation. Other systems are also gradually taking this route[ii], as the merits of merger become apparent.

Strive for function preservation, not just fault protection. In keeping with the recommendation to keep fault protection and normal control together in pursuit of their shared intent of meeting objectives, the aim should not be merely to react mechanically to some a priori list of failure modes with rote responses, but instead to acknowledge any threat to objectives, striving to preserve functionality no matter what the cause.

This insight creates a different mind-set about fault protection. It helps one see, for instance, that the purpose of fault trees, failure modes and effects analyses, hazards analyses, risk assessments, and the like is not for the assembly of a list of monitors needing responses. That approach unveils only a limited model of certain aspects of certain chains of influence in certain circumstances. Rather these must be thought of as just part of an encompassing effort to fully model all behavior, both normal and faulty, that will help clarify what is happening and what is at stake. That is, it is equally important to understand all the things that must work right for the system to succeed, recognizing and responding to their loss, even if the manner of this loss is unforeseen. This is what a true control system does, when fully cognizant of the system it controls.

This mode of thought should also disabuse engineers of notions that fault protection has no role after it responds to the "first"[5] fault, or that fault protection V&V is finished when everything on the monitor and response lists has been checked off, or that operator errors or environmentally induced errors count as "first" faults that absolve fault protection of further action, or that latent faults don't count at all. When the perceived role of fault protection is to preserve functionality, such incidentals are irrelevant. Any project choosing to depart from this principle had better spell it out plainly in its fault tolerance policy, so everyone understands the risk. This is a common point of miscommunication.

Test systems, not fault protection; test behavior, not reflexes. If fault protection is an integral part of a system, it should be tested that way. Unfortunately, this often doesn't happen the way it should — a natural consequence of viewing fault protection as a distinct supervisory function. Systems tend to be integrated and tested from the bottom up, so in the conventional view of things, fault protection gets integrated and tested last, and specifically *after* normal capability has been integrated and tested. Even when fault protection functions are present earlier, it is common to do much testing with fault protection disabled, or if enabled, to ignore indicators of its performance (e.g., detection margins), because tests are focused on normal functions. The net result is that nothing is really tested right. The magnitude of this blow to system integrity is brought into focus when one views the role of fault protection as function preservation. In this role, objectives are not just commands to be executed; they are conditions to be monitored for achievement. Similarly, diagnosis models don't just signal faults; they predict normal behavior too. Consequently, in this integral role, fault protection fulfills the essential role of guarding expectations for the system's performance. One of the greatest benefits of fault protection during testing, therefore, is to help watch for aberrations across the system, even if a test is focused on normal functionality in a particular area.

This control system view of fault protection has the additional advantage of elevating attention, beyond rote reflexive responses, to behavioral characteristics that provide a basis for confidence in the integrity of the underlying design. This is similar to the way normal GN&C control functions are analyzed and verified. The idea, for instance, is not to test that the right thruster fires when a control error threshold is exceeded, but rather to test that pointing objectives are

[5] This could be "second", if there's a two fault tolerance policy in effect, and so on.

consistently met in a reasonable manner. Applying the same idea to fault protection shifts the emphasis from futile attempts at exhaustive testing toward a more relevant exploration of emergent behavior and robustness. That in turn motivates proper mechanisms for well-behaved transparent control from the start, closing the circle back to sound principled system architecture.

Review all the data. Of course, the presence of fault protection does not relieve testers of responsibility to review data, but here again an altered point of view about the role of fault protection is helpful. If fault monitoring is focused solely on triggering fault responses, a limited kind of diagnostic capability results. However, if monitoring is focused on general, model-based expectations, making note of even small departures, while leaving control functions to decide whether departures merit action, then a much more transparent system results, and the testers' task gets easier. If that seems a substantial increase in scope, keep in mind that not all such capabilities must reside on the flight system. Fault protection should be considered just as integral a part of ground functionality as any other. The same misperceptions and opaque implementations divide ground systems too though, with the same consequent loss of insightfulness.

Cleanly establish a delineation of mainline control functions from transcendent issues. Despite moving most fault protection out of a separate supervisory role, there remains a need for supervisory level fault protection in systems. These functions should in general be limited though to managing the platforms on which other control functions run, and to exercising certain meta-level policies, as alluded to earlier, that are beyond resolution by usual means. Examples of the former include overseeing installation of new software, monitoring software execution, switching from a failed computer to a backup, and so on. An example of the latter is the regularly occurring catch-22 during critical activities (e.g., orbit insertion), where a choice must be made between preserving evidence of a mishap, or daring a final attempt at success that is almost certain to destroy the system along with its evidence. No objective seeks the latter outcome, yet this is the path most projects take. Aside from the meta-level issues involved with such functions, another good reason for their delineation is to ensure they get appropriate visibility, given their far-reaching implications. (Limitations of space here require neglecting certain thorny implications to fault protection from meta-level control, involving preservation of credible state data, tolerance to control outages, etc.)

Solve problems locally, if possible; explicitly manage broader impacts, if not. Localization is actually a natural byproduct of keeping normal and faulty behavior management together. The advantage of localization is clearly to contain the complexity of the behavior associated with faults. In the best case (e.g., when functionality can be quickly restored, or an equivalent backup is available) this can be minimally disruptive to the rest of the system. The one item that cannot be localized, however, is the resulting change of behavior of the element. Any reduction in the range of objectives the element is able to accept, even if it is temporary, must be accommodated by other elements dependent on this capability. These effects may ripple broadly across the system and over time, disrupting plans. Thus, the new capability landscape after a fault results in new system level behavior. This is one of the emergent implications of failure, which must be handled through transparent management of model dependencies, as described above.

Respond to the situation as it is, not as it is hoped to be. There are a number of aspects to this, but the bottom line is that action calls for continual reassessment and revision. State and objectives are both subject to change at any time, so in keeping with the principle of responding only to these items, any action that ignores their change is wrong. A common violation of this is to embark on a course of action that requires blocking out other responses or reassessments of state until the action is over. This presumes that the initial decision will remain the correct one, with a predictable outcome, no matter what develops in the meantime.

Distinguish fault diagnosis from fault response initiation. Detection and diagnosis are state determination functions, not separable from determining normal functionality, so they should have nothing to do with deciding, or even triggering, control actions. Diagnosis of a failure is merely the impartial assertion of new knowledge regarding some failure to satisfy design functionality expectations. It is unconcerned with implications to the system. Deciding whether or not a fault diagnosis merits a response is a control action, and control actions are driven by objectives. Therefore, the only role of fault diagnosis is to supply the information used in a control function's determination that an objective may be threatened or have failed. In the absence of an objective, or in the presence of a fault that does not threaten objectives, no response is warranted. Likewise, if an objective is threatened, response is warranted, fault diagnosis or not.

The reverse of this is also important. In every decision is the risk of being wrong, so a decision to act on one of multiple possibilities is not the same as a determination that that possibility is in fact true. Knowledge remains as uncertain right after the decision as it was before. Later, having embarked on a particular path, success or failure in making headway against a problem can be taken as evidence that the conjecture behind the choice was correct or incorrect, respectively. The way to gain this knowledge, however, is not to simply assert that conclusion in the fault response (as many fault protection systems do), thereby mixing estimation with control and dividing opinion, but rather for estimation functions to observe all actions taken, as they would any other evidence, and revise state knowledge accordingly. Subsequent control actions can then be adjusted according to the updated knowledge, as in principle they should.

Use control actions to narrow uncertainty, if possible. A familiar situation is to encounter an error that clearly indicates something is wrong, but lack the ability to discriminate among various possibilities. Similarly, there are often situations where the difference in behavior between normal and faulty operation is hard to see. There are two general strategies that can be followed in such cases to resolve the uncertainty.

The first is to remove sources of ambiguity, given that the problem may be far more apparent under other, typically simpler conditions. In GN&C systems, this tends to appear in the form of regression to simpler modes of control involving fewer parts, and incrementally rebuilding to full capability. The fewer the parts involved, the fewer the potential culprits, should the symptoms remain; and should symptoms disappear in a simpler mode, options narrow through a process of elimination. By moving to the simplest modes first and rebuilding, ambiguity and vulnerability to symptoms are both minimized; and of course, there is always the option of stopping part way and waiting for ground help, once minimal capability for safing is established. Fault protection systems usually accomplish such things by rote, through safing and normal reacquisition steps, but in keeping with the principle of transparency, it is better to connect the acts of simplifying and rebuilding with the presence of uncertainty and with threats to objectives.

Sometimes the reverse strategy is preferable, where it is advisable to bring in additional evidence to corroborate one alternative versus another, rather than making a series of arbitrary choices, expecting that the situation will resolve itself eventually. In general, better knowledge leads to better decisions, which is ultimately the safest route to follow, when the option is available. The common thread between these two approaches is simplification. Make diagnosis easier by removing complications and adding information.

Make objectives explicit for everything. Nothing a control system is responsible for should be in response to an unstated objective. Moreover, whether or not such objectives are being met should be explicitly apparent, as with any other objective. In this way, enforcing operational constraints and flight rules, avoiding hazards, managing resources, and so on *all* become a normalized part of the control functionality of the system, as opposed to additional overriding layers of control or awkwardly integrated side-functions.

Make sure objectives express your full intent. Fault protection is less likely to respond to problems correctly, if the objectives provided are not what are actually required. A good example of this is the typical specification of an orbit insertion objective in terms of a delta-velocity vector. If the insertion burn could be guaranteed to occur at the right time, this would be okay, but critical objectives also need to be satisfied when things go wrong. A better objective, therefore, would be delta–energy, which is significantly less sensitive to timing, and is nearly as easy to accomplish if velocity information is already available. This not only permits greater freedom in choosing how to respond, but the resulting response can have a smaller penalty to the system. One can apply this notion in many places with similar benefit, and the only necessary enabler is to make objectives more transparent, as basic principles would suggest anyway.

Transparency of objectives can be accomplished in GN&C in many other ways, as well, with consequent benefit to fault protection. For example, in a well-designed, feed-forward, profiled motion system, the motion expected is just that specified in the objective. Feedback need only accommodate disturbances, so any significant control error becomes a clear sign of trouble, indicating departure of behavior from both the objective and the underlying model. Other systems that rely instead on a controller’s transient characteristics to profile motion are substantially harder to assess, due to this loss of transparency. Similar observations apply regarding state determination systems, where substantial transparency gains can be made through more direct application of models to define expectations.

Reduce sensitivity to modeling errors. As discussed, closed loop control exists in large measure to diminish the effects of variability in a system. Therefore, one of the side effects of poor transparency in a control system is that the effects of variability are poorly contained. This can happen if a design is too dependent on the detailed correctness of the models it uses. Given a choice, robustness always favors less sensitive designs. To decide whether this criterion is met, it pays to assess sensitivity in all its dimensions and pay close attention to those aspects that dominate. Alternatives might be found that reduce sensitivity, if you’re looking for them; and if not, it will become obvious where extra-healthy margins are a good idea.

Alternatively, a more proactive approach may be worthwhile. The profiled motion system mentioned above, for example, will begin to show significant transients, even with the feed-forward actions in place, if the feed-forward model is incorrect. Therefore, a control system can substantially improve its transparency by attempting to estimate the parameters of this model (essentially as additional system state) and incorporate the results into its feed-forward actions. This not only improves its ability to recognize faults, but also promises to maintain operation within a more linear realm, where the system is better behaved and easier to characterize. Moreover, the additional insight gained by the system via the additional estimated states can be used as a forewarning of future problems or an indicator of reduced margins that alone may require action. It is always better to respond to a problem early than to wait idly for a crisis.

Follow the path of least regret. Transparent models require honesty about likelihoods. Unlike most of the random things in normal GN&C, which are expected to occur and usually have well characterized statistics, few faults are viewed in advance as likely, and their statistics are rarely known, even crudely. Moreover, most faults that happen are either unanticipated or transpire in surprising ways. Therefore, any presumption about which faults are more likely than others must be considered flawed. In general, fault protection decisions ought to be based on criteria other than fault likelihood. What should be of more concern is the list of things one thinks the system might be able to recover from, if given a fighting chance to do so.

Sometimes, even when the odds seem dramatically in favor of one possibility, one needs to create one's own luck. For example, what is extremely likely to be just a sensor false alarm might in fact be a genuine fault in the sensed system that could turn out to be catastrophic. Getting a second chance in this case may require a suboptimal first choice with all its nuisance value. Given such possibilities, one might suppose it's always best to assume the worst and sort out the details later, but this course can also lead to trouble, when actions taken by fault protection are not benign or reversible (such as wetting a backup propulsion branch). False alarms commonly present such dilemmas (not to mention desensitizing operators). With enough foresight (and budget) a system can often be instrumented well enough to eliminate such ambiguities, but in the end there will still be a few that just need to be thought through with great care. The art in fault protection is not always to be right, but rather to be wrong as painlessly as possible.

Take the analysis of all contingencies to their logical conclusion. The problems of viewing fault protection as distinct from other control functions have already been cited. However, there are also problems with viewing fault protection as confined to flight system health. As with most other systems, fault protection is not closed, so it should never be treated this way, even though much of what it does must be autonomous. The true scope of fault protection extends to the entire operations system and supporting ground elements, and possibly even to other missions. Moreover, what may be a short-term victory could in the long run threaten the mission or deprive operators of all hope of ever finding out what went wrong. Therefore, it is important to follow all branches in the contingency space and to carry them to their absolute logical conclusion in order to see whether the fully integrated design (and plans for it) supports them.

Successful completion of a fault response is consequently not a logical conclusion. Spacecraft have been lost by neglecting a larger issue, even though the flight system managed to enter a safe mode that could be maintained indefinitely. To be truly finished, one must carry responses all the way to final resolution, including all the side effects, trap states, and so on. One must consider mission operations' role and preparedness in monitoring the system and responding to problems, restoration of routine operation (including all necessary replanning), the long-term viability of the system and mission in the recovered state (if restoration takes much time), threats to consumable resources, to upcoming critical activities, and to planetary protection, and adequacy and availability of data to support both a timely operations response and a complete post facto assessment. Such concerns are all related to control, so just because they involve processes somewhere besides the onboard system doesn't mean there are different rules for them. The larger system should strive for the same level of transparency as any other control system.

Never underestimate the value of operational flexibility. Another particular aspect of driving contingencies to their logical conclusion also deserves mention. This is the vital importance of operational flexibility. Many systems have been recovered from the brink through astute actions and clever workarounds. Therefore, any design feature that forecloses options also removes one of the most valuable assets a spacecraft can possess. If the logical conclusion of a contingency path is the inability to act because alternate spacecraft capability cannot be applied, it is time to revisit the design. Unfortunately, such exploration of contingencies often occurs late in the development cycle, rather than early when the design is malleable. (This is partly due to the misconception that reprogrammable software is all that's needed.) Furthermore, late design changes for other reasons tend to be made without fully revisiting contingency capabilities. The fact that so many systems have been recovered nonetheless reflects a deep-seated appreciation among engineers for the importance of this principle.

There are a variety of ways to do this. These include making it possible to safely operate prime and backup equipment at the same time; creating finer grain fault containment regions; avoiding rigid modal designs and canned procedures in favor of flexibly exercisable modular capabilities; making parameter changes easy and safe, even under spotty telecommunication capability; being able to confidently direct the system at every level in the design; having very tolerant safing systems; and so on.

Allow for all reasonable possibilities — even the implausible ones. The logical conclusion of some conditions may very well appear hopeless, but it is good practice to keep trying, while certain basic system objectives such as safety and communication remain unsatisfied. Having exhausted all likely suspects without resolving a problem, it is important to try the unlikely ones, as well, even if they have been deemed “not credible”. As stressed above, fault responses are not exhausted just because all the identified failure modes have been addressed. The objective is to preserve functionality, even when the unexpected happens.

This concern applies not just to what faults might occur, but also to what conditions might ensue, from which recovery will be necessary. For dynamic states, this could include high or reversed rates, flat spin, incomplete or unlatched deployments, excessively energetic deployments, toppled passive attitudes (e.g., in gravity gradients), incorrect or unexpected products of inertia, excessive (even destabilizing) flexibility or slosh, inadequate damping, thermal flutter, unexpectedly large variations (e.g., in misalignment, external disturbances, torque or force imbalance, or mass center offset), and so on. Other kinds of states and models are also vulnerable, whether they be device modes, sensor obscuration or interference, or whatever. Even common-mode problems should be considered.

Pursuing this idea further, one must also conclude that what is usually considered essential certain knowledge of the system may need to be abandoned. For example, setting a sun sensor detection threshold is a straightforward activity, performed automatically in some interplanetary systems as a function of distance from the Sun, derived from trajectory data; but suppose the Sun cannot be detected, despite every appeal to redundancy, search, and so on. At some point, one might very well suspect that information regarding the distance to the Sun is wrong, even though there would be little reason to doubt it. As a last resort then, it would be advisable to discard this state knowledge anyway, and try setting the threshold by trial and error. This is another application of the admonition to simplify as far as possible.

Note that the sorts of things listed above would largely fall under the category of design or operator errors, and many would argue, given the daunting scope that is suggested, that fault protection needn't protect against such things, since there are processes in place to avoid them. Unfortunately, history cautions against such a point of view. If conditions such as these are physically plausible, some gremlin is likely to solve the puzzle for entering them, even when system designers can't. The rather sobering reality, in fact, is that more space systems are lost to such errors, than to random faults per se.[iii] Therefore, to presume that the design precludes them is not particularly insightful. It takes extraordinary effort to be absolutely sure (and right) about such things, which raises its own issues of scope and suggests merit for a second thought.

So why attempt to cover such possibilities? It's hard enough getting a GN&C design to work under normal conditions; so throwing in all these abnormal possibilities in all their uncharacterized variability may seem like an impractical and unaffordable stretch. A little effort will sometimes go a long way though, and may make the difference between any sort of survival and no survival at all. For example, stabilizing rotation rates is simply a matter of damping rotational energy, which can generally be done quite simply with the right system configuration. Most systems already have this capability in order to handle initial launch vehicle tip-off, so applying it to fault recovery may be an easy extension, if anticipated early enough. Likewise, even crude control may be enough to ensure a power-positive, thermally safe attitude, and with slow enough responses could be made stable under most conditions. Safing capabilities are always planned anyway and can be tuned for minimal requirements, if permitted by the design, so it's worth a trade study to decide how far into the space of abnormality it may be worthwhile to venture. Safing should provide fallbacks that are as absolutely rock bottom foolproof as possible. Then operational flexibility might come to the rescue, once operators can get involved. So don't surrender; simplify, and try to get by.

Design spacecraft to be safe in safing. Don't wait to figure this out later. Many of the difficulties in supplying a truly robust safing capability, as described above, go away if considerations for safing are part of the original design concept. Consider, for instance, the passively stable orientation and rotation of a spacecraft. There's a fair chance the spacecraft will find its way into this state at some point and stay there for a while, whether one wants it to or not. If safing does not use such as trap state, then safing will always have this threat against it. So much the better then, if there is only one such trap and it happens to be one that can be survived indefinitely with reasonable support for telecommunications. It's always better not to have to fight upstream, so design the system so it helps you. The same argument can be applied to other critical states. Making these things happen is a direct consequence of putting fault protection and normal operation together from the outset. In the same way that no GN&C designer would ever allow a spacecraft configuration that was not a cooperative partner in control, fault protection designers must take the same ownership of the system behavior they are expected to manage, both normal and abnormal.

Include failsafe hardware features. In the same vein, various failsafe hardware features can also help. For example, it should never be possible to request a torque, force, or rate indefinitely; such commands should always time out quickly unless reinforced. There should never be illegal combinations of bits that can cause action. One should never need to know a priori what the state of a device is in order to be able to command it to a safe state. There should be a watchdog for command source liveness. Resets and watchdog triggers should always establish a safe hardware state. And so on. Such features dramatically reduce opportunities for problems to proliferate for lack of attention, while other serious matters are disrupting operation.

Check out safing systems early in flight. As a final check, it is always prudent to never leave a spacecraft unattended after launch until safe mode has been demonstrated. And of course, in case something does go wrong, the contingency capability to deal with it must be in place. This includes operators on duty, adequate telemetry and ability to uplink (even under adverse conditions), good margins, readily adjustable behavior (e.g., parameters, mode and device selections, and so on), good testbeds where problems can be readily duplicated, and access to the right experts when you need them. As described above, supporting capabilities for such contingencies should be added early in development.

Carefully validate all models. Modeling is a double-edged sword. Good models are clearly necessary for proper diagnosis and response, but having the ability to work around at least some imperfection in models has also been emphasized, since knowledge of behavior is never perfect. Some may argue, in fact, that model-based designs (especially model-based fault protection) are vulnerable precisely for this reason, though this begs the question of how any design at all is possible without expectations of behavior. The question then, whether models appear explicitly in an implementation or only in the most indirect way through designs, is how one knows they are valid (i.e., good enough to use).

Model validation must certainly appeal to expertise. Those most familiar with a modeled item should develop or at least review its model, with full awareness of how the model will be used and the need to keep it current. Consequently, communicating models effectively and consistently is essential. Model validations aren’t closed any more than systems are.

Ultimately though, model validation is best when supported by data, and this is where it is critical to understand the fundamental irony of model validation: *data can only invalidate models*. Model validation is consequently an exercise in looking for contradictory data. One does this by considering a broad set of conditions to test the model’s predictive powers, and also by considering data sets capable of discriminating among competing models. Models that are mere data fits, or that only loosely appeal to physical phenomena, have very little explicatory power and are prone to mislead or mask issues. How one produces such data to validate failure modes (including operation outside of operating range) naturally raises many questions. Where feasible, it is always best to get the data anyway, even if difficult, and where not feasible, one should double up on experts. Space systems tend to be merciless, when you get it wrong. One thing is certain though. Testing a spacecraft with all fault protection always enabled is the best way to discover whether or not your models *at least* get normal behavior right.

How good a model must be is more subjective. One technique for deciding is to exaggerate the effects of everything that comes to mind in order to see if anything could ever be a problem, eliminating only those effects with provably implausible significance. What one can always count on though is that assumptions regarding a “credible” range of operation are always suspect, especially where faults are involved.

Three more cautions are in order, the first being to avoid the blunder of validating a model against itself. No one would foolishly do this in any blatant way, of course, but it happens from time to time, nonetheless, when a testbed includes in its simulation a model from the same source or assumptions as those behind the flight design. When errors or inappropriate simplifications are repeated between the two, the system will obligingly work fine — until it flies. The second caution is related. Suspect validation by similarity or extrapolation of prior experience. Even modest departures can be fatal if handled incorrectly. Besides, your source might have been wrong. Finally, in a similar vein, piecemeal testing is less satisfactory than end-to-end testing, if you have to make a choice, and both are required in order to find compensating errors.

Given a choice, design systems that are more easily modeled. This advice is easily appreciated, but not always the most obvious to apply. Every engineer has seen a seemingly simple system concept devolve into endless analysis and testing, as the enormity of the operating space that it allows, and therefore within which it may need to operate, is revealed. There is a tendency in this age of ultra-sophisticated modeling tools and supercomputer simulations to soldier on in confidence of the brute strength offered by these techniques. Nothing, however, will ever substitute for an elegant form that is simpler to analyze. Therefore, one must be prepared to face the reality that often the best concept is the one that can be modeled (and have the models validated) more easily, even if it requires a few more parts and a bit more control.

Elegance of form is useful in control system architectures as well. All of the principles of transparency discussed here have the ultimate goal of making control systems more easily modeled, which makes systems easier to understand, and hence less complex. This can result in great simplifications elsewhere, especially in fault protection, a worthy note on which to end this discussion.

There is much that could be added to this exposition, which has been a necessarily simplistic race through issues that are deep and complex. Essential topics have been neglected. The objective, however, is not to provide a comprehensive guide or rigorous theoretical treatment, but rather to demonstrate that there is value in adhering to ideas from the roots of our discipline, even in today’s complex world.

5 CONCLUSION

Managing the complexity of a large system requires confident control, and one of the most difficult aspects of that control is fault tolerance. The premise here has been that fault protection, as fundamentally a control issue, will be well served through a more careful grounding in the same patterns of thought and principles of design that have guided successful GN&C systems through decades of development. However, this modest exploration of connections is an attempt, not just to suggest some merit to that idea, but also to argue that this is not uniquely a fault protection concern. That is, it appears that fault protection complexity must be addressed, not merely as an integrated control element of a system, but as an *integral* part of a *unified* approach to system control.

If this is correct, then it follows that solving the fault protection problem requires nothing short of solving the complexity problem overall. That insight is important, because we are moving into a new era of unprecedented complexity, where present issues of fault protection provide only a sampling of what is to come for systems at large. One sees this, for instance, in space systems moving into more uncertain environments, where the differences diminish greatly between responding to random faults and responding to random environmental influences that obstruct progress. What was once required only of fault protection is slowly but surely becoming the norm for routine operation in such systems, especially as demands for responsiveness and efficiency escalate. This is just one of many such challenges we can anticipate. However, if past experience with fault protection is any guide, difficulty in satisfying this evolution of needs will hold the future out of reach, for we have yet to truly come to grips with the complexity issue.

The solution to any complexity problem lies in understanding, but understanding requires a transparent framework for ideas. Returning to the roots of systems theory and principles of control is offered here as a significant part of this framework. By looking at GN&C fault protection fundamentals in this light, it is hoped that this review of principles will be useful both on its own merit, and as a guide to future developments in both fault protection and system management. Techniques of most relevance to this approach, where the fundamental notions of state, models, and objectives appear in their most transparent form, are the state- and model-based, goal-driven approaches to systems engineering and control.[iv,v,vi]

The key to applying these ideas, however, is not just to embrace another set of rules. No concept maintains its integrity during development without the structure of formal architecture and rigorous systems engineering methodology. Providing this is not the easiest thing we can do, but simplistic notions do not confront complexity; they merely shift it elsewhere, as much of our experience with fault protection demonstrates. The approach recommended here is to weave ideas of transparent control deeply into our architectures, so we can take space systems to the next level.

6 References

1. D. Dvorak, M. Ingham, J.R. Morris, J. Gersh, "Goal-Based Operations: An Overview," *Proceedings of AIAA Infotech@Aerospace Conference*, Rohnert Park, California, May 2007.
2. E. Seale, "SPIDER: A Simple Emergent System Architecture For Autonomous Spacecraft Fault Protection," *Proceedings of AIAA Space 2001 Conference and Exposition*, Albuquerque, NM, Aug 2001.
3. J. Newman, "Failure-Space, A Systems Engineering Look at 50 Space System Failures," *Acta Astronautica*, 48, 517-527, 2001.
4. L. Fesq, M. Ingham, M. Pekala, J. Eepoel, D. Watson, B. Williams, "Model-based Autonomy for the Next Generation of Robotic Spacecraft," *Proceedings of 53rd International Astronautical Congress of the International Astronautical Federation*, Oct 2002.
5. M. Bennett, R. Knight, R. Rasmussen, M. Ingham, "State-Based Models for Planning and Execution," *Proceedings of 15th International Conference on Planning and Scheduling,* Monterey, CA, Jun 2005.
6. R. Rasmussen, M. Ingham, D. Dvorak, "Achieving Control and Interoperability Through Unified Model-Based Engineering and Software Engineering," *Proceedings of AIAA Infotech@Aerospace Conference*, Arlington, VA, Sep 2005.

Final Report

NASA Study on Flight Software Complexity

Appendix H — Thinking Outside the Box to Reduce Complexity in NASA Flight Software

Robert D. Rasmussen,
Jet Propulsion Laboratory,
California Institute of Technology

Contents

1 Introduction

A study of growth in flight software complexity has been commissioned by NASA’s Office of the Chief Engineer (OCE). **Special Interest 2** in the proposal for this study is defined in terms of the following questions:

- How can unnecessary growth in complexity be curtailed?
- How can necessary growth in complexity be better engineered and managed?
- Are there effective strategies to make smart trades in selecting & rejecting requirements for inclusion in flight software to keep systems lean?
- When growth in complexity and software size are necessary, what are the successful strategies for effectively dealing with it?

Among several approaches listed in the proposal for addressing the questions of Special Interest 2 was **Approach 4**, stated as follows:

> “We propose developing a position paper that provides an "out-of-the-box" or controversial approach to complexity management, then seeking feedback on it from a selected group of reviewers in NASA, and selected members of industry and academia.”

Such an approach is the topic of this position paper.

1.1 The Box

In arguing for the merits of the approach described here, an attempt will be made to show that this approach is **outside the box** only in the original sense of that phrase: **as a solution that is obvious, once described**, demonstrating the avoidable constraints of preconceptions. In this sense, the approach described here is ironic, in that it attains its controversial stature without positing novel or unfamiliar ideas. Instead, it merely requires the more assiduous application of a few well-tested principles — principles that are not strangers to many NASA engineers, and indeed, principles that are already in broad use on NASA projects.

One might suppose, therefore, that there should be no controversy in this approach; or more to the point, that the principles it embodies should already have obviated the complexity issue, if they truly have the potency supposed here. To be most effective, however, principles must be consciously embraced and employed with unwavering conviction, a fundamental premise mocked famously by Groucho Marx, who once said,

> “Those are my principles, and if you don't like them . . . well, I have others.”

We certainly like to think of NASA as a highly principled culture. That we find ourselves adrift from these principles is therefore an added irony, presuming one accepts their value.

This seems to be due largely to inertia in our risk-averse culture, which has nurtured old expedients in the name of conservatism. For example, we find little abstraction at the system level to deal with common system level issues, even though system level interactions are one of the knottiest elements of the complexity problem. Instead, we see only modest refinements of ideas from the early years of spacecraft software development, when that was all that would fit into the tiny computing capacity of the day. Trying something markedly different is no longer out of reach. Yet we hold tenaciously to old ideas that have long since proven their limitations, and we do this mainly because they are the devils we know and have mastered. Thus, what was once a tentative response to severe computing limitations and nascent capabilities in embedded software has resolved itself over time into dis-innovative skepticism of the very methods required to address growing software complexity.

This has not halted progress, by any means, but what it *has* done, as evolution has retreated into obscure niches, is to fragment advances into localized, parochial realms. Consequently, the principles that maintain *system* integrity have been submerged in an untidy sea of subsystem details.

This largely explains the complexity problem. What remains at the system level tends to be raw procedural[1] and connectional[2] mechanisms, intrinsically lacking conceptual depth or coherence, and therefore ready for abuse. Lacking any other unifying structure, a tangled assortment of parameters, commonly numbering in the thousands, is typically needed to tune systems to proper behavior. Such features not only hobble the ability of flight software to address complexity. They also divert the attention of systems engineers and mission operators into a multitude of obscure details — manipulating these mechanisms to gain the desired effects.

The approach outlined here is an attempt to reestablish system order by identifying and codifying the elements of coordinated system operation, according to basic systems theoretic principles of control. The aim is not to squeeze systems back into some previous era of presumed simplicity. Rather, on the assumption that complexity is inevitable, the objective is to provide a principled framework within which this complexity can be managed.

1.2 The Complexity Issue

Uncontrolled complexity in flight software is manifest in many ways, the most obvious being that software is often delivered late or incomplete. With these shortfalls come higher cost, less rigorous testing, poorer operability, reduced performance, inhibited usage, lower reliability, and occasionally lost opportunities or lost missions.

A natural reaction to this catalog of gloom is to view complexity as a deep flaw — the root of all evil, so to speak. We look for the culprits behind complexity, for surely there must be something we are doing wrong. Banish the culprits and the problems leave with them. Is this really the case though? Maybe not.

For instance, delay or weakness in requirements and other supporting information are perhaps an excuse for the deficits we see. However, given the nature and purpose of software, such defenses are likely to be false cover for deeper problems, to a large extent in the software itself. Moreover, with the appropriate structural and theoretical foundations in place in software, the same foundations can be used to bolster systems engineering activities and products such that software

[1] Typical examples are sequence engines, forward chaining rule engines, or straight code.
[2] Typical examples are messaging systems, global variables, or straight function calls.

gets what it needs from them. Seeking to displace the software complexity problem to other processes is therefore not the purpose here. Instead, it will be generally assumed that software is consigned a reasonable place in the life cycle and workflow to fill the role of system flexibility for which it has been created, and that supporting processes are equally well positioned and structured. If they are not, they need to be fixed, but fixing them will not solve the software complexity problem.

Complexity symptoms could also be described as largely a consequence of software scope having exceeded the resources of projects to master it. If so, then it might be said that this is simply an outcome of projects scrimping on software investment and reaping the bleak consequences of their penny-pinching. Indeed, there is plenty of anecdotal evidence for this, with some managers claiming their only recourse for holding the line on software scope to be an artificial cap on spending. Whether one sympathizes with this strategy or not, the inevitable hair-raising climax before launch (or some other critical episode), as resources are finally released to complete the job, has become almost routine. This dilemma apparently stems from a broadly held perception that software would grow to an unbounded extent, if left to its own devices. In fact, there is a hint of this in the statement of Special Interest 2 of this study, which refers to "unnecessary growth in complexity" and "rejecting requirements … to keep systems lean", suggesting that the problem results from a certain lack of restraint. Software developers have even been accused on occasion of lacking the normal sensibilities of "real engineers"; this despite the admirable fact that software cost, as a fraction of total budget, has remained small and more-or-less level, despite *exponential* growth in software functionality from project to project, compared to other system elements. Indulging in such finger pointing, however, is also not the purpose here. Instead, it will be generally assumed that both developers and managers operate with the best intentions. If they do not, they need to be fixed, but fixing them will not solve the complexity problem.

To be sure, even the best-funded efforts with modest requirements and relaxed schedules are capable of producing software that is inappropriate, brittle, hard to maintain or reuse, and the source of a seemingly endless stream of bugs. To some extent, this can be attributed to basic errors, where the software is not built as intended; and these errors are themselves often attributable to ill-advised programming practices. However, such issues arise from the inherently intricate nature of software in general, mostly unrelated to the application at hand. Since it tells us little of the root causes for externally driven complexity, addressing software coding integrity is *also* not the purpose here. Instead, it will be generally assumed that best practices and tools are in place for the generation and validation of reliable code. If they are not, they need to be fixed, but fixing them will not solve the complexity problem.

We see then that complexity is not merely a symptom of some other shortcoming. It is generally understood that even the best managed, highest quality software developments are confronting an issue that goes beyond present methods to contain it. The most "correct", restrained flight software can be capable of astonishing misbehavior, when put in the context of actual flight environments and scenarios, and every year the complexity of these situations and tasks grows. It is our rising inability to fully understand the broad implications of what we do in software, and *to* software, that has created the crisis prompting this study.

Assuming, therefore, that complexity is a fundamental issue in its own right, independent of human and software frailties, it is necessary to tackle it from a holistic, system point of view, understanding the nature of engineering complexity itself (software or otherwise) and the special role that software plays in its management. This is the perspective from which the approach outlined here has been derived. Foundations for this perspective are laid in the next few pages.

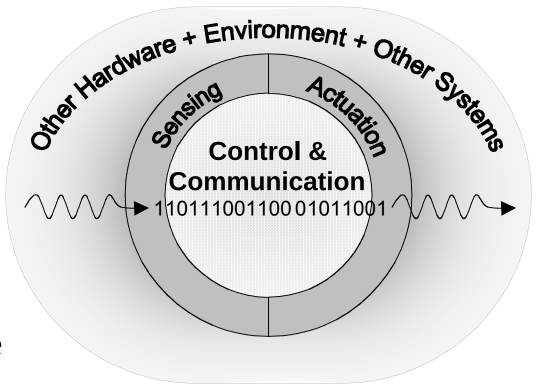
1.3 Sources of Software Complexity

Early flight software systems performed a relatively small number of tasks, primarily because that is all the computers of the time could do. This consisted at first of basic command and telemetry functions, drawn into software to provide design flexibility. Attitude and pointing functions followed quickly, along with various hardware management functions, and eventually a rudimentary form of fault protection. In these initial stages, software essentially supplanted equivalent hardware functions.

Of course, not all hardware functions can move into software though. To see this most clearly, it helps to imagine the canonical robotic[3] spacecraft from an informational point of view. This may not seem to be an objective point of view, since large parts of any spacecraft are dedicated to purely physical endeavors (e.g., the collection, transformation, storage and distribution of energy). However, with rare exceptions, such elements are generally there in service of the informational objectives of the system[4], and in most cases are dependent on the informational parts for proper function. Moreover, it is generally the case that the informational parts of a system need to be “cognizant” of the non-informational parts (a topic addressed below). Therefore, an informational point of view turns out to be a broadly interesting and helpful point of view for both systems and software engineers.

From an informational point of view then, systems may generally be divided into four basic domains:

1. **Sensing**, either the system or its environment — the transduction of physical phenomena into information.
2. **Control**, of the system or its environment — the interpretation of sensing information and production of appropriate action information in order to effect changes commensurate with system objectives.
3. **Communication**, either to or from other systems — the storage, transformation, and movement of information, often in service of control functions.
4. **Actuation**, upon the system or its environment — the transduction of information into physical phenomena, either for control or for communication.



3 Functions are shared in crewed systems between software and humans, so they are harder to describe. Nonetheless, the ideas outlined here are extensible to such systems.
4 Physical objectives are becoming more common, but their inclusion would not substantially alter the conclusions here.

This may seem like a short list, considering all that systems do, but any informational function that does not contribute in one of these four areas probably is not needed on a robotic spacecraft.

In the picture above, with information wrapped inside a physical perimeter, sensing and actuation are necessarily hardware functions. Everything else within this physical perimeter, however, is at least a candidate for implementation in software (or for other logical implementations, such as gate arrays: essentially re-hardened software…but that is another story).

The distinction here between physical phenomena and informational phenomena should not be interpreted to mean that informational elements are not physical. Rather, the nature of informational elements is that their physical implementation is incidental to their function, in the sense that their function can generally be described in terms of representational or transformational concepts that make no direct appeal to physical phenomena. Thus, despite its home on physical hardware, software lies essentially within the realm of information.

The reason for converting informational functions into software is that software is *soft*, naturally — a universal machine, providing virtually unlimited flexibility, which can be adapted and changed at will. Moreover, software remains malleable (at least in principle) throughout development and into mission operations. The resulting economics of software, being virtually free to fabricate and transport, is unique in the engineering world. This leads to a necessarily different life cycle, where design is an incremental and iterative process (some would say trial and error, though it will be shown here how to avoid that), in contrast to hardware, where little iteration is affordable.

The value of this flexibility is twofold: first, in reducing the scope and increasing the standardization of more expensive hardware, as mission unique functions are moved to software; and second, in preserving the adaptability of the system overall, as evolving understanding of the mission is accommodated in software. Both of these features are relevant to the complexity problem, because they explain many of the challenging scaling characteristics of software.

For example, as capabilities mature and become more complex, changes to systems happen both at their physical perimeter and in their logical interior. However, the interior tends to grow in a markedly strong disproportion to the perimeter (see the sidebar at the right for one example), while recurring perimeter elements simultaneously grow "thinner" as common hardware capabilities are standardized.

A similar scaling effect occurs in the mission dimension,

A Scaling Effects Example

Adding reaction wheels to a spacecraft creates a modest amount of hardware complexity (each with a housing, bearings and wheel, motor and tachometer, power supply and switches, interface and control circuitry, temperature and other measurements, and cables).

In software, there are counterparts for each of the active or dynamic hardware elements, in order to monitor, estimate, and control their state.

In addition, there must be provisions for configuring the wheels appropriately to each scenario or mode, managing their redundancy and accounting for associated capability differences, using them in precision attitude control, keeping usage statistics, monitoring and controlling speed and temperature, estimating and compensating for bearing friction and windage, determining and accounting for mechanical misalignment, avoiding operating constraints, modeling and anticipating momentum accumulation, dumping or biasing momentum, accounting for resulting ΔV, managing variable power usage, monitoring, diagnosing, and responding to faults and their side effects, providing telemetry of appropriate data, adjusting parameters for these functions, and so on.

where each extra phase and sub-phase adds disproportionately to the overall complexity. There are more modes to accommodate, more options to consider, and more complex transitions to be managed — an explosion of possibilities to design and validate. More complex missions also tend to suffer from an increasing number of competing design constraints, adding to the operating rules that must be enforced, and making time and resource management ever harder.

Another scaling effect appears in the form of increasing performance demands. Second order effects, which in simpler systems could be safely ignored, may require an order of magnitude greater sophistication to overcome, and there are systems in the works today, where even this is not enough. Each layer of performance improvement adds functions for monitoring, characterizing, and mitigating these effects, and the number of such effects grows substantially between layers.

Yet another scaling effect arises from venturing into ever more dynamic and uncertain environments. No longer is it possible to plot a single path through a scenario, where the biggest worry is a fault that puts the system into a safe mode. In more complex environments with obstacles to be skirted and hazards to be avoided, control becomes likewise more complex, posing requirements for rich situational awareness, broad robustness to variations, and numerous branching behaviors, in order to follow whatever path nature reveals — yet another deluge of obligations.

Attendant with this variety of scaling effects are also the growing hazards to their proper operation. The more functions a system is asked to perform and the higher the performance demands, the more they interact and the harder it is to keep them working smoothly. Things are likely to go wrong occasionally, even when there are no hard failures, and systems tend to be less forgiving in the consequences of error.

Not surprisingly, the time and analysis required to understand such complications exceeds what can be allocated to hardware development, given the need to commit early and irrevocably to a hardware design. Consequently, beginning soon after the introduction of computing to flight systems, software has become the complexity sponge enabling continuing progress in the presence of the compounded nonlinearities of scaling.[5] Under this economic reality, most new functions find their first instantiations in software.

1.4 Putting It into Perspective

The growing alarm with which this situation is being viewed is understandable, but any hope that this trend can somehow be suppressed, or even reversed, is futile. Yes, there are plenty of examples where designers have gone overboard, either by failing to push back on inflated requirements or by gratuitously embellishing a design, but these failings explain very little of the overall growth in software through the years. Indeed, careful examination of cases where “unnecessary” code is cited as a contributing factor to failure, show that the problem was not so much the extra software *per se*, but rather a poor model of adaptation or reuse that failed to

[5] Interestingly, there is a reasonable basis for arguing that the observed growth in software is in fact throttled under Moore’s Law by the limitations of computing hardware capability; it being the *latter* that establishes the rate at which mission complexity can grow. In this light, the exponential growth of flight software seems almost natural.

address complexity in appropriate terms. Moreover, there is no gremlin out there picking through the code and inserting errors preferentially into unnecessary functions. Remove every unnecessary function from a program, and you *still* have a very large and complicated program that is prone to the same sorts of error — and the next project's software is likely to be even bigger. Fretting over unnecessary functions only diverts attention from the real issue, which is that missions and systems will continue to present increasing challenges, as our reach extends ever further into space.

Therefore, assuming that we choose not to limit our ambitions in order to deal with software complexity, we must necessarily turn the problem around and ask by what means growing complexity can *continue* to be accommodated by software. Flight software, after all, has been contending with phenomenal growth so far, and has been doing so acceptably well, all things considered (and discounting the well-practiced drama at the end). The failures we have suffered are cause for soul-searching, to be sure. Nonetheless, there should be a sense of satisfaction that we have done as well as we have, with both flight software costs and failure rates remaining essentially level while flight software grows exponentially. If anything, it proves there is hope, for steadily evolving methods *have* been dealing with growing complexity for a long time.

Looking forward though, there is a general sense of unease in the flight software community that present methods have been stretched far enough and are in want of something fresh. Beyond recurring problems in software and a persistent gap between expectations and capability, expectations continue to race ahead. Needs to reduce operations costs on long duration missions, more difficult undertakings in remote or hazardous locations, further migration into *in situ* exploration, close cooperation of systems of systems, movement of crewed systems beyond ready help from the ground, and so on all suggest an end to business as usual. We should have no expectation that reiterating present methods on ever-harder challenges, no matter how much we refine them, will produce a better result; nor will timid retreat to some fortified foxhole of minimalism. Thus, despite there being a great deal of room for improvement in present methods, the *real* question is what needs to be done to reach the *next* level, a step that will unavoidably require dealing with current issues in any event.

We must look forward then to a new model of software design — not focus solely on the foibles of the present. What must happen next, though, is not merely acquiring faster computers, better software languages and tools, bigger budgets, longer development cycles, more tests, more rules, and more police to enforce them. The threshold before us is a conceptual one, requiring us to gather the lessons of the past into a firmer foundation upon which future advances can be built. That is the gist of the approach to be put forward here.

Moreover, it is not just a flight software leap that is required, but rather a change in the way we think of systems overall. The understanding and accommodation of all that these scaling effects throw in software's direction are, after all, systems engineering issues. No solution to the software problem will be found without a close companion in the systems world. The implications of this should become apparent once the nature of complexity is clarified. As we shall see, what is wrong is not unnecessary functions, but rather that we introduce unnecessary complexity in the way we specify and implement necessary functions.

2 The Nature of Complexity

It will be helpful at this point to be more particular about the word “complexity”. This term is often used in an absolute sense, suggesting that complexity is some intrinsic property of a system, fixed, and independent of any perception of it. Such a view is self-centered, of course, as anyone can attest who has witnessed the ease of a master in some arcane skill. As with beauty, complexity is in the eye of the beholder.

At the core of the software complexity issue, therefore, is **understanding**. When software is late, incomplete, poorly specified, and hard to cost accurately, this may well be because its developers are working in unfamiliar, dynamic territory that leaves them groping for complete, accurate **understanding** of the issues software must address. When software is not easily reusable, awkward to maintain, or hard to operate, this may well be because intricate arrangements of highly interactive parts make **understanding** of the software difficult. When software is brittle or capable of unpleasant surprises, this may well be because lack of thorough **understanding** leads to compromise and inflexibility, where changes, or deviations from routine use become dangerous. In each case, the consequences of complexity are the toll we pay for lack of **understanding**. That is, complexity is a measure of how hard something is to understand.

2.1 Patterns in Science

Clearly, complexity is not unique to the software world, or to engineering in general. Humanity’s attempts at understanding far predate any technology. Such understanding has generally been confounded in a few recurring ways, such as by the number of different kinds of parts of a thing one must be aware of, by the number and irregularity of interconnections among these parts, and by the difficulty of effectively describing each part in isolation. Such features can pose an intimidating landscape of endless, unpredictable variety. By finding **patterns** in the chaos, though, people began to assemble understanding; and with understanding have eventually come the power and control that we associate with modern technologies. What was a world of magic and mystery to our distant ancestors has been revealed by science as something we can largely understand.

In science, the patterns we seek involve 1) **recurring structure** — the invariants among items, which otherwise may appear on the surface to be different, 2) **layered descriptions** — ideas explained in terms of what is already understood, and 3) **separation of concerns —** limits on what must be considered at one moment. We see these patterns at work, for instance, in biology, where DNA is the recurring structure that governs most terrestrial life forms. Similarly, biology builds upon deeply layered descriptions from elementary particles through atoms, molecules, biochemistry, organelles, cells, and organs, leading finally to plants and animals, ecologies, and beyond. Yet, in this hierarchy it remains possible in varying extents to separate the concerns of photosynthesis, mitosis, protein formation, and many other biological activities.

Note that, in this process of assembling patterns, it is not enough merely to categorize and name the things one knows about — though this may be the seed for eventual understanding. What one really cares about is *explaining* patterns, in order, as Albert Einstein said, to “cover the greatest number of empirical facts by logical deduction from the smallest number of hypotheses or axioms." With good patterns, the world need not appear more complex with each novel circumstance, as long as each is explainable through already understood patterns.

This observation has elevated certain principles, such as Occam's Razor and the Principle of Parsimony, to the stature of meta-laws in science, where much of the undertaking becomes one of separating the **essential** from the **incidental** in order to expose the true sources of order in the world. The elegance, or even beauty, of the result is not just a feature the cognoscenti come to appreciate deeply; it is in fact a mark of validation for which all scientific endeavors strive. In this way, we see **elegance** as **the obverse of complexity**: the apparent complications of one side grounded and explained in the patterns of the other; a laudable goal for engineering endeavors, as well.

In the search for elegance, it is the nature of science to steadily extend the reach of patterns: the power to better explain what we see, to control it with greater dexterity, to extend our understanding to new situations, and to build layer upon layer into new concepts and capabilities. As this sophistication has unfolded, patterns have tended to become conceptually more abstract and layered — aided by careful conceptual separation, which enables interplaying patterns to explain complex phenomena without too many layers. For instance, evolution, which was first appreciated in a biological context, can be explained conceptually as a distinct phenomenon, without appeal to organic molecular mechanisms. Scientific patterns are therefore woven together in multiple conceptual orders, not dependent on any single reductionistic regime for all things.

2.2 Managing Engineering Complexity

Should we expect the same ideals in engineering? Let's see.

In nature, the patterns that govern the world already exist; it is the work of science to discover them. In human affairs, however, where we wish to avoid chaos, we must impose patterns. These are the languages, social customs, game rules, constitutions, monetary systems, and so on that create a means of civil discourse. These notions find counterparts in the technological world through conventions for container shipping, the electrical power grid, television and video broadcasts, internet protocols, and so on — thousands of examples, all of which represent human agreements on mutually advantageous engineering patterns.

As in science, patterns in modern technology are also many layers deep. Desktop computers, for example, are patterned assemblies of dozens of components, each built, technology-upon-technology, through perhaps dozens of layers, bottoming out in basic materials and ultimately in quantum mechanics — the end result, a marvelous amalgam of natural and technological patterns, which is just the root of continuing layers in the computer's software. Thus, it is as much the aim in engineering to build on prior engineering patterns, as it is to build on scientific patterns. In fact, engineering art may be defined fittingly as the convergence of social and scientific endeavors in pattern building.

Improving patterns are what enable progress … as long as one can depend upon the assembled pieces! The point, of course, is that all of this falls apart without concerted, sustained effort. That is, **it takes energy to maintain the order that engineering patterns offer**, a phenomenon sometimes referred to as the **Law of Design Entropy**, in analogy with its physical counterpart. This means there will need to be more functions to hide details and mask variations, more layers upon which to build functionality, more partitioning to isolate distinct functions, and so on, in order to impose the recurring structure, layered descriptions, and separation of concerns we depend upon to establish order. Design without such patterns leads to confusion and paralysis, because the resulting complexity makes systems too hard to analyze, build, and operate, too hard to update with new functionality, and too hard to change or reuse — *a fairly good description of the present complexity crisis in flight software*. We have let design entropy get the better of us.

2.3 Finding Design Patterns

So, where do good design patterns come from? To our great fortune, much of the work has already been done for us. For instance, in software several layers of patterns already reside above the computing hardware, beginning with the programming languages that organize processor instructions, memory, and other functions into a humanly accessible structure. On this layer are added data structures, algorithms, and other core elements of computer programming, provided in broadly used code libraries. Operating systems and associated common service functions such as storage and networking fill out the programming base. Then, as programs grow from this base, principles of good design become important (modularity, low coupling, high cohesion, information hiding, encapsulation, extensibility, standards, and so on), for which there is a large academic and commercial base to draw from.

To this point, the pattern layers are generic. Going further though, we see an ever-growing wealth of capability in higher level services, application frameworks, development tools, and so on, to the extent that today it is possible to invoke millions of lines of software capability in a new desktop computer application through a mere handful of pages of new code. That is, one can do this if one swims in the great ocean of personal computing and the web. Somewhere in the progression though — more or less around the point where patterns transitioned from generic to discipline-specific — flight software (along with many other less common specialties) has been left on its own, and we have had to become responsible for our own patterns. How well we fulfill this responsibility depends on how well we grasp the fundamentals of pattern building. (At a basic level, the techniques used to define patterns in software are essentially the same as those for any other discipline, so in the following description, design elements can be thought of in terms of either hardware or software.)

Two of the simpler of these ideas are the notions of **decomposition** and **hierarchy**. Complexity considerations suggest decomposition of systems into many small, easily understood elements. However, the smaller the elements the more of them there are to understand at a system level. Hierarchies address this problem by taking advantage of a tendency for clustering in relationships, advantageously grouping functionally related elements together and presenting them to the rest of the system as a single unit — something that can be done progressively at several levels. The process by which this is accomplished is commonly referred to as **functional decomposition**, the mainstay of systems engineering and design for both hardware and software. Functional decomposition leads naturally to a single containment hierarchy at the implementation level, wherein each element may properly be said to reside within some element of the next higher level. Thus, one speaks of system, subsystems, assemblies, subassemblies, components, and so on.

In the hardware world, where it makes little sense to describe, for instance, an assembly that resides simultaneously in two subsystems, this makes great sense. It does not take long though to realize that distinct copies of the same assembly **design *type*** can reside in more than one subsystem, and likewise at other levels. This separation of the notions of design from implementation is one of the first big steps in engineering patterns, especially when the copies become interchangeable — a strikingly simple idea, yet the great leap behind modern industrialization, once it was mastered.

The next (now) obvious step in the evolution of this idea is the realization that interchangeability is a property only of interfaces (and other externally expressed attributes), not of the elements themselves. Two elements may in fact be different inside, as long as they present the same "black box" façade to the external world. The values of such **uniform interfaces** across a system, while hiding (encapsulating) internal differences, are broadly appreciated across the engineering world, from simple power plugs to complex GPS satellite signals.

Of course, then one realizes that not all interfaces need be exactly identical in order to interoperate, as long as certain subsets of their features are invariant. Another major step is taken, therefore, when the **separation of invariants from variation points** becomes a formal part of engineering patterns. We see this in universal interfaces such as USB and FireWire.

Now, something very interesting happens! It is no longer necessary to talk just about a specific element or interface design, of which copies are made. One can now describe ***classes* of designs** that share certain features; and among classes, one can describe classes of classes, and so on, building a completely different kind of engineering hierarchy, unlike the containment hierarchy of functional decomposition — a **class hierarchy**. Of course, it has always been proper to speak generically of *type* hierarchies: consider Spirit, for example, which is a Mars rover, which is vehicle, which is a machine, which is a human artifact, etc. However, we can make such statements much more precise from an engineering point of view, such that the notion of class carries with it certain powers to explain in an unambiguous, constructive way — the same sort of expectations we have for patterns in nature. The introduction of **abstraction** in this manner is what allows one to connect nearly any sort of modern camera to one's computer, and have it recognized as a USB device of class *camera*, which is of class USB mass storage device, which is of class mass storage device, which is of class device, and so on. Each classification carries successively additional expectations as the class is specialized, and with these expectations in place, everything naturally clicks.

Remembering the essential features of patterns, we see in the concept of class 1) recurring structure, allowing us to reduce the information required to describe or use related items, 2) layered descriptions, allowing us to design interfaces to conceal details, and 3) separation of concerns, allowing us to focus attention on matters of interest. For instance, all USB cameras are handled alike (recurring structure); retrieving images from cameras is the same as retrieving files from any storage device (layered descriptions); and one need know nothing of optical or other characteristics of a camera to be able to find the images it has stored (separation of concerns).

One can extend these same ideas from single elements to collections of elements, where certain **recurring groupings** (*compositions*) **and structures of interaction** (*design patterns)* can also be described, and if appropriate, described in further levels of abstraction. Such patterns can describe collaboration arrangements that effectively solve common problems, rules of interaction that enforce engineering principles, regularity of form and relationships that facilitate reasoning and analysis, and so on. Common examples range from specific interaction protocols, such as TWAIN, which mediates communication between imaging devices (scanners, cameras…) and software applications (image editing, character recognition…), to general patterns, such as the so-called "iterator" that provides sequential access to a set of items without requiring knowledge of their internal implementation. Collections of such patterns are routinely used to express essential architectural features and serve as the basis of **design frameworks**.

A vital aspect of this notion of class is that class hierarchies are not exclusive. Classes define designs, not instances or archetypes of that design, so when a class is described in terms of other classes, this implies no exclusive containment.[6] Any of the constituent classes could participate in any number of other compositions. In this way, multiple overlapping abstraction hierarchies can coexist in a design, reminiscent of the interwoven conceptual orders within science. This is important, because it **avoids the tyranny of a dominant decomposition**, where One True Hierarchy (as in functional decomposition) necessarily relegates all others to collateral status.

This turns out to have a profound effect on the way one views systems. Without the imposition of a primary ranking among elements, architectures tend to flatten (fewer layers) — generally a good thing, when understanding is at stake. This relaxes artificial constraints on relationships, while maintaining a separate identify and structure for each abstract hierarchy, and allowing overlapping concerns to be managed more transparently. To the extent that layering remains as an architectural feature, it describes just one dependency ranking (typically element design dependencies) without necessarily restricting others. This opens up astonishing possibilities.

2.4 The Payoff

We have come a long way from the typical decomposition hierarchies of exclusive containment that are used to describe how hardware is organized. In such concepts as class and design pattern, modern electronics technology has found a potent mechanism for flexible design, wide-ranging interoperability, rapid innovation, efficiency, and reduced cost. The abstract patterns at play are a product of intense industrial and academic interplay at a far greater level of sophistication than the basic interface standards of most other industries.

These ideas have found their greatest payoff, however, in the software that runs these devices, and just about everything else these days. The software community has run with these ideas to create the modern world of networked computing, where all kinds of abstract ideas with no counterpart in nature have been realized, many in forms so compelling that we treat them as concrete entities. Meanwhile, "real" world patterns are being re-rendered gradually into software abstractions, with broad expectations (given Moore's Law) that people will be spending increasingly greater fractions of their lives working and playing together within virtual realms that are not much different from the real world, except that magic there will be genuine.

[6] In fact, it need not imply containment at all. Abstractions do not contain other abstractions. Instead, one generally speaks of composition or **inheritance**, where new abstractions are constructed by combining or extending existing ones. The extension of one class to two or more distinct subclasses is called **polymorphism**, a powerful way to implement recurring structure.

Third Law of Prediction: "Any sufficiently advanced technology is indistinguishable from magic" [Arthur C. Clarke].

The remarkable message to take away from this is that it is being accomplished as a loose collaborative effort spread across the globe, despite its *extraordinary* complexity. Steadily improving patterns are making this possible.

The path forward, therefore, seems clear. Patterns enable complex designs by making them less complex than they might otherwise be; and the better the patterns, the easier it will be to manage growing complexity.

Not just any patterns will do; they must be good patterns — elegant patterns — as described above. Moreover, they require continuing investment to establish and maintain them, and to build and improve upon them. Failing to appreciate and act upon the abiding criticality of this investment has been a great wrong inflicted on many software efforts, and in the case of flight software, largely explains our present predicament. Yet, there can be no doubt that flight software developers understand this. Thus, for such a ubiquitous idea, one must ask where the obstacles lay in its application.

2.5 Pattern Pitfalls

It turns out there are a number of ways to stumble in applying the patterns of abstraction described above.

First is the presumption that, if simple is good, simpler must always be better. In fact, the implementation overhead of more patterned designs (not to mention the investment required to develop and sustain them) is frequently cited as a major reason to avoid them. Some have even gone so far as to characterize the introduction of pattern to design as unnecessary complexity! This is a denial of the Law of Design Entropy though, which guarantees that the innate complexity of the engineering challenge at hand will not disappear. Such false economy leads ultimately and inevitably to other, even larger overhead and cost later in the development. Given the difficulty of adequate response at this late stage, these impacts generally go unresolved, transforming into increased risk.

This is not to say that it cannot be overly expensive to apply the wrong patterns, or that the right patterns cannot be implemented in an overly expensive manner. However, this generally amounts to a failure to attain elegance. The answer thus boils down to choosing good patterns, not oversimplifying them.

Recall that the key to simplifying properly is to note that certain elements of a pattern are in fact incidental to the matter at hand. Rainbows, for example, do not require rain clouds or the Sun. The refractive and surface tension properties of water droplets in the presence of a directional light source are sufficient to explain the phenomenon[7]; and in fact, even water is an incidental detail. Any simplification, however, that removes or replaces an essential element of a pattern, leaves it capable of explaining too little: the result being unanswered questions, false

[7] Full explanation, accounting for supernumerary rainbows and other features, requires an appeal to wave mechanics.

conclusions, inconsistencies, and a steady accumulation of exceptional cases. Every quirk of oversimplification erodes understanding, thus *contributing* to complexity, rather than resolving it.

Thus, simplistic and simple are not the same thing. Yet in engineering, it is common to see the omission of essential pattern elements in the name of simplification, only to see unresolved essentialness reassert itself elsewhere, as essentials are wont to do. Whether or not one sees this as a reasonable engineering tactic depends largely on whether or not one is its stooge.

Second, we also know that complex results can flow from simple processes through self-reference, emergent behaviors, chaotic dynamics, combinatorial explosion, and the like. Having found a simple pattern is consequently not the same as having made the appearance or behavior of the system uncomplicated. Nonetheless, it often makes people ill at ease being asked to accept complicated results from a pattern that purports to explain them — engineers being no less susceptible to such suspicions.

In denying the ability of some pattern to explain complicated observations, there is a human tendency to substitute a simplistic pattern that does not even *try* to explain. Instead, complication is merely taken as a given. The simplistic pattern itself is then embraced as the thing to be understood, and one sets about blithely hacking through the untamed complication by brute force alone. Not only does this contribute to the unmanaged complexity we so often see, but a dependent culture may actually build around the feeding and accommodation of these complexities, such that their taming is seen as a threat.

In defending such tactics, the recurring argument against the rigor of more appropriate patterns is that there is no obvious way to validate the claim that these patterns do indeed explain or handle the complications of a system. The connections can indeed be unobvious to casual inspection, leading to calls for proof that can never be fully satisfied. It does not help that provable correctness has come to be misconstrued by some as the Holy Grail of reliable flight software design, despite its fundamental unachievability. The essential misunderstanding here is analogous to a belief that the use of provable mathematics in science necessarily implies that science itself is a provable enterprise. The same is true in software, where provability of certain essential software properties provides a valuable foundation upon which we can confidently build patterns of design that may not themselves stand up under the same rigor. With this understanding, adept designers appreciate the potential for so-called leaky abstractions, and design accordingly.

The *real* engineering criterion for acceptance, therefore, is not the absolute truth of the patterns we use, but rather, whether or not these patterns are something one can reasonably depend upon. As in science, this is something one can ultimately learn through analysis and experimentation to accept, until something better comes along. Elegance thus reiterates itself as a more important factor than appearance alone. It is how we gain confidence that we are moving in the right direction. Imperfect as they might be, ever improving patterns of design are our only defense against complexity.

To deny the inconvenient practicality of this more “scientific” approach is to accept complication *beyond* explanation. One common entry point for such subterfuge in engineered systems is the maddeningly common practice of assigning correctness to a system that successfully passes an assigned set of tests, as though this somehow amounts to a verification (i.e., a demonstration of the truth) of correctness. Even worse is smugly taking such "verification" as evidence that stronger patterns in its development were not really necessary, after all — assertions frequently wrapped in self-satisfied platitudes, such as "better is the enemy of good enough", "don't fix what isn't broken", and so on.

Nothing in this process though, no matter how simple the design, provides any insight into the larger space of behavioral possibilities the system might possess; and if common experience means anything, then it strongly suggests that this range of possibilities is as complicated as any that a more elegant system might be capable of, especially as complexity increases. The main difference is merely that the simplistic design lacks the patterns required to fully explain its behavior. So-called "verification" of a simplistic system is therefore a hollow victory.

We often try to make up for such shortfalls through further testing, broadly categorized under the heading of "validation", including tests (e.g., stress tests) specifically designed to probe the edges of sensible behavior. Yet such techniques provide at best an empirical basis for confidence, falling far short of the innate integrity of a principled conceptual framework. This is why finding the right patterns is so important, and why it is necessary to look past outward appearances to understand the merits of the underlying patterns. Only good testing *and* good patterns together add up to a trustworthy system.

A third issue in the engineering application of patterns is the careless ease with which they are so often set aside, having proven to be a challenge to follow. This applies broadly, but is especially true in software, where commitment to some pattern can be abandoned cavalierly in a single line of code. Such practice no doubt explains much of the hesitance to adopt broad patterns in the first place; and not having witnessed the benefits of broad patterns, engineers find them easier to dismiss. It is a vicious cycle.

What is lost in this practice, of course, is the conceptual integrity for which patterns are introduced in the first place. Most new designs start out with at least some notions of the overall structure giving order to the system. It is a common mistake though to believe the work is done, once the initial patterns are in place and attention shifts to implementation. Such efforts lack strong architectural guidance, so as design progresses and problems pop up, the expedience of a quick fix that sidesteps a pattern can be tempting. This is especially so, when the patterns behind the design are poorly articulated and supported, or disappoint by being overly strict, logically flawed, expensive, inappropriate to the domain, too weak or specialized to provide broad guidance, inattentive to essential issues, dubious in value, or cumbersome to apply in some other way. At this point, one of a few outcomes is likely.

One outcome is to initially accept the first few expediencies, rather than addressing the apparent flaws in the adopted patterns, since the workarounds are no doubt faster and cheaper to make than revisiting the patterns. Inexorably, one such compromise leads to another, each becoming easier to justify than the last, even while it becomes harder to validate, as any hope for order from the underlying patterns erodes anyway. Gradually, mindlessly, developments are led to their doom. Whether or not any pretense of the original structure is retained becomes moot. Along this path lies a brittle design so devoid of structure that even small changes are viewed with alarm. This fragility follows the design into operation, where only the tried and true is exercised with any confidence.

A related outcome leaves the problematic patterns in place, paying lip service to their adherence, but overlaying other *ad hoc* structure without ever claiming it to be patterned. The underlying patterns in these cases are not so much perverted as they are subverted through clever but inappropriate manipulation. The patterns may still be intact, but there is little practical difference in the outcome.

Another likely outcome is for the patterns to morph steadily in an attempt to remain relevant to the design. Exceptions are turned into features: an extra interface here, a few more configuration parameters there. Eventually, it becomes hard to discern what is pattern and what is design, since the pattern attempts to be everything to everybody — with just a little tuning. Such accretions have little explanatory power, because the patterns themselves are complex. Moreover, by bending to every compulsion, they fail to provide any real structure and guidance to the design after all.

However it is done, the unfortunate effect of these various outcomes is a system where engineering patterns are optional. We should expect nothing good from this. Imagine if the laws of nature were only followed *most* of the time or only for *most* things. They would hardly have merited the moniker of “law” for one thing, and it is doubtful we would be living in a technological world today. It is our conviction that the world is *not* like this that drives all of science. The world is not obliged to conform to elaborately contrived ideals. Thus, when we find exceptions, we can be sure it is our understanding of nature’s law that is at fault and not the actual patterns nature follows. Exceptions become the basis for new refinement, and our ability to understand gradually improves.

2.6 Choosing Good Design Patterns

What we ultimately seek in science are patterns that are both **stable**, in that they will not need frequent or substantial revision, and **fundamental**, in that they broadly address the important issues of the domain. We should strive for the same in our engineering patterns. Stability gives assurance of consistency and continuity throughout development, while focus on fundamentals ensures that the structure provided will be flexible and extensible to a broad range of applications within the domain.

If patterns are chosen well, then exceptions should be rare; and the rare exception should be treated as motive to revisit the pattern. This way, we get the outcome we need, a solid, coherent vision that ties the design together, guiding and sustaining it flexibly throughout development without erosion, accretion, or subversion. On the other hand, accepting serious conceptual flaws is a major source of error, arising from a basic loss of architectural integrity.

Putting good patterns in place is a real art, and definitely not something to be approached casually or hastily. Yet projects tend to invest far too little time on such structure, rushing from requirements to code in an attempt to demonstrate progress. Such behavior is often accompanied by the familiar argument of inheritance from a prior design, but this case rarely has merit, unless the prior design was build upon good patterns in the first place. Otherwise, the absence of helpful patterns will lead quickly to disintegration, and most code will be rewritten anyway. The stable, fundamental patterns beneath a design ultimately carry the real value.

Similarly, requirements do not help when they are cast in terms that either relate poorly to the chosen patterns, or are hamstrung by attempts to match poor patterns. In either case, it is hard to express system needs in an effective way, because of the enormous semantics gap between expressions of system interest and expressions of software implementation. Either software engineers get requirements they do not fully understand, or systems engineers write requirements they do not fully understand. We already know what that means: lack of understanding equals complexity — complexity that is incidental in this case, because it is not imposed by the application domain itself. Thus, with inadequate attention to choosing good patterns, poor requirements are essentially guaranteed. They will be incomplete, easily misinterpreted, hard to verify objectively, misdirected from the real needs, or downright incorrect.

A key to the development of good patterns is to keep in mind the principles of recurring structure, layered descriptions, and separation of concerns. But when we say “recurring structure, layered descriptions, and separation of concerns”, we mean recurring structure, layered descriptions, and separation of concerns *of what?*

This turns out to be a profoundly important question, because if one thinks it means *of the software*, then the whole point has been lost. Remember, the problems we try to solve with software are outside the software. Self-absorbed software is of virtually no value. Thus, when we define an abstraction in software, either it is a conceptual representation *of* something else, or it is the embodiment of a concept *for* something else. That “something else” may be other software, but that is beside the point. What matters is that one seeks recurring structure, layered descriptions, and separation of concerns in the problem to be solved. If the intricacies of that problem are not recognizable in the patterns applied to their solution, then the solution probably lies elsewhere, and all the software is doing is providing a shell for whatever that might be.

One might ask for example, what pattern is typically used to address the detection of problems in uplink communication with a spacecraft? If the answer is “a timer, restarted by an uplink command” (not uncommon), then it is logical to wonder how this pattern addresses communication schedules, worst case duration requirements, weather or emergency outages, constraints on antenna pointing, disruption of critical functions, obscurations or other interference, uplink errors, clock drift, time units, or other concerns commonly associated with the use of such a timer. The implementation of the timer is simple, but it leaves most aspects of the problem unaddressed. Moreover, because these issues are not addressed in the provided pattern, they must be addressed elsewhere — usually many elsewheres. Requirements for the individual issues are likely to be spread around, such that a coherent picture of the integrated function is hard to assemble. What one gets instead, because the implementation is almost entirely invested in itself and not in the problem at hand, is a software gadget. It meets the criterion of being a concept used by something else, but the “something else” that is supposed to embody uplink loss is missing in any carefully patterned form.

To reiterate, a key to the development of good patterns is to keep in mind the principles of recurring structure, layered descriptions, and separation of concerns — *in the problem we are trying to solve*. There will no doubt be many aspects to this problem, as in the example above.

However, knowing that different abstractions (recurring structures) can coexist and overlap, we understand that each should try to represent a particular idea or unique point of view (separation of concerns).

By way of illustration, we can think for a moment about the function of managing the star trackers found on most spacecraft. Is this function about optics, detectors and signal processing, astrometry and star catalogs, spacecraft attitude and rates, data collection, time tagging, power and temperature management, real time scheduling, or what? The answer, of course, is all of the above and more, but if the pattern we apply to deal with this is nothing but a catchall "star tracking" module with generic interfaces, we gain almost nothing. On the other hand, a separate set of abstractions and design patterns for each of the constituent concerns could provide major gains in assembling and understanding such a module. This is especially true when the same abstractions and design patterns (e.g., data collection, time tagging, power and temperature management, real time scheduling…) find repeated use for other items across the system or between systems.

Observations of recurring structure, and so on, are common, as might be expected. It also happens though that these observations are gradually refined, as details are untangled and incidentals are sorted from essentials. Under healthy circumstances, a regular occurrence, as understanding evolves, is the so-called *re–factoring* of abstractions and patterns to achieve more fundamental and complete concepts within each view, and better separation of concerns among views. One might come to appreciate, for example, that time tagging is only incidentally about when data is received, and is more essentially a part of data generation. A re-factoring that corrects this association helps to ensure that time tags will reliably mean what one expects them to mean, thus plugging a leaky abstraction.

It is also common to observe, not only recurring patterns of application, but recurring *principles* of application, as well. The principles reflect accumulating experience and understanding that can then be codified in patterns as a way to ensure best practice in the topic covered by the pattern. A general principle regarding measurement instances, for example, that their source and sample time should be knowable, is something one can make explicit in a general pattern for measurements, from which all measurements could inherit. This solidifies such principles far more effectively than reliance on hearsay or familiarity.

As patterns are identified, therefore, it is important to articulate the principles and properties of them that one values. The process is not merely about encapsulating the implementation of some set of requirements. The idea is to look for common solutions to common problems across applications and functions, while choosing patterns flexible enough to build upon and easy to extend without change. It is also important to broadly solve general domain problems, providing real benefit to development through explanatory power that supports design and analysis — not merely naming modules and interfaces (as in "the command interface of the star tracking module").

The product of such an effort is often a deep conceptual hierarchy, which can present its own issues of understandability, if it not properly factored. It takes diligence, therefore, to maintain conceptual integrity. With attention to this issue, flatter architectures will result that can provide useful abstraction, and still avoid burying essential details out of reach.

3 An Assessment of Current Practice

With this brief glimpse at the use of abstractions and design patterns in the management of engineering complexity, it is natural to ask how current practice in the flight software world stacks up. There is no equally brief answer to this question, though, that would survive direct comparison to any particular flight software implementation. The description here must therefore be limited to a broad outline. Nonetheless, a few observations might fairly be made, given that there must be *something* prompting the complexity study. Thus, with apologies to the exceptions to the following generalizations, here are some of the recurring design features one might expect to see.

3.1 Common Features

The most common approach to structuring flight software is functional decomposition. A typical architectural description at the top level or two is a block diagram (boxes and lines). There are no particular semantics associated with these diagrams, though in most cases boxes are modules or collections of modules, while lines show connections of some sort. Boxes will name functions, without giving much of a hint (except through familiarity or tradition) regarding their nature. Whether boxes are processes, libraries, shared data, files, exchange media, or something else must usually be inferred from some separate key or description. Lines say something about implementation dependencies — generally an interface of some sort, also named but not explained.

Some of the broadly used interfaces, such as “command” and “telemetry” interfaces may be documented carefully. Other interfaces at the high level are also likely to be documented in some way, though this is quite uneven, and many are documented only in the source code. The latter will tend to be merely syntactic descriptions, rather than careful descriptions of associated behavior, usage assumptions, and so on.

The remainder of the architectural description in most systems will commonly refer to the way in which certain broadly used functional modules are implemented.

Communication within the system is generally either through direct procedure invocations or through some sort of messaging middleware (increasingly common). Shared variable interconnections are also still common. Whatever the case, the interconnection architecture is generally agnostic about anything beyond the basic mechanics of the implementation.

Execution is generally multi-threaded with preemptive real-time scheduling accomplished through priorities, generally arranged in rate-monotonic or deadline-monotonic order. Rate groups are common, and may or may not be managed centrally. Shared software resources are generally protected through semaphores or comparable mechanisms, though unadvisedly, timers are also widely used. External resources and other items requiring coordinated use tend to be managed through more modal mechanisms, if at all (enforced externally in some systems through operation rules). Protections often find their way into software in an *ad hoc* manner as issues are discovered during testing, rather than through any methodical and pervasive coordination system.

Storage has shifted gradually from operator management of memory blocks and sequential storage systems to software management of heaps and file systems. Mechanisms generally manual invocation rather than using garbage collection or other more automated management approaches. These mechanisms also remain largely unconcerned with content. Operators are still involved in overall allocations.

Command and sequencing modules generally provide a basic scripting service. These scripts, or sequences, are often multi-threaded, but sharing a common time base, such that thread advancement remains synchronized. Advancement can sometimes be conditional, waiting for certain expected events. The procedural "language" of these scripts normally provides little if any support for abstraction. The commands issued by these sequences are typically either simple asynchronous procedure calls or messages to other modules, or they are direct commands to hardware.

Telemetry systems vary widely, but whether data is polled centrally according to some schedule or is generated by the modules and forwarded to a central location, data tends to consist either of large products or of *ad hoc* bundles of small items. Abstraction is generally limited from the telemetry system's point of view, providing basic services for storage and transport with essentially nothing to say about content.

System coordination functions are often provided to implement the primary modes and redundancy configurations of the system (including "safing"), generally described in a small handful of interconnected state machines, organized in a shallow hierarchy. "Mode" generally refers to a simple configuration choice: what equipment, algorithms, and so on are active, and what key settings dictate performance. Transitions usually respond either to sequenced commands or to *ad hoc* system events. Modes are augmented with counters, flags, timers, and the like to keep track of sub-states, mark progress, remember objectives, record events, and so on.

Fault protection in most systems resides partly in local *ad hoc* implementations that vary across modules, and otherwise in one or a few central but separate modules for faults requiring system level assessment or response. Faults are generally detected through a wide variety of triggers looking for symptoms, ranging from explicitly detected constraint violations, to unexpected hardware behavior, to excessive performance errors, to broken deadlines and more. Thresholds, persistence limits, time delays, priorities, response levels, and the like generally round out a parameterization of the detection system, which is then tuned for correct behavior.

The fault response system for these triggers varies little in substance across systems, though the actual implementation varies a lot, ranging from accommodation in existing state machines, to sequence "engine" scripts, to forward-chaining rules. The primary feature that delineates systems is whether responses are executed sequentially or concurrently. If sequentially, the onus on designers is to put off other demands as little as possible (as in juggling acts), but there is seldom any architectural support to ensure that delays are tolerable. If concurrent, conflict resolution tends to be through adept timing (as in figure-eight racing), though one occasionally sees basic locks of some sort employed to make coordination more explicit. Resources are rarely managed explicitly, except in a gross way, such as indiscriminately switching off non-essential loads.

Progress through responses tends to be captured in various additional modes, counters, flags, timers, etc. Normal activity is generally abandoned when a serious fault is detected, except in so-called critical scenarios, which are usually handled through what amounts to special-purpose software (even if implemented in part though command scripting), specifically designed for these particular episodes.

In all of this, the state of the software — what it knows, what it is intending to do, what progress it has made toward these ends, and so on — is captured in a scattered and often obfuscated medley of *ad hoc* details, public and private, with little organizing structure or semantics beyond what each module developer assigns to them individually.

At lower levels, most architectures generally fragment into an assortment of discipline-specific (and programmer-specific) structures and algorithms, the common “pattern” being for module owners to deliver relatively large, opaque, vertically integrated modules — stovepipes. Beyond middleware, core function libraries, and other low-level system services (plus the need to tie into the generic infrastructure of commands, modes, telemetry, and so on), architectures place little demand on the internal character of these functional silos.

The degree to which abstraction and design patterns exist within modules is left largely in the hands of module developers. At this level, we typically see more appreciation for pattern. However, these ideas are generally of a specialized nature, and rarely escape the local confines of the module. Moreover, where the abstractions *do* represent common concerns, there is a tendency either to reinvent them in multiple places or to consolidate them externally into a black box service or library of some sort, either way, losing the benefits of shared abstractions.

Reuse of such systems follows the so-called “clone and own” strategy. Some prior system has an X, a Y, and a Z, and a new system needs an X', a Y', and a Z', so all one has to do is copy the old system and change it to match the new functional requirements, which should be sort of the same. The tuning required is often “aided” through an exuberantly munificent parameterization of the software, on the theory that parameter changes are benign, since they do not change the “logic” of the software, despite the fact that software behavior changes either way, and often in surprisingly illogical ways. The originators of such designs tend to become vital for their reuse, inspiring unwise pronouncements such as, “you reuse people, not software”. Regardless of the dubious merits of this approach, however, the real issue is that the *only* architectural variants of these designs are parametric. Once these narrow options are exhausted, everything else is reengineering.

We see similar thinking all the way down to the core tools of development. For example, in a rather peculiar reversal of progress, flight software implementation these days is almost exclusively in C. This is a minimalist language designed around 1970 to make compilation into machine language more straightforward (for compilers, not for people), and to ease access to low-level machine features (for systems programmers, not application programmers). Given this genesis, C provides little direct support for abstraction (not to mention a great many other features found in modern languages). This is not to say that one cannot implement such things in C. The problem is that one can implement virtually *anything* at all in C, good or bad, with equal facility. The language itself does not help you do anything well, so without a great deal of skill and effort, the Law of Design Entropy makes “bad” things far more likely. Consequently, making C palatable as a language for highly reliable systems requires a suite of rules and external checkers to assert a level of discipline lacking in the language itself — and it still falls far short.

3.2 Common Problems

It is worthwhile to look more closely at a few of the common problems one finds in many flight software designs.

3.2.1 Ripple Effects

Changes are inevitable in software — intrinsic to its application, in fact. Once a design is in place, changes are generally not motivated by broad issues. Far more common is that a particular item needs to be changed to address a discovered problem or an altered requirement. As development progresses, however, one often encounters a great deal of trepidation for even the simplest change.

The reason for this, of course, is inadequate understanding of the effects this will have elsewhere in the system. For example, adjusting a parameter might alter timing, which might change the order of events, which might expose a constraint to potential violation, which might trigger a fault response, which might interrupt normal system operation, which might cause a critical objective to be lost. Such scenarios are both legendary and routine — the famous butterfly effect of chaos theory, writ in software — and the nature of the beast, as we have come to know it.

3.2.2 Tuning

Parameters abound in typical flight software designs, some for good reasons, but many not. The reasoning generally goes something like this: Why add a semaphore, when a judiciously chosen time delay can avoid a resource conflict? Why check for completion, if one simply waits long enough? Why bother explicitly ordering tasks, when a priority adjustment will do? Why check for overflow, if the queue can be made a little longer? Why refine a monitor prone to false alarms, when a persistence threshold can just be raised? Why logically sort through conflicting data, when a weighted sum is so much easier? Why arrange for fault response preemption, when adjusting thresholds can get the order right in the first place? . . . Besides, *everyone knows* that it is harder to change the logic of a program than to just change a set of parameters, right? Thus, repeatedly, a parameter takes the place of something else that would directly address the problem on principled theoretical grounds. All you have to do is get the parameters right — all several thousand of them. Therefore, repeatedly, we witness rounds of adjustment, where a tweak to correct one problem pops out a new problem somewhere else.

Parameters also become problematic when multiple parameters need to be set collectively to maintain some involved relationship. Neglecting to change all appropriately is obviously a problem, but it is not always obvious (or well documented) what these connections are. The converse of this occurs when related issues are conflated in a single parameter. Sometimes this is unavoidable, such as when a detection threshold must be adjusted simultaneously for high probability of detection and low false alarm rate (though more discriminating tests may be possible). Sometimes it is just a poor design choice, as when jitter requirements are compromised to achieve response speed, instead of applying feed-forward actions. In either case, finding just the right value for the parameter can require a lot of fiddling, and there are no guarantees that a viable set even exists to handle all the cases one might encounter.

Pervasive tuning is such a common phenomenon that it is even formalized in some flight software development processes. Moreover, the parameter landscape is normally so fraught with unmarked landmines that there are commonly additional processes to document, review, validate, restrict, and safeguard parameter selections and updates. Having found a parameter set that "works", it can become nearly untouchable, even in the face of obvious weakness, because the consequences of change are too hard to understand.

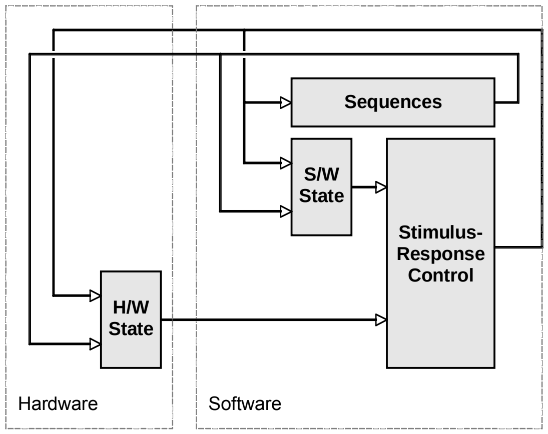
3.2.3 Stimulus-Response Misbehavior

The main purpose of flight software is to control the system in which it resides. This requires coordinated, thoughtful action. Yet, flight software generally possesses very little deliberative capability. Instead, systems are built to react reflexively to the circumstances at hand. Whether or not the longer-term consequences of such actions are appropriate is determined mainly via *a priori* analysis, validated by test. Thus, as long as the circumstances encountered in flight fit more or less into previously assessed cases, this has a reasonable chance of working. In attempting to limit the number of cases that need to be understood, however, it does tend to force both implementation and operation into a modal style. This can become a slippery slope to inflexibility and brittleness.

In this reflexive approach, the idea is that each stimulus deserving attention triggers a response that commands changes in hardware and/or software state.

With notable exceptions, such as spacecraft attitude, the state of the hardware under nominal conditions is largely either represented by itself (with minimally interpreted data from the hardware frequently taken as a direct indication of its state) or merely assumed to be as commanded (unless separate fault management intervenes). Because the hardware state is silent on its future, the software works with what it has and generally lives in the moment, as well.

Meanwhile, software state (modes, flags…) tends to be oriented towards its own function, acting in many ways as a peer to hardware state — both being equal targets of control intent. Thus, it is common to see hardware and software commands intermingled without differentiation in command sequences. Moreover, since future intentions are captured primarily in the sequence, which is typically a separate execution function operating largely open loop, software state outside the sequence is similarly silent on its own future.



In this style of stimulus-response control, hardware and software state are considered collectively, stimulating in certain events a response that will change hardware or software state. There are many ways to implement such behavior. Nonetheless, whether the implementation is through state machines, rules, scripts, or otherwise, this basic picture applies for the majority of software control behaviors. Yes, it is a control loop, but there is nothing intrinsic in this structure to guide control design. Reconciling the quality and consistency of data with needs, resolving conflicting intentions or system interactions, coordinating responses with each other or with sequences, and so on, must all be handled without the aid of this structure. Moreover, the structure is self-referential (discussed below), which is a good recipe for undesirable emergent behavior.

It is possible to exercise sound control principles within this structure, but not because the structure itself has anything to say about it. Therefore, it is easy to violate sound control principles, too. Where such principles *are* introduced, this is largely on a haphazard basis, with no general crosscutting structure to tie these applications together across the system. Either way, such systems have proven as a rule to be hard to understand, requiring a fair amount of trial and error to get “right”.

3.2.4 Multiple Versions of the Truth

The picture above has been drawn as though control is a single-minded affair. Never minding the fact that sequencing is already a separate control function, reflecting the minds of ground controllers, the control function itself tends to be highly partitioned. Actual systems are comprised of many interacting control functions, each responding to some subset of system state. This state information is subject to interpretation though. Hardware state information is often noisy or intermittent, and not always reliable. The semantics of myriad software states of all kinds are generally ambiguous. Different software modules may choose to use different combinations of data to make conclusions about the same thing. Thus, in typical stovepiped developments, these opportunities for inconsistency can result in a schizophrenic system characterized by contradictory behaviors. Even when mechanisms are in place to avoid outright conflict, episodes of looping behaviors where a system alternates between competing behaviors, are familiar to many test programs and a few unfortunate flights.

This dilemma can propagate into grounds systems as well, when operators (and their tools) are faced with volumes of raw data and various flight system interpretations. This is especially difficult when the data arrives in fragmented chunks, so that a complete picture unfolds unevenly over time. In any case, flight or ground, the consequence of being of two minds (or more) is risky and confusing. No system with multiple conflicting understandings of a situation can be said to understand anything.

3.2.5 Intertwined Control and Estimation

The failure to require a single-minded view of the system, even if it comes from multiple sources, can be traced to a lack of appreciation for the difference between information about state and knowledge of state. When data from hardware is interpreted directly as state knowledge, despite its flaws, no overt attempt is made to formulate a separate representation of state knowledge within the software. Nor in this case is there an attempt to assign a measure of uncertainty to this knowledge. Proper state knowledge requires one’s best, consolidated interpretation of *all* the evidence, given expectations of the behavior of the state and the available evidence, and proper state knowledge *always* includes the resulting uncertainty.

Instead, a frequent pattern in flight software is to treat hardware data directly as state knowledge unless its veracity is called into question by the detection of some anomalous condition. Data flows from input to output through splitting and recombining stages of processing with nothing in the middle that is recognizably a proper representation of state knowledge and nothing to ensure a consistency of opinion and decisions among the various paths. Accordingly, there is nothing to delimit impartial assessment of system state from objective-driven decisions on how to change it. That is, the amorphous transmutation of data from input to output yields no point in the software where a unique understanding can be obtained of what the software thinks it is responding to. This failure to separate concerns results in a design that is harder to test and harder to reuse.

As a result of this blurring of responsibility, one may find an inappropriate assignment of functionality within the system. For instance, data may be smoothed in order to reduce control jitter, with the consequence that transients become masked to other functions looking for constraint violations in the smoothed data. Such deficits from functional entanglement are often hard to discover, and the software behavior overall becomes quite hard to understand.

3.2.6 Confusion about Predicted vs. Desired Behavior

As mentioned above, hardware state and software state are often viewed as peer targets of system control. Unfortunately, the self-reference involved in the software part of this is problematic in attempts to explain behavior. Moreover, it is common to find elements of the hardware involved with computing to be among the controlled items, such that self-referential control arises through hardware paths as well.

One problem with self-reference is the difficulty of injecting into control functions understanding of the items under control, when these items include the control functions themselves. The usual way to avoid this is to arrange control functions into a command hierarchy, but this is seldom completely successful. Thus, one finds sequences that enable fault responses capable of disabling sequences, fault responses that request mode changes that alter fault responses, software that commands power to devices that can cause an under-voltage that resets the computer and restarts the software, software commands that poke patches or parameter updates into working software, and so on.

Even in a relatively clean control hierarchy, the general problem of understanding where in the hierarchy one is dealing can occasionally be tricky. For example, one might write a requirement that certain actions are to be avoided when a hardware element is in a vulnerable state, but this could be interpreted in software as an avoidance of these actions when software is in the mode that would normally put the hardware element in that vulnerable state. That would be an incorrect implementation for three reasons. First, the mode used to condition the avoidance might cover other hardware element states in addition to the vulnerable one, in which case the avoidance is overly restrictive, perhaps to the point of disallowing required operations. More insidiously though, a later change to the mode structure that enables the vulnerable state in additional modes could go unprotected. Finally, because commanding a hardware element to some state is not the same as having achieved that state (bad things happen, after all), the hardware element might still be in a vulnerable state, even though the mode has changed. This would leave it unprotected.

Such misunderstandings are common due to a general failure to make clear distinctions among various related control concepts (e.g., objectives versus expectations) and to carefully delineate control system layers.

3.2.7 Nearly Duplicate Functionality

As described above, stovepiped developments are common due to the use of functional decomposition in system design, as well as other factors. When a designer needs access to some capability clearly delimited elsewhere in the functional hierarchy, creating an interface to that function is the natural thing to do, and if there are multiple users of that capability, some commonality across interface implementations can be expected. Similarly, routine internal functions (math, file access, etc.) will generally prompt a call to some shared library or service. In a well-managed design, one should not find too much duplication.

Nonetheless, duplication seems to happen routinely. This is largely because anything short of identical needs is taken as justification for a separate design. Shared implementations, after all, require effort to seek out and identify the overlap, to negotiate a separation from other features, to sustain the shared capability as issues arise in each of the contexts where it is used, to deal with the proliferation of side effects from errors in the shared capability, and so on. Spreading out the development team, or trying to inherit such functions, only makes this more awkward. Thus, in the myopic wisdom of the moment, going one’s own way makes perfect sense.

Aggravating this tendency is the prevalence of generous parameterization as the primary means of adapting or tuning software to the application. Thus, one occasionally sees instances of functions tuned in or out of an application by parameter changes, while leaving the code in place. When the code cannot be the variable in adaptation, it tends to fall into locally tailored forms, with consequent duplication of functionality.

The longer-term consequences of such tactics, however, are bleak. Anyone trying to understand the system (not just one module of it) — how all its parts interact, how to debug it, how to operate it, how to reuse it — has a daunting task. What makes sense at a local level becomes a headache at the system level. Moreover, because of the scattered variations, there is no shared experience that gradually accumulates over time resulting in ultimate refinement into a mature capability confidently applied wherever and whenever needed. Instead, much of what is done on each project is a wasteful reinvention. Costs escalate, flexibility is lost, progress slows, and understanding remains ever trapped under the clouds of needless duplication.

3.2.8 Gadgets and Gizmos

When reviewing the sorts of mechanisms used in flight software for routine functions, the list is likely to include counters, flags, enumerated modes, IDs, addresses, timers, and so on. There will also be code to manipulate them at the appropriate times in the appropriate ways. There is no surprise here. These are basic building blocks of all software.

However, when one asks a spacecraft operator what they are doing, and their answer involves counters, flags, enumerated modes, IDs, addresses, timers, and so on, this is rather alarming, especially when their attention to such things is in service of something of system level importance. For example, the health of uplink equipment is obviously of prime importance, but if enabling the detection of problems requires operators to worry about when certain timer values should be set, and to what values, then something is seriously wrong. Unfortunately, it is not hard to find many other examples of this, or cases where dealing with systems at this level have proven to be detrimental.

Such gadgets and gizmos are common because they are quick and easy to implement, but dealing with the implications of simplistic implementation can be long lasting and hard, mainly because these designs do little to aid understanding of the system issues one cares about, and because their shortcomings add incidental complexity that further confounds understanding.

3.2.9 Test as You Fly and Fly as You Test

As described above, modal designs are popular, because they ostensibly limit the number of design cases one must analyze and test. When read narrowly, the inescapable “Test as You Fly, and Fly as You Test” dictum can be seen to reflect this idea. Indeed, this philosophy presumes that one *can* test all of the ways one might wish to fly a system, and that the system will *never* need to fly in any other way than what prescient designers and testers have devised *a priori*. With such rigid ambitions, it makes sense to build systems that are equally inflexible, and that very often is what we get.

Inflexibility, however, also connotes brittleness and fragility. By designing and testing for a circumscribed and discrete set of cases, one leaves systems vulnerable to any circumstance outside the norm. Yet the attitude seems to be that there is no need to accommodate such variation, no need to characterize it, and no expectations for it, if it happens. Maybe that is okay for equipment on the assembly line, hammering out one copy after another of some device, but it seems odd that we would accept the same for spacecraft exploring the universe. Nonetheless, it is routine to hear from operators that they will avoid even the simplest departures from practiced routines until they have thoroughly tested the variation.

Amazingly, this notion is even applied to fault management functions, which by their very nature are off the beaten path. Having tested the several hundred scenarios deemed adequate to cover the accepted list of failure modes in the prescribed list of operating modes, success is declared. Of course, everyone knows that no fault scenario in flight has ever transpired as scripted in some test. Nevertheless, if there is a change to the fault management software, some subset of this list will be repeated verbatim as feeble evidence that the change was done correctly. As likely as not, not even a small change in parameters or test conditions will be introduced in the regression test — repeatability being viewed as the touchstone of a good design (as long as “good” means “rigid”).

Are real systems really so intolerant to variation that they must be treated delicately? No, of course not. Engineers take great pride in designs that can bend without breaking. Taken at it purest meaning, the TAYF/FAYT mantra should really mean “test as you might have to fly even when things go wrong, and fly with the confidence that your testing has proven the robustness of the system.” At a local level, designs probably meet this to a fair extent.

Something happens though, between local design and integrated systems. There will be interactions not fully understood or accommodated, errors fixed with expedient patches that compromise integrity, resource limitations that couple otherwise unrelated functions and compromise performance and robustness, and so on. The system as a whole will be different than the sum of its parts. However, in becoming this new thing, whether a comfortable understanding of it will also be born is too often left to chance. When this understanding is lacking, reverting to operational inflexibility will be the only way to cope.

3.2.10 Faint Praise for Minimalism

The general pattern to observe in all this is the *very* restrained use of system-level abstraction and design patterns. Designs are hierarchical and modular, but with little exploitation across the system of recurring structure and layered concepts, except in localized contexts. At the higher levels, all we have really done is to name things: power, telecom, data management, command sequencing, attitude control, fault protection, and so on. There is little, if anything, in this association to suggest the fundamental organizing concepts of the design, the properties or principles enforced, or the common structure exploited. In fact, the organization at this level is more likely to reflect the One True Hierarchy of work breakdown and issues of human communication rather than any patterns of design that are intrinsic to the problem itself.

Instead, the architectural resources that software developers have to work with are mostly concerned with minimalist software mechanization (procedures, data, events, timers, messages, files, memory, threads, priority, locks, and the like). Architectural patterns are essentially absent for supporting spacecraft issues (the things systems engineers care about, such as situational awareness, planning and control, resource management, coordination and conflict resolution, health and safety, flight rule and constraint enforcement, end-to-end data management, and so on). This is not to say that such functions are missing, but rather that the available architectural structure serves only as a coarse container for such functions — generally as cut-and-dried services, if they exist at all. In such a context, high-level considerations are routinely upstaged by low level concerns. It is a sort of Chemistry Set approach to architecture. Nothing is beyond reach, but everything is hard or tedious.

This minimalism is not to be condemned out of hand. Ours is a risk-averse culture that moves with deliberation, waiting for the best and safest from the wider software community to filter through its barriers. One can hardly argue with that.

Minimalism is also the product of a flight software culture that spent the first three decades of its four-decade history confined to computing environments so highly constrained as to be unrecognizable in today’s commercial world. In such limited systems, performing on the edge of their capability, the substantial investment required to introduce patterns is met with understandable caution. Therefore, the raw exposure of complexity, out of necessity, has shifted the burden of understanding to the designers and operators of the system. We have come to count on these surgeons of spacecraft software to know what they are doing. Their uncommon skill is valued highly, so we have tended to bestow upon them a fair amount of individual discretion. A weak architecture of loosely connected modules (stovepipes) with little exploitation of recurring structure is consequently well suited to such a culture[8].

There has also been genuine and abiding trepidation over embracing potentially flawed abstractions. Leaky abstractions, after all, are a major source of error in software, when defensive programming is not practiced rigorously. Understandable caution consequently discourages trust in such “helpful” but seemingly opaque or complicated ideas.

[8] Denigration of this as a “cowboy” culture, as sometimes happens, is a disservice to the cautious discipline these practitioners bring to their art.

It must be acknowledged, as well, that minimalism is partly driven by an under-appreciation of abstraction. Abstraction appears primarily in information technologies (including some digital designs), but our spacecraft engineering processes have been driven historically by concrete hardware concerns where hierarchical, functional decomposition dominates. Indeed, software has only recently broken free of its "embedding" in hardware to be viewed as a design peer to hardware, a fact yet barely visible in the upper layers of NASA management. Understandable caution therefore discourages trust in notions increasingly distant from their concrete hardware roots.

Caution *also* argues though that it is time to move on. Continued attachment to minimalist ideas fosters a shortsighted belief that we can understand everything, if we keep it simple enough, and so need not do more — even as the volume of code demanded from each software engineer continues to grow. Thus, we risk being overwhelmed by complexity, as conceptual integrity falters. We cannot keep focusing on arcane details and still shift our attention to greater things. Forced to make a choice, either our ambitions will stall, or details will be neglected with perilous consequences.

Consider by way of illustration the sorts of grave problems in flight software that are generally reported. Do we hear of spacecraft in peril due to complex navigation algorithms, progressive autonomy, or other bogeymen? No. The causes are far more likely to be utterly mundane issues like uninitialized variables, stale data, errors in coordinates or physical units, deadly embraces, priority inversions, race conditions, parameter inconsistencies, memory leaks, variable overflows, phasing errors, addressing errors, and the like. As an excuse, one might argue that, in attempting functions that are more complex, we present a broader cross-section of exposure to such elementary problems, or divert attention from their discovery. However, that premise does not explain common experience, where many errors occur in the most basic and routine functions. Rather, the problem appears far more to be a consequence of uncontained detail. **In demanding exposure of every detail, we leave ourselves vulnerable to every detail.** Consequently, given the absence of abstractions and patterns that obviate such basic errors when one builds upon a minimalist base, the persistence of basic problems is not surprising.

3.2.11 The Systems–Software Gap

Another common cause of serious problems is poor requirement specification. One can generally trace this to a mismatch between the natural expression of systems engineering concerns and the manner in which these concerns must be couched for software implementation. In the absence of system-level abstractions and patterns in the software that relate to the issues systems engineers care about, either requirements are warped into an attempt at specifying software implementation, or implementers attempt their own strained translations. A common example of the former is an interface requirement that defines data content, format, and protocol without a word about the behavior behind the interface. As for the latter, when one considers the fact that lines of code tend to outnumber requirements by several orders of magnitude, there is little doubt that most software answers to requirements in only a weak manner.

So poorly are systems and software modes of communication matched, that a great deal of emphasis is generally placed on documenting so-called "rationale" for each requirement in the vain hope that this will somehow make up for a requirement statement that is itself broken. Nonetheless, even to the extent rationale of this sort might help, they tend to be unevenly available, occasionally in conflict with the requirement to be explained, and in any event subject to the same conceptual limitations as requirements to bridge the gulf between the domains.

Moreover, the *proper* use of rationale is not to explain what requirements mean, but rather to *justify* them in the context of the overall design. In the eagerness to explain poorly written requirements, justifications go wanting. Thus, a faulty attempt to address one weakness exacerbates another. One might expect though, even where justification is given, that it will devolve to a functional decomposition at the system level, since little other structure typically exists at this level.

Whether software engineering error or systems engineering shortfall, it is ironic that the two engineering disciplines with the most to say about correct system behavior have so much to correct in the way they approach and share system understanding. There is also no doubt that such problems represent a serious and growing risk — one we should collectively want to do something about.

3.3 Dis-Integration

We also suffer broadly from the consequences of stovepiped development, which caters more to the personal responsibility of programmers than to the overall conceptual integrity of the integrated product. As a result, many key issues are managed in an *ad hoc* manner, so…

- Formal requirements tend to be shallow or incomplete, interface types proliferate and go underspecified, and opportunities for error multiply.
- Separation of concerns via functional decomposition is often a poorly disguised fiction, as functions are duplicated to satisfy local needs.
- Products become an accretion of stopgap measures and piecemeal improvements, leading to inflexibility and brittleness for both design and operations.
- Systems engineers lose track of the big picture, so the system that ultimately emerges from its subsystems is a surprise.
- Essential communication among developers is thwarted.
- Reuse is hampered.
- Growth and improvement become slow and expensive.
- Analysis and validation become increasingly difficult, and thus less effective.
- Management insight is reduced to superficial oversight, so processes shift to policing of methodology, etc.

In short, there is a wide-ranging failure to adequately address system-level complexity.

Every place you find a conceptual deficit or mismatch, you will likely find data loss or corruption, and people filling the gap to manage incidental or unhandled complexity. This is expensive and risky. Yet skill at absorbing and coping with these arcana has too often become the standard measure by which talent and dedication are assessed, while advocates of advancement in complexity management are marginalized as “not understanding” what the business is all about. This needs to change.

4 The Need for System Architecture

What is generally missing in present systems is system architecture that goes beyond the boxes and lines of functional decomposition; an architecture that goes beyond functional requirements to identify essential properties of interest from all points of view, and principles for achieving them; an architecture that has more to say about system integration besides middleware mechanisms; an architecture that describes systems as more than just interacting applications in some loose chain-of-command hierarchy; an architecture that enhances communication between its developers and the engineers responsible for system specification and operation; an architecture that explains, not just *what* gets built, but *why* it gets built the way it does.

True system architecture brings with it a principled theory of design for each of the crosscutting domains of interest in the system. It expresses design in workable abstractions and patterns of interaction that openly reflect the fundamental concepts of each domain. It guides the design, helping to uphold principles throughout the development lifecycle. True architecture — Architecture with a capital ‘A’ — is required in order to manage system complexity.

In a sense then, one can think of Architecture as Laws of Design, analogous to the Laws of Nature: rules of order that underlie the complexities of a design, while themselves remaining steadfast. Therefore, like the scientific pursuit of Laws of Nature, one must view Architecture as something to be developed, nurtured, and stabilized over time, building on a growing body of experience to refine and extend concepts, as necessary, in pursuit of ever-increasing elegance.

This view of Architecture as the **invariants of design** strikes a compelling and useful counterpoint to the dynamic nature of the software life cycle. Recall that the very *role* of software is to address the growing and changing aspects of a system. During development, continual churn of added and refined requirements is *expected*. The notion that software requirements might somehow be nailed down early, after which one need merely translate them into code, once and for all, is a futile violation of software’s *raison d'être*. Conversely, a freewheeling approach to software that keeps all options open deprives software of the skeletal structure needed to maintain integrity across the life cycle.

For a manageable and affordable software development, one must steer the middle course, embracing both the need for a strong Architecture that provides shape and stability throughout development, as well as a strong Architecture that provides the expressiveness necessary to articulate the inevitable diversity and variability of a responsive, evolving software design. Holding these diametric ideas of fluidity and invariance in mind at the same time is the essence of Architecture.

With such an Architecture in place, one can step early and confidently into the software development for a particular project, getting to a working version quickly, and then iteratively refining it as system understanding matures, without fear of losing control. This permits early and continual exploration of system behavior and customer expectations, which is fruitful, not just for software, but for the system overall.

A system Architecture for software also provides the only meaningful basis for effective reuse in software. Software is not like hardware, where inheritance from project to project can amortize design, manufacturing capability, and qualification costs across many production copies. In fact, software generally provides the essential design flexibility required to *enable* hardware reproduction; so almost by necessity, software is generally not inherited verbatim. However, understanding a software component well enough to change it with confidence in a new application is difficult and expensive. A common observation is that modified code re-incurs most of the original development cost, and more careful attempts at tailored reuse through parameterization of application level components have at best dubious claims of success. Thus, straight reuse is seldom an effective inheritance model for software, except perhaps for lower level elements tied to specific reusable hardware (where the software need not vary as long as the hardware does not vary) or to broadly applicable libraries (which generally embody universal lower level abstractions, such as in mathematics).

In seeking a basis for software reuse, one looks therefore, not to all software design, but rather to the invariants of software design. For this to be effective though, fundamental invariants must be carefully teased apart from the rest of the design, so they will remain stable across applications, between systems, and through time. Thus, invariants being the bases for true Architecture, one gets the greatest benefit in software, project to project, from *Architecture* reuse.

In this realization also comes the natural corollary that Architecture does not generally consist of a set of reusable components copied intact from one application to the next. The invariants of a strong Architecture are its concepts, as represented in abstractions, patterns of design, and so on. This is why software Architectures are so often cast in the form of so-called frameworks. Frameworks are skeletal structures of incomplete applications, where points of variation are carefully circumscribed, while the conceptual structures of the program and the principles behind them are rendered in well tested, long lived, and stable framework elements (typically hierarchies of class definitions and structural patterns in what are commonly referred to as object-oriented designs).[9]

In order to outline an Architectural approach to the problem of system complexity, therefore, we must start with the following basic question. What is there about issues pervasive at the system level that is invariant? In the answer to this question, we will find our Architecture.

4.1 Promising System Invariants

The way forward is now clear. Recall the elements of pattern — recurring structure, layered descriptions, and separation of concerns. Thus, we want the source of Architectural invariants to be the recurring structure shared across applications; we want these invariants to provide a conceptually stable foundation over which more specialized notions can be confidently layered; and we want each invariant carefully separated from others so that each is a simple uncluttered idea.

Recall as well that we are seeking, not to describe such patterns as might exist within our software, but rather to discover those patterns that explain the complexities of the problem to be solved, which we may then capture within our software. The purpose of software, after all, is to

[9] In this light one might generously view a parameterized component as a framework element, but this is merely the weakest form of such variability.

solve a complex problem, not to be the problem itself. Therefore, all preconceived notions of software construction must be set aside. We thereby hope to introduce as little incidental complexity beyond the target domain as possible, so that the software itself is not a further source of complexity.

Recall finally, that the role of robotic flight software, as described earlier, is the flexible manipulation of information in order to control the system and communicate with other systems. You can control and communicate without software, of course, and most systems will have certain functions involving control and communication that (often purposefully) bypass software. Nonetheless, where there is software, it will be mainly concerned with control and communication, and most control and communication will be accomplished by software.

Now, if that were all here were to it, we would have accomplished very little. The recurring pattern: bits come in, bits go out, somewhere in the middle we try to do something smart, and everything else is layered on that — one just has to sort out the separate pieces. To come to grips with the problem though, we need to start finding more structure than that, and to do that it will help to clear up a few subtleties regarding the meaning of the words “control,” “communication”, and “system”.

We can begin with the word “control” and for the time being focus on control in a single, isolated context. Once this idea has been polished a bit, we can add the possibility of multiple control contexts, at which point it will be more productive to discuss “communication” and “system.”

In an engineering context, the verb “control” generally means to deliberately determine, influence, or oversee the behavior of something. That seems clear, but there are a number of important concepts here:

- First and foremost is the concept of **behavior**. Something that is forever unchanging can hardly be said to possess a behavior or to be controllable. Therefore, behavior requires change, and change requires time. The behavior of a thing is what determines the manner, and perhaps likelihood, in which changes to that thing can occur. Control is therefore the business of manipulating how things change.

  For the time being, we can call the thing that changes a “system,” recognizing that we must return to this word later for more clarity.

- We commonly refer to the condition of the changeable aspects of a system as its **state**, taken in a classical sense, where a system manifests a unique state at each moment in time. Behaviors then are the rules that determine what histories of state over time are possible.

  Of course, when we refer to such state in an engineering context, we generally mean some macro-state, which ignores details, preserving only those features we care about, and *know* that we should care about. This results in approximations, randomness, or other variations, which must always be figured as an intrinsic aspect of state. Behaviors are similarly uncertain, perhaps requiring statistical methods of description.

We can generally assume that behavioral effects propagate only forward in time. What the state actually is (or was) at any moment may not be readily apparent, but we can assume with confidence that its past history is immutable, even though our knowledge of it might evolve.

- When we describe a control function, we do not usually mean this as a description of the intrinsic behavior of something. For instance, although one might casually say that the orbit of a spacecraft is “controlled” by gravitational dynamics, one does not generally attribute this act of falling to some control function. Falling is just what spacecraft do, intrinsically, when left alone. However, if one were to observe a healthy spacecraft deviating from a free fall trajectory, one might immediately suspect that there are navigators in the neighborhood exercising their control by some means. Thus, when we say “control,” we generally mean this in the sense of one entity intentionally manipulating the behavior of another; and when we say things are out of control, we generally mean that one entity is prone to changing in undesirable ways from another entity’s point of view.

  Thus, control requires separation: two distinct entities are always involved. And control carries with it the clear sense of a directed relationship: one entity being controlled, and the other entity exercising the control. These will generally be referred to henceforth as the **system under control**[10] and the **control system**, respectively.

  It would generally make little sense to engineer a system where each of two entities tries to control the other. Of course, there may be looping arrangements where this is possible. For example, a flight computer may control power switches, including one that controls power to the flight computer. However, such arrangements are generally problematic and are to be avoided, where possible. Where not possible, assurances are usually necessary to obviate strange or dangerous behavior — often very tricky to do. Self-reference is generally not a good thing in engineering, at least for now.[11]

  Not to be confused with such self-referential loops are loops of interaction within the behavior of the system under control. While problematic and quite common, these interaction loops at least present no self-reference issues. Nonetheless, dealing with such effect loops in the system under control is standard fare for control systems. We will see shortly that the topology of such interconnections in general can be quite influential on control system design. Paying close attention to this topology can therefore be helpful in deciding how to accommodate it.

- Control does not typically mean undirected or purposeless influence either. Two instruments arranged side-by-side on a platform may be thermally coupled, but we do not generally say that the transfer of heat between them is an act of control. Rather, some notion of intentionality needs to be involved in the influence before we are willing to call it control. The presence of intent requires that there be some objective or purpose for a controller. Indeed, from an engineering point of view, it is hard to imagine why one would put a control function in place without an explicit purpose or a means of establishing **objectives**.

[10] The term “plant” is sometimes used instead of “system under control”, but this is not a particularly descriptive word in this context, and is not in common use in either systems or software engineering. Moreover, it tends to suggest only manufactured items, even though their environment may play a significant role and should be included.

[11] Self-reference is key to complex natural phenomena, such as procreation and consciousness. However, engineered systems have yet to achieve mastery of the emergent properties intrinsic to such high accomplishments. Self control is therefore not something considered here.

Moreover, in a properly engineered system, one presumably can tell whether or not the control function is meeting its intent. Otherwise, there would be no basis for deciding either that a control function is correct, or needs to be altered or replaced. There are no half steps between “fix” and “good enough.” Therefore, expressions of intent should be clear-cut. This is not to say that additional criteria might not apply in order to choose among alternative actions that all meet the objectives. Such tuning, or even optimization, generally appeals to some quantitative measure of goodness (i.e., an objective function) that can be evaluated for each option. While certainly an essential aspect of control, it is nonetheless always subordinate to success by any means. Subsequent discussion here is consequently focused on the discrete success-versus-failure issues of control, with recognition that there is more to the story than that.

Note also that the objectives of the control system are about the behavior of the system under control, *not* about the behavior of the control system. Equally, failure is ultimately a consequence, not to the control system, but to the system under control. It is the state of the system under control, after all, that is our ultimate concern. Thus, control objectives need to be unambiguous criteria for success or failure expressed in terms of the state of the system under control.

- Presumably, control would not be needed if the behavior of the system under control always met the intent without external influence. If control is needed, control needs to be able to **command** the required changes to the system under control. This is not accomplished by wishful thinking, obviously. There must be an actual connection of some sort between the control system and the system under control that enables this action.

  It is best to think of the influence of a command as purely immediate, though the system can “remember” the influence, such that its effects can be longer lasting. This leaves issues of behavior over time squarely in the realm of state. Commands do not have behavior.

- Not just any influence will accomplish the intent. Appropriate commands can be devised only if apposite traits of the behavior of the controlled entity are available for consideration, including *at least* some expectations for the intrinsic behavior of the system under control and how the commands to be exercised will affect it. Whether such **behavior models** and **command models** remain strictly within the purview of the control function designer, or somehow find their way in a more explicit manner into the controller implementation, the need for these expectations is undeniable. One cannot control what one does not understand.

- Moreover, when *a priori* expectations alone are not enough, due to unpredictable variations in behavior or uncertain initial conditions, the controller must also have insight into the state of the system under control as time progresses. That is, in such cases there must be a way, not just for commands to flow from the controller to the controlled, but also for status to flow from the controlled to the controller. This status will be in the form of **measurements**, which sample some observation of the state at discrete[12] moments in time. The influence exercised by the controller must therefore be governed by this status, in addition to expectations of behavior.

[12] Analog control systems operating in continuous time are possible, of course. However, software operates only in discrete time and can accommodate only measurements at discrete times.

- Proper interpretation of this status will also require **measurement models**, of course, which provide expectations for how measurement values relate to states of the system under control.
- Where measurements are used, the influence exercised by the controller is chosen moment by moment, according to the status and behavior of the system under control, given present and future objectives. This defines the ever-present **closed loop** — not just a design option, but an inevitable consequence of the fundamental concepts of control.

  A key feature of this idea is that control decisions are continually subject to revision as new measurements become available or as objectives change. As an aid to making good control decisions, planning may be taking place, using models of behavior to anticipate the consequences of current actions, but control decisions nonetheless need appeal to no other information besides measurements, models, and objectives in order to meet objectives, *as long as objectives are achievable*. This remains true, even when objectives include provisions for tuning. Once a control design is in place, it should never need to consult any other information.
- The caveat above regarding achievability has to do with decisions regarding either failed objectives, or conflicting objectives that are not simultaneously achievable. These possibilities must be contemplated in every system, despite the best of efforts. Hardware can fail; the environment can behave strangely; operators can make unreasonable demands; or the odds may simply not fall in one's favor. Whatever the reason, every control system must include in its design the means by which it recognizes and manages conflicting or failed objectives, and the criteria for making appropriate choices about them. A control system without such **failure handling** provisions is not merely incomplete; it is really only half done.

  Decisions about success and failure require the assignment of an order to sets of objectives, trying to do the most stringent set first, then backing off to successively less challenging sets in some preferred order until one is achievable.

The preceding description identifies the most elementary aspects of a feedback control system. Of course, the flow of commands and measurements between the two entities in this loop can be contemplated only because we have taken the trouble to delineate controller from controlled. The demands on the latter are objectives of the former; the behavior of the latter dictates the design of the former, and so on. Therefore, without this explicit partitioning, control becomes a hazy concept.

It is also worth noting that commands and measurements, models, and the control functions they support are all intrinsically informational in nature. Physical phenomena play a role only in that the sources of measurements and the targets of commands may need to be physical. Control systems are intrinsically informational, independent of their physical implementation, as asserted previously. They are good candidates for software implementation.

To be clear though, the line between control system and system under control is not necessarily the line between informational and physical parts of the system. The control interface can be drawn anywhere within the informational part of the system. This raises the possibility that the system under control for one control system might actually contain another control system. Moreover, since the informational system is hosted on a physical implementation, that control system is subject to both informational and physical influence. Therefore, from the point of view of any control system, whether or not the system under its control is informational or physical (or a mixture of both) does not matter; and whether or not the system under its control contains another control system does not matter. The *only* things that matters are the behavior of the system under its control, the control system's objectives against that behavior, and the access the control system has at their interface.

What we hope to accomplish with this expanded view is to begin to see the *fundamental* abstractions and patterns that describe the control problem. These can then become the basis for control software design. In doing so, it is important to keep in mind that the control problem exists *outside* the software, while the control system is to be implemented *within* the software. Therefore, the abstractions and patterns used by the software must represent control issues of the system under control. That is, the system under control exists first; then we create a control system in software to solve its control problems using a set of abstractions and patterns that describe the key elements of control. Here then is the candidate set, so far:

- **Control System —** a system put in place to control the behavior of another system, such that the system under its control meets a set of objectives.
- **System under Control —** an entity distinct from the control system and under its influence (but not vice versa) with an explicitly drawn interface between them.
- **State —** pertinent aspects of the system under control that change over time that the control system designer is aware of.
- **Behavior —** the (typically uncertain) manner in which the states of a system under control change.
- **Command —** a means of affecting the behavior of the system under control that is exercised by the control system via its interface to the system under control.
- **Measurement —** evidence of the state of the system under control that is obtained by the control system via its interface to the system under control.
- **Behavior Model —** expectations regarding the behavior of the system under control that are the basis for the control system design.
- **Command Model —** expectations regarding the change of state in the system under control that will result from a command, given its present state.
- **Measurement Model —** expectations regarding the value of a measurement from the system under control, given its present state.
- **Objective —** the desired behavior of the system under control, expressed in a way that can be evaluated for success or failure.
- **Closed Loop —** control actions to influence the system under control determined by the state and behavior of the system under control and the objectives against it.
- **Failure Handling —** the process and criteria by which competing or failed objectives against the system under control are handled.

The word "system" can now be dealt with in a straightforward manner.

First, it is important to realize that all of the systems we care about are open systems. The pairing of system under control with control system discussed, for instance, serves no purpose if it is closed. Either some aspect of the system under control (including its environment) is visible to an external entity, and the control system serves to adjust behavior to suit external needs, or the system under control is *not* visible, and the control system serves as a means to gain access to the system under control (including its environment) — usually both. In any event, the system under control and control system together comprise an entity within the context of something larger.

In addition, in defining the system under control there was no need to circumscribe it in any particular way. There were aspects of its behavior of interest to some external entity, and there were objectives against that behavior that justified putting a control system in place. Thus, from a control system's point of view, the system under control is only what it needs to see and understand in order to bring about that behavior. The chosen model of behavior, including the states through which it is defined and against which objectives are asserted, determines the scope of the system under control from the control system's point of view.

We see then why the natural temptation to take the spacecraft wrapper around the control system software as the system under control would generally be incorrect. This is a concrete entity, but it can never be the entire system under control, because no model of a spacecraft could ever be complete without reference to the spacecraft's environment. Therefore, the system under control for a spacecraft control system is larger than this, including major gravitational bodies, radiation and field sources, physical obstacles, targets of observations, and so on. The extent of this environmental extension goes as far as it takes to encompass an adequate model for meeting control objectives.

In both regards therefore, it seems that "system" is whatever is engendered by control objectives. To a control system, its system under control is "the system"; while the closed loop of control system and system under control is "the system" to something larger. The only entity in all of this that can be circumscribed in any concrete sense is the control system itself, with its carefully delineated interface to the system under control, and its external interface to whatever sets its objectives. The control system does not need an abstraction of itself though. Only when we introduce meta-control issues or cooperation among distinct peer control systems do we need to invoke an abstraction of a control system.[13]

This is where "communication" comes in, as an interaction among control systems.

Communication has two aspects, one physical, and the other informational. The physical aspect includes telecommunications equipment and other components involved in the physical movement and storage of data. This is just an ordinary part of the system under control, where objectives are driven by such things as the need to establish a suitable physical link, models being extended as appropriate to describe the situation. No new concepts are necessary.

The informational aspect of communication creates a twist, however, which some find non-intuitive. Namely, the state of the system under control includes the data it is responsible for

[13] Meta-control and peer-to-peer control interactions are important topics, but largely beyond the scope of discussion here. The essential principle to note is that such relationships must be delineated as carefully as that between control system and system under control.

storing and communicating. Therefore, intent against the system under control can include objectives on the content of data stores and on modifications to this data through transmission, reception, addition, deletion, compression, filtering, and other informational processes. Engineers are not used to thinking about data collection, processing, and transmission as a control problem — until success criteria are introduced. Then it becomes clear that the objective of such activities lies solely in the content, quality, and location of data, which comprise the *state of data collections* in the system under control. Therefore, this too requires no new concepts. In short, communication lies within the realm of control.

There is one tricky part though. Control system software resides on a physical platform. The state of the control system software (described below as its knowledge and intent) is likely to share the same physical platform as other information in the system under control. If this software state were strictly private, there would be no issues. However, this information competes for space with other data in the system under control; it is vulnerable to phenomena in the system under control; and the control system needs to communicate this information to other systems through the system under control. This puts it in the position of having to control its own state, which is self-referential — an arrangement already described above as problematic.

There is no completely satisfactory solution to this dilemma. In one sense, it can be regarded as a meta-control problem, where some higher entity concerns itself with the state of the control system. Reset recovery is an example of this, where the state of the control system must be restored before it can resume operation. Inevitably though, some part of the self-reference problem is likely to remain, so there must be principles for dealing with it. This specific issue is secondary, however, relative to the key message that such principles are indeed important in general. Many such principles will be cited below for illustration. Nonetheless the list will be left incomplete, unfortunately including the self-reference problem here. These are topics for a broader, more formal exposition.

4.2 Hide and Seek

This review of system control and communication ideas has been so basic, and control and communication are such large parts of spacecraft software, that it is natural to expect these ideas to be patently evident everywhere in our flight software implementations. They are not. Consider:

4.2.1 Control System and System under Control

Flight software is generally delineated by a set of hardware-software interfaces. Some might declare this as the interface between the system under control and the control system, as well. However, there are typically functions close to the hardware devoted to tending interfaces, conditioning and processing data, and similar functions that are generally not viewed as control functions. This suggests drawing the line around control somewhat inside these margins. Where inside, however, is not generally easy to spot in customary implementations, and what these control functions view as their system under control can also be unclear.

For example, a module assigned to manage propulsion states would typically ignore intervening device drivers, busses, and so on, dealing with them in general as an ideal communication link. The propulsion manager might also rely on other related functions, such as sensor hardware management, but treat these as equally ideal. Such relationships are understandable, since intermediate and supporting functions are usually designed to enable idealized abstractions of their behavior. This is a good example of design layering and separation of concerns, as described earlier.

Nonetheless, the control concepts above make no allowance for a third element, whether it be an intermediate function or some other entity. Therefore, from the propulsion manager's point of view, the system elements that these idealizations conceal should be either part of the system under control or peers within a larger control system. Depending on which role they are assigned, control concepts create certain expectations for their makeup.

For instance, the state and behavior of interface elements are clearly of interest to the overall control system, including to propulsion management. Potential issues may include timing jitter, time tagging uncertainty, latency, data corruption, competition for bus access, failure modes, and so on, which can all shape how well propulsion objectives are managed. However, by dealing with interface functions idealistically, control issues such as expected behavior and success criteria (i.e., objectives) stay submerged in the architectural dialog. Users of this capability must either have an ironclad guarantee that the abstraction will never leak, or when the time comes must deal with their non-ideal reality in a non-architectural (i.e., *ad hoc*) manner.

A better approach is to acknowledge such elements as part of the system under control from the outset, and as such, incorporate all significant threats to the functionality they present as part of a more realistic abstraction of their behavior. The control system can then be designed accordingly, this time with all relevant issues covered by the architecture.

Similarly, if managing sensor hardware were taken as a control system function (sensor hardware remaining in the system under control), then it and the propulsion management function would be collaborative elements within one control system, where a principled architecture for control coordination could guide their interaction.

The alternative of pushing sensor management into the propulsion manager's system under control is also plausible, resulting in two layers of control. In fact, this would be done in most systems as part of a functional decomposition hierarchy. However, there are generally dozens, if not hundreds, of such functions in flight software. Such an approach could result in deep layering that is harder to model, raising questions of whether partial ordering among them is always possible (to avoid loops), and opening unsolved questions of how to decide on the module decomposition and how to determine the appropriate relationships among modules. Such issues are generally handled as they arise, rather than methodically according to control principles, and the result can be very hard to understand.

It is generally preferable for improved understandability to flatten the conceptual layers, thinking of all these control functions as cooperating peers within a shared control system. That way, other modules in the software need not be modeled as part of the system under control. As we shall see, when these modules are peers, one can regularize the discourse among them, adding additional structure to the architecture. Moreover, a pattern for the relationships among such modules can be developed that is explained fully by the model of the system under control.

4.2.2 State, Behavior, and Model

If one were to ask how the states of the system under control and their behavior are related to various constructs within typical control system software, the reply would vary quite a bit, depending on which functions are discussed. Even within one module, the picture would likely be quite mixed. Some states will be clear-cut. For example, there will be attitude determination functions producing self-contained, complete, explicit estimates of the attitude state, and there will be clear dependencies of the attitude determination and control functions on attitude behavior (e.g., equations of motion), which will be well documented. On the other hand, the state of a hardware device is likely to be dealt with as reported by status telemetry, occasionally processed though error or transient screens. Usability of the device is likely to be handled elsewhere. Associated control decisions, including fault responses, are likely to be fairly mechanistic (e.g., if device built-in-test indicators are set for three read cycles or more, issue a reset command) and not always found in just one place, and so on. The result is that neither motivating objectives nor resulting behavior are readily apparent from the implementation.

Other variations relate to the way in which uncertainty is handled, the way time dependencies of state are dealt with, the way state is recorded or predicted, how internal consistency is maintained as updates occur, and so on. This does not necessarily mean that systems and software engineers lack understanding of the states and behavior. The problem is that it is hard to tell. Architectures are generally silent on how state and behavior of the system under control are to impress themselves on the control system design. Therefore, these issues do not get uniform treatment. Methods vary, rigor varies, consistency varies, and much energy is devoted to reconciling this variety across the system.

Yet another problem with the piecemeal variability across a system regarding how state and behavior are treated is that there is no home for the consideration of emergent behaviors, either within the system under control, or within the larger system, once the control system is added. Emergence is inevitable in any complex system, but it is a system level phenomenon. Only a concerted effort to understand state and behavior across the system under control and to deal with it cooperatively across the control system has a chance of dealing with emergence systematically.

4.2.3 Objective

As described above, objectives should be unambiguous expressions of the desired behavior of the system under control. This rarely happens in actual flight systems, at least not in a way that is easy to figure out. A pointing command, for example, provides a pointing direction, but not the criteria for how well it must be achieved. This might arrive in a separate deadband command, or it might be implied by the selected control system mode for which a requirement exists somewhere, or it might be in a fault monitor threshold that looks at pointing errors, or it might be in a persistence check that limits the duration of outages in attitude sensor data. The pointing command is also silent on the deadline for its achievement, on the orientation around the pointing direction, on how long to hold the pointing direction, on how to evaluate competing commands for priority or compromise, and so on. In spite of all that, pointing commands are a one of the *better* cases.

Objectives are frequently specified to systems poorly. They often specify control system behavior (e.g., modes) rather than behavior of the system under control. Many are implicit (e.g., limiting power cycles) or never fully explained at all (e.g., safety). The means of supplying objectives to a system (e.g., commands sequences) generally lacks a full expression of intent, such that the system gets no guidance from the objectives on how to recover when things fail. Mechanisms for handling failure (e.g., fault responses) are generally independent of mechanisms for ensuring success (e.g., critical sequences). Overall, there is very little about the way objectives are handled in flight systems that appeals overtly to the fundamentals of control. Only when the notions of objectives and failure are definitively linked and unambiguous will that be possible.

4.2.4 Commands and Measurements

Two ubiquitous concepts in the spacecraft world are so-called command and telemetry. The typical definition of “command” is data sent to make something happen, while “telemetry” means data received from something that indicates what happened. Unlike the careful definitions described in the control fundamentals above, where commands and measurements are modelable entities exclusively reserved to the interface between system under control and control system, the sources and targets of *generic* commands and telemetry can be just about anything, and their content is largely *ad hoc*. As an illustrate of this hodgepodge, one might have one command to open a valve, another to set a pointing direction, another to update a software parameter, another to set a clock, another to relay trajectory information from the ground, another to disable a fault monitor, another to read out memory, another to set the attitude control mode, another to initiate a calibration, and another to stop a command sequence. Various categories have developed over time (sequence commands, hardware commands, etc.), but there is no particular rhyme or reason to what can be in a command.

Telemetry is the same. Items could be raw scientific observations from an instrument, or interpreted measurements from an engineering sensor, or modes, flags, and counters from software, or a stream of software or hardware events, or estimates of attitude, or a log of fault responses, or a memory readout. Anything is fair game.

These observations should not be taken as criticism that something unnecessary is being done. Rather, from a control point of view, it is hard to see how most of these examples fit into the basic control picture. Which commands carry influence from a control system to a system under control? Which commands establish objectives, and which do not? What telemetry represents measurements flowing from a system under control to a control system? How do these relate to the state and behavior of the system under control? What is the role of telemetry in reporting success or failure?

There are no clear answers to such questions in conventional architectures, because generic commands and telemetry, despite the control-oriented role these words suggest, are conceptually little more than data, perhaps with side effects. On the other hand, the words “command” and “measurement”, *as defined above for control*, describe specific, modelable roles: one being a directive to alter the state of the system under control, and the other being evidence of the state of the system under control. These concepts can be made even more explicit, with a little thought, as we will see below. In doing so, it becomes possible to apply principles of use that cannot be applied to their more amorphous counterparts.

4.2.5 Close Loop

Commands and telemetry are only two forms of data in a typical flight software control system without consistent and clear mappings to control concepts. As we have seen, objectives can also be unclear or missing; and assumed states and behaviors can be ambiguous or merely implicit. Given the confusion of different sorts of information, it is therefore not surprising to find that control decisions also depend on a variety of items, many of which are difficult to map to the state and behavior of the system under control or to control objectives against it. These include modes, flags, counters, and so on, as described previously. As a result, many control functions are merely unprincipled gadgets. “Objective”, as well, can be given a more specific meaning, including the notions of success and failure that will be the basis for a far more principled approach to closing loops.

4.2.6 Failure Handling

The same applies to decisions regarding failures, when objectives become unachievable. Without appeal to the fundamental concepts of control and the principles behind them, control system behavior and the behaviors they induce in the system overall can never be mastered. Yet in most systems, failure-handling design is typically disjoint, often without direct ties to control objectives and nominal control processes, and rarely imposed upon a system until after “nominal” control functions are in place. Fault management systems generally respond to problematic “errors”, as undesirable deviations, but there is little discrimination between deviations from modeled behavior, deviations from predictions, deviations from objectives, or deviations from “nominal” or “safe” conditions (neither of which is well defined). Similarly, it is not clear, when a threshold is tripped, whether this reflects an assessment of system state (e.g., a device has failed), an objective violation (e.g., the device cannot perform some required function), or a control decision (e.g., something must be done about the failure). In conventional designs, it could be any or all of these, conflated and demoted to an inscrutable act of arithmetic.

A word about communication is also in order here. A great deal of effort goes into deciding what goes back and forth across space links. Mostly, it is generic commands and telemetry, with all the lack of specificity these ideas bring to the discussion, as described above. When these notions are replaced by control system concepts (commands, measurements, objectives, and successes or failures…), communication with a control system becomes more structured and modelable. Moreover, a clean separation can be made between communication at the level of the control system and communication in the system under control, below, or in meta-control functions, above. For example, science observations are destined for communication at the level of the system under control; control objectives are communicated at the level of the control system, such as between peers; and software updates are communicated at the level of the meta-control system. Such partitioning is essential in avoiding dangerous tangles among the levels.

4.3 Transparency

The fundamental concepts of control identified so far (with communication as a key aspect) have revealed a number of shortfalls in the way typical flight software systems deal with control issues. To reiterate, although these systems obviously perform a control function, it is often hard to tell how implementation choices map to fundamental control concepts. If these relationships are not clear, then the application of control principles, best defined in term of these concepts, will be a haphazard affair. It does little good to appeal to specifications, comments in code, operator’s manuals, or other documentation to provide the explanations for what was intended in the design. The opaque muddling of control ideas is likely to be just as pervasive in these artifacts.

What we need is full transparency, such that the fundamental concepts of control are explicit and it is clear that their implementation is faithful to control principles; and of course, the proper approach to establishing such rules of design is through Architecture. That Architecture can then guide, not only the flight software design, but also the systems engineering processes that provide requirements, models, objectives, validation, and so on to that design. The Architecture then becomes the *lingua franca* through which systems and software engineers communicate in meaningful unambiguous terms. The regularity of an Architecture that appeals broadly to the concepts and principles of a design is also a boon to management through the transparent insight it provides throughout development. Architecture is thus the essential element for coherence and stability across the whole enterprise.

So far, though, the control concepts described above are still somewhat indefinite, so not in a form that is readily translatable into the software abstractions and patterns of an Architecture. In particular, little has been mentioned about the nature of the process that lies inside a control system, amidst objectives, measurements, models, and commands. In order to give this function and its environs a more definitive shape, there is another concept of control that needs to be introduced. It is not quite as fundamental as the rest, but it is nonetheless a familiar aspect of all sufficiently complex control systems — even if not explicitly so. This is the concept of **cognizance**.

Through its knowledge of state, behavior, and objectives, a control system is in some meaningful way cognizant of the system it controls. Such awareness and knowledge may not appear overtly in a design, but it is present in the sense that, were state or objective to change, the control system would respond to it, and were behavior to change, control system plans or designs would have to be revisited.

In its cognizance role, a closed loop control system solves the major problems of *how* to achieve an objective, while hiding these details from external entities. However, it leaves to these external entities the vital problems of understanding *what* objectives mean exactly, *what* objectives are actually plausible, and *what* to expect when objectives fail.

5 A Cognizant Approach

In the interest of transparency then, what we need is a more explicit meaning for statements regarding the use of state, objectives, and models in the control system. This occurs through three steps: 1) the introduction of explicit state knowledge, 2) the expression and manipulation of objectives in terms of state constraints, and 3) the invocation of state-based models in the analysis and achievement of these objectives. This architectural triad forms the base of our "out-of-the-box" approach to complexity.

5.1 State-Based Architecture

The first step in endowing a control system with cognizance is to overtly express within it a representation of the state of the system under control, to the extent the control system is able to construct it from the available evidence.

A control system with explicit state knowledge is able to respond to objectives expressed directly in terms of the state of the system under control. We refer to such an architecture as a **state-based architecture** whenever knowledge and objectives of state are explicitly expressed in a principled fashion for all control functions.

All this means is that a temperature controller will have within it a principled knowledge representation of the temperature it is controlling; an attitude controller will have within it a principled knowledge representation of the attitude it is controlling, and so on.

Likewise, objectives on temperature will be expressed in terms of temperature; objectives on attitude will be expressed in terms of attitude, and so on. It is far better to give a control system such direct objectives than to have to issue commands to heaters, thrusters, mode managers, and so on to make the same things happen.

> **Common Points of Confusion**
>
> Whenever a control system possesses state knowledge, that knowledge becomes part of the state of the control system, in that this knowledge is something stored and manipulated within the control system over time. It is essential to note, however, that control system state knowledge never means knowledge of the state of the control system. Rather, it means the control system's knowledge of the state of the system it controls.
>
> Furthermore, the existence of state in a control system (e.g., in state machines) does not make it a state-based architecture, as defined here. In fact, it can be argued persuasively that, aside from state knowledge and knowledge of its own objectives, a control system should have no other state of its own — no modes, no counters, no flags, etc. Practical issues occasionally work against that ideal, but it is something to keep in mind as a control system comes together. Every bit of state in a control system, beyond the basic need for state knowledge and objectives, is a source of incidental complexity that should be managed carefully, and eliminated, if possible.

State-based control hardly seems like a noteworthy idea, considering that this is a relatively common feature in spacecraft control designs — at least on the surface. However, it is easy to find many exceptions. Moreover, even in unexceptional cases, principled adherence to this concept is generally incomplete or implicit.

Consider, for example, early attitude control systems in which flight control systems merely attempted to null sensor signals. Nulling a sun sensor signal had the effect of pointing the sensor at the Sun; other pointing objectives were accomplished by injecting a bias voltage into the sun sensor signal. The control system might be said to have had cognizance of the sun sensor signal, but certainly not of spacecraft attitude. That cognizance remained with ground operators, who *did* need to know the attitude. However, since the flight control system was not cognizant of attitude, the job of these ground operators was more complex, due to all the incidental details they needed to consider.

Just as importantly though, and for the same reason, the job of other flight functions was *also* more complex, wherever there was an interest in spacecraft attitude. Thus, lack of transparency did not merely move complexity from one place to another. It actually added incidental complexity across the system.

Such things are far less common in attitude control these days, but one still finds other examples where knowledge of the state of the system under control is the demesne of some entity *outside* the control system assigned to its oversight. Objectives regarding such states are consequently not expressed in terms of state either.

Cognizant control systems, on the other hand, have knowledge of the states under their control. Moreover, in keeping with our aims here, they possess this state knowledge in a principled manner. Here is a brief sampling of some of these principles:

- Since state exists continuously in time, state knowledge should be represented as a continuous, uninterrupted *history*[14] of state values.
- Physical units, reference frames, and other semantic information required to interpret state knowledge should always be unambiguous.
- Recognizing that all state knowledge is necessarily imperfect, principled representations of state knowledge should include explicit measures of uncertainty, as well as accommodations for staleness and for extrapolation or interpolation in time.
- State knowledge should be impartial, bending to no other need than to honestly interpret all available evidence.
- There should be no redundancy (or other potential for inconsistency) in state knowledge within a control system.

These and other principles are the sorts of things one can begin to capture and enforce in abstractions and patterns within an Architecture. We will return to a few of these later in more detail.

Having become explicit regarding such principles, violations become apparent, even in seemingly well-engineered systems. For example, even in the fastest flight computers, a state estimate (i.e., state knowledge) is typically produced only at discrete moments in time, to be used later in control decisions, and then replaced in the subsequent cycle. How does this violate control principles? Well, with few exceptions, an estimate is assumed to be present before it is needed. It is assumed to be up-to-date when used. It is treated as unchanging over an update interval. And its lifetime is assumed to be little longer than one update interval. Yet little of this is typically made explicit in the implementation, and it is unevenly considered in the

[14] The term "history" is used loosely here, since we require it to include the future.

documentation. Trouble with timing variations, missed cycles, incomplete initializations, stale data, and other problems are common, exposing such assumptions as serious vulnerabilities in the design.

One could treat such issues as just a normal part of software development — bugs to be exposed and repaired. However, they need not have been there in the first place, if principles of state knowledge in control had been enforced. Moreover, one can never be sure that all such vulnerabilities will be found through *ad hoc* measures. Multiply this exposure by hundreds or thousands for all the states of a system, multiply again for all the other control principles threatened by weak architecture, and one has a complexity nightmare.

As we shall see, there are a remarkable number of such principles, when one takes care to enumerate them, even though there are only a few fundamental control concepts, which one might otherwise be inclined to take too lightly. The value of making these few concepts transparent in an Architecture is that the principles become much easier to apply and validate, and with the application of fundamental principles comes design integrity and managed complexity. Then again, concepts that are out of sight lead to principles that are out of mind. Many serious flight system software anomalies can be traced to such neglect.

Let us consider a few more principles, continuing with state knowledge.

As described above, the generation of state knowledge should be impartial. However, clean separation of state determination from state control is essential, if state knowledge is to remain impartial. As described previously in the discussion of intertwined control and estimation, the temptation to entangle these responsibilities, when they are not acknowledged as separate, can seriously compromise the integrity of state knowledge. This suggests a logical division of control system functions into two distinct parts, one part to produce state knowledge, and the other to use this knowledge for control.

The production of state knowledge is commonly referred to either as **state determination** or estimation. This includes obvious examples, such as attitude determination or temperature estimation, but other activities (e.g., calibration, consumable tracking, fault diagnosis…) fall under this definition, as well. Making control decisions based on this state knowledge is referred to, naturally enough, as **state control**.

A further benefit of this division is that state knowledge must appear overtly at the interface between the two parts. That is, not only is there a representation of state knowledge in the control system, but it appears explicitly as an architectural entity, as well, not hidden within some other module. In order for this information to be persistent, however, since it will be subject to continual consultation and revision, state knowledge must be viewed as more than fleeting interface data. Rather state knowledge must have an independent, enduring existence. We commonly refer to such a persistent representation of state knowledge for a particular state as a **state variable**.

The explicit existence of state variables makes state knowledge easier to inspect for adherence to principles, easier to capture within architectural framework software, easier to share across other control functions, easier to avoid divergence of opinion across the system, and easier to understand the reason behind control decisions. Moreover, state knowledge becomes a natural candidate for communication to an external source of objectives, which will learn not only whether objectives have been met, but also how well and in what manner — all of which contributes to transparency.

The provision of state knowledge is not nearly as elementary as typical implementations would lead one to believe. For instance, since state spans time, knowledge of it should also span time in a manner consistent with behavior models. However, in most implementations this is accomplished instead merely by interpolation or extrapolation, and then often implicitly and even carelessly. Extrapolation, for instance, often amounts to simply using old data, assuming it will be used or replaced soon enough. Indeed, for most state information (and the evidence used to derive it), little if any thought is typically given to this.

In this light, it is clear that discrete events (measurements, detections…) are not state knowledge. They are merely evidence, which should contribute to state knowledge only through the application of behavior models, as described below. Even a simple measurement, invariably used only after some delay (small as that may be), requires a model-mediated extrapolation from event to knowledge in order to presume its usability over time. Accordingly, the supposition of usability after a delay must be viewed as a simple model of behavior (whether or not intended as such) regarding both the changeability of the states being measured and latency in the control system's use of the measurement. Such models are usually implicit in standard practice, but models they are nonetheless, since they describe how knowledge of state is to be carried forward in time. Therefore, they are in potential disagreement with other models, and could be flat out wrong, as recurrent problems with stale measurement data indicate. Avoiding this requires transparency.

Besides corruption from models that disagree, state knowledge may also appear in multiple versions within a system, and in fact often does (e.g., from different measurement sources). This can happen when further modularization (beyond the separation of state control from state determination) is accomplished by partitioning state determination into separate modules. Unprincipled modularization that results in multiple versions of state knowledge is problematic, because of the potential for control decisions working at cross-purposes due to different views of system state.

There is more to this problem than just conflicting state knowledge though. Separate opinions can never be as good as a shared opinion gained by considering all information sources together. Thus, there being no point in making matters worse than necessary, it is useful in any system to strive for a single source of truth for each item of state knowledge — one that is the broker for *all* evidence regarding that state. In a state-based architecture, therefore, all state variables address distinct elements of state within the system under control, and each state determination module is given sole responsibility for some subset of these state variables.

In keeping with this principle, it is also necessary to exercise discipline not to copy and store state knowledge information around the system, but rather, to always return to its source state variable in the control system on each use, if feasible — or understand the consequences, if not.

Moreover, no state determination module should ever form a private opinion about some state, when that state knowledge can be obtained from a state variable managed by another module. For example, before accepting a measurement, a state determination module may perform some test of the measurement's voracity (e.g., a range check). Perhaps surprisingly, unless that module is also responsible for estimating the health of the measurement source, this test would be improper, because the test amounts to a health estimate. Instead, this module should be consulting health state for the sensor from an external state variable.[15] Otherwise, the system is prone to inconsistent actions regarding the sensor's health. Similar reasoning applies to calibration-derived states, and so on. It is only through rigorous application of principles enabled by transparent concepts that such subtle errors can be caught.

Note that state determination modules have been treated above as peers within the state determination part of a single control system, interconnected via the exchange of state knowledge through state variables, which are also peers. Since the task of state determination overall is to provide a single consistent view of the state of the system under control, layered abstractions must be avoided unless they are pure distillations of available state knowledge. Thus, no state determination module need be defined in terms of others, no state determination module need contain another, and derived state information can be developed by other peers. The resulting overall state determination composition is flat, which is a desirable architectural feature, as explained previously. As we shall see below, there are additional principles that can guide modularization of a state-based architecture.

As described earlier, another key aspect of state knowledge is that it is always imperfect or incomplete, there being no direct, continuous way to access the actual state itself. Such knowledge can be determined only indirectly through limited models and the collection and interpretation of ambiguous or noisy evidence from the system. Thus, while a system is truly in a unique state at any instant, a control system's knowledge of this state must necessarily allow for a range of possibilities that are more or less credible depending on their ability to explain available evidence. This is what representations of uncertainty try to express.

A particularly troublesome aspect of this is that knowledge uncertainty across state variables can be correlated. Thus, while states themselves may be separable, their knowledge representations may not always be. Examples appear commonly in navigation and attitude control, where for instance sensor bias knowledge can be strongly correlated with position or orientation knowledge, respectively. Indeed, position can be correlated with orientation — generally an unhappy situation. These are by no means the only cases though — another common case arising when symptoms of an anomaly are attributable to multiple causes. In all such cases, where correlation is a significant factor in interpreting evidence or making control decisions, a shared representation of uncertainty across state variables becomes necessary. Control decisions can then be tailored to accommodate the ambiguity, including taking measures to reduce the ambiguity, where warranted.

[15] Alternatively, health state information may be extracted from the measurement by the health estimator, which passes along a compensated measurement with its own measurement model — one independent of health. Contrast this with the ill-advised "virtual device" approach, requiring the inversion of generally non-invertible measurement models.

Even this is not quite the full story though, regarding uncertainty. Among the possibilities represented in state knowledge must generally be an allowance for behavior outside of understood alternatives. That is, a control system should be able to recognize when the system it controls is in a state for which confident expectations of behavior are apparently lacking. This creates a subtle but important difference between saying "the state of the system is unknown" and saying the "state of the system is unidentified". By representing such modeling issues in state knowledge, appropriately cautious control decisions are possible.

Doubt, ambiguity, and misunderstanding all create a scene of confusion for control systems. When confronted with multiple possibilities, control decisions become more complicated. However, to edit the possibilities before presenting a selected one for control action (as when hopelessly ambiguous readings are relegated to true or false by a threshold test) amounts to making precisely this sort of decision anyway. This is one of many insidious ways that abound in systems to muddle the boundary between state determination and control, and hence the location of state knowledge (more on this in the next subsection). In doing so, transparency is compromised and the system is a step further on the slippery slope to lost architectural integrity. It is far better to acknowledge knowledge uncertainty in all its guises, represent it honestly, and make control decisions accordingly with proper consideration of the risks involved.

Part of representing knowledge uncertainty honestly is also acknowledging that this uncertainty usually degrades further with time unless knowledge is refreshed. This too should be overtly dealt with. For example, beyond some point in time, knowledge may need to be declared entirely invalid unless updated with new evidence. To keep the line between knowledge and control clean, this must be the responsibility of the producer of the data, not its consumer. Following this principle eliminates the problem of uninitialized or stale data.

Yet another concern is the choice of state representations — transparent choices being easiest to use correctly. With this in mind, certain obvious violations come to mind, such as the absence of explicitly defined units or frames of reference for physical values, which has already been cited.

Other recurring offenders, appearing frequently in fault management functions, serve as additional examples. For instance, consider persistence counters, the most basic of which tally consecutive instances of an error. If such a counter is necessary, then presumably lone errors are insignificant — a statistical improbability under normal conditions perhaps. And if that is the idea behind persistence, then it is apparently a measure of likelihood that the system is in an abnormal state. This is actually a good start, since uncertainty in state knowledge has entered the picture, as it ought. Moreover, before the persistence threshold is reached, one often finds systems rightfully discarding suspect data, indicating that the diagnosis of a potential fault has already been made, if not overtly. Yet, upon asking how a persistence threshold is selected, it quickly becomes clear in many systems that any of a variety of criteria might apply: allowance for an occasional operational fluke; the level of system error tolerance; a delay to control precedence or timing of fault responses; an empirical value that avoids false alarms — often multiple reasons — depending on the motive for its last adjustment. Thus, poor overloaded persistence actually represents almost anything *but* likelihood, as commonly used, and its threshold has as much to do with making and coordinating control decisions as with making a fault diagnosis. As a result, persistence thresholds join the ranks of a great many parameters in typical fault management systems as tuning knobs, not directly relatable to states, models, or objectives, and consequently beyond help from any of these fundamentals.

Error monitors in general tend to have analogous problems when errors are not interpreted through models or correlated and reconciled with other evidence. When such a diagnostic layer is absent, and responses are triggered directly by monitor events, it becomes hard to put one's finger on what exactly a system believes it is responding to. This loss of cognizance is an invitation for all kinds of interesting emergent behavior.

Similar issues occur when poor models of behavior are used. As described below, for example, timers in general have little if anything to say about the nature or conditionality of behavior. Do such timers represent objectives, state knowledge, measurements, models, or what? From the gyrations developers and operators go through to manage them, and the inevitable mistakes made due to their typically opaque, raw character, it is clear that no one quite knows how such appliances fit into the control framework, or how they relate to other elements of the system.

There are many other problematic ways to represent state knowledge in a system. Others are described in the next subsection on control. A final one worth mentioning here though, is the disable flag for a fault monitor. If one disables a monitor, is this because the fault is no longer credible, or that the monitor is believed to be dangerously incorrect, or that its false trips have become a nuisance, or that the fault it detects is no longer considered a threat, or that triggering the fault response under present circumstances would conflict with another activity, or what? The first few possibilities clearly refer to modeling issues; so, it is prudent to have a way (arguably not this one) to temporarily preempt an incorrect model. The latter ones though amount to nothing less than direction for the control system to promulgate incorrect state knowledge. That's not transparency, it's cover-up — yet not unheard of in some fault management systems.

Here then is a summary of additional principles (see the first few above, as well) gathered from the preceding discussion:

- State control should be functionally separate from state determination.
- State knowledge should be a product only of state determination.
- State knowledge should appear explicitly and persistently in state variables, as the only interface between state control and state determination.
- Interpolation or extrapolation of state knowledge over time, including the propagation of uncertainty, should be consistent with models of the system under control.
- State knowledge should always be represented in a manner that makes its degradation in time apparent.
- State variable representations should be chosen solely for their ability to clearly and honestly describe state knowledge and its uncertainty.
- Shared representations of uncertainty across state variables should be used where correlation of uncertainty can be significant.
- State knowledge should never be edited to eliminate or misrepresent possibilities, regardless of the implications to state control.

- Observation of unexplained behavior should be among the expressions of uncertainty in representations of state knowledge, where plausible, as a way to acknowledge potential modeling shortfalls.
- State knowledge representations should be sufficiently expressive to cover all significant variations in system behavior.
- No control system mechanism should ever have the effect of masking or corrupting true state knowledge.
- All evidence regarding the state of the system under control should be used for control decisions *only* via its contribution to state knowledge, as interpreted and reconciled by state determination through models.
- Applications of state knowledge outside of state variables should not be persistent.
- If state determination is modularized, state determination modules (including those presenting abstracted or derived views of system state) should be collaborating peers in order to avoid masking state knowledge.
- In a modularized state determination system, each state variable should be the unique product of one state determination module (its single source of "truth").
- Nothing in the interpretation of shared evidence by a state determination module should amount to an internal opinion regarding state knowledge that is produced elsewhere.
- Timers should not be used as substitutes for cognizant models of state behavior.

Turning now to state control, we can identify several more principles.

It has already been noted that knowledge of controlled system states and state behavior, and objectives on controlled system state are usually sufficient to choose control actions. There may be additional criteria (e.g., optimization) for choosing options with the range of possibilities admitted by this principle, but as mentioned before, the focus here will be on basic control success or failure. These additional criteria can be accommodated, once success is ensured. Other apparent exceptions involve cases where a system finds itself in a quandary among many poor choices, suggesting an appeal to some meta-level policy that does not quite fit into the present schema. However, in most cases, this resolves to a nesting of objective sets, where objectives that are more stringent may need to be abandoned for something looser. Thus, control decisions for failure handling remain within the realm of ordinary state objectives.

Basing control decisions on information other than behavior, objectives, and state knowledge can actually be detrimental to a system. Presumably, the only motive for using other data would be either that some other "real" objective besides the imposed state-based one is known, or that state and behavior knowledge has not had the benefit of other available evidence or expertise. However, these conditions violate the principles of transparent objectives, models, and knowledge. Using ulterior objectives subverts the very notion of control. Similarly, using extra data effectively puts the reconciliation of multiple information sources into the control function, creating a second, different version of state knowledge. As diversity of authority and opinion proliferates through such practice, the integrity of control decisions becomes increasingly suspect. Transparency of control consequently requires adherence to the principle that control decisions depend *only* on the canonical three items, for which some consensus has been established.

Unfortunately, violating this principle turns out to be an easy mistake, even with the most innocent of intentions. For example, having noted only that a previous error remains, a control function that tries an alternative action has violated the principle, effectively having made an independent determination of the state of the system. The proper approach would be for estimation processes to monitor control actions, making appropriate modifications to state knowledge to account for the observed results. Control actions would then be chosen differently on the next attempt, given the new state knowledge. For similar reasons raw measurements should be off-limits to control functions, certain arcane issues of stability aside. Control modes are also problematic, as described below under transparent objectives, and likewise for disable flags on fault responses, when misused as control mechanisms (like disabling fault protection during critical activities), and not just as stoppers on modeling or implementation glitches. There are many other examples.

The flip side of using extraneous data is to ignore the state data you have. All kinds of commonly used mechanisms share this problem, such as time-based sequences that make no appeal to state knowledge, relying instead on fault monitors to catch dangerous missteps. Also problematic are responses with no basis at all, other than poor behavior. For instance, if errors are large, but nothing else seems broken, then resetting the offending algorithm makes things different for sure, and with luck might actually accomplish something. However, there is no basis for such a response from the point of view of control principles. (Reset and similar actions at a meta-level of control are different, because they involve the health of the control system itself.) A more likely candidate for puzzling problems of this sort is a modeling error, at which point the system should either shift to a more defensive posture or try to correct the model. Admittedly, the latter is a tall order, out of scope for many systems, but the first line of defense at that point, having validated models, has already been breached. To avoid getting into this situation one must honor the principle of control transparency. Non-specific responses (like reset) signal an opaque design that demands scrutiny.

A useful byproduct of control transparency is that much of the complexity one often sees in control, when viewed properly, turns out to be a state or behavior knowledge problem. This is no consolation to the providers of this knowledge, though it does mean that knowledge is likely to become easier to understand as transparency is improved. Other benefits appear on the control side, as well, such as a more straightforward decomposition of activities, and consequent simplifications in planning.

Finally, it should be noted that the principles of modularity described for state determination apply similarly to state control, where a flat structure is preferred. This is described further in the next section.

The preceding discussion is summarized in the following additional principles:

- Except for secondary criteria (e.g., optimization) or meta-criteria (e.g., priority), control decisions should depend only on state-based behavior and objectives, and on state knowledge.

- State control functions should never directly use measurements or infer anything about the effects of issued commands.

A state-based architecture introduces these additional concepts to the fundamentals of control already identified:

- **State Knowledge —** a representation of a control system's estimate of the history of the state of the system under control, including its uncertainty in this knowledge.
- **State Variable —** the container of persistent state knowledge for some state of the system under control, and intervening between state determination and state control.
- **State Determination —** the source state knowledge in a control system, based on available evidence from the system under control and interpreted though models.
- **State Control —** the source of control decisions in a control system, based on state knowledge, behavior models, and control objectives.

5.2 Goal-Based Architecture

Closed loop control does not eliminate the need for outside entities to know the behavior of the system. Rather, it simplifies their invocation of desired behavior by permitting direct commanding of it in the form of objectives — at least when objectives are transparent. As described above, a state-based architecture will define objectives transparently by expressing them in terms of the state of the system under control. We can go much further though. The second step in endowing a control system with cognizance is to overtly assign criteria for success versus failure to each objective.

Given the definition of behavior and the purpose of control, the success criterion for an objective is nothing more or less than a model of desired changes of state in the system under control. That is, there are certain histories of the state of the system under control that are acceptable (i.e., satisfy the intent of the issuer), while the rest are not. The objective of the control system is to achieve one of the acceptable histories. The success criterion for an objective is therefore a constraint on the history of the state of the system under control. Either the history of the state satisfies this constraint, or the objective has failed. We call such a verifiable objective a **goal**. Similarly, we refer to an architecture that expresses all objectives as goals, and which explicitly and scrupulously coordinates, plans, schedules, and monitors all control system activities in terms of

Common Points of Confusion

Most control systems may be said to exhibit goal-directed behavior. Though it is usually reserved for behaviors that are more complex, requiring multiple steps, the term is applicable wherever an objective is defined as a state to be achieved. It is common, however, for the objective to refer to a control state (e.g., a "safing" goal), since the distinction between control system and system under control is often unclear.

Moreover, even if a goal is against the state of the system under control, it is commonly not expressed in a way that makes the criterion for success explicit, or that associates failure to a particular objective.

Furthermore, the expression of objectives as goals, as defined here (explicit constraints on state history), is generally not exploited directly and universally in the coordination, planning, scheduling, and monitoring of objectives.

The presence of goal-directed behaviors is therefore not enough to qualify an architecture as goal-based.

goal criteria, as a **goal-based architecture**. A goal-based architecture is necessarily state-based, as well.

Note the difference between a naïve state-based objective and a true goal. An objective to point a camera at a target, for example, might simply indicate some pointing direction; but deciding whether or not this objective is achieved is problematic, because there is no criterion for how far from a perfect orientation is tolerable. On the other hand, a goal (as defined here) to point a camera might allow any meander among the orientations that align the boresight within some tolerance of the target direction. The distinction then is that issuers of true goals must say exactly what behaviors they consider tolerable, rather than allowing this to remain implicit or open to interpretation. Goals provide the control system with criteria to verify achievement of the required behavior. Thus, goals, being verifiable state constraints, are far more transparent as expressions of objectives.

Another subtlety in the definition of a goal has to do with the manner in which it is verified. A control system, after all, has no access to the state directly, but can only base its assessment of success or failure on state knowledge. That means that the state constraint of a goal must be expressed in terms of the available state knowledge representation, including its expression of uncertainty. A more precise definition of a goal must therefore refer to constraints on *knowledge* of state history.[16]

This clarifies a common issue in control system design, where many of the actions taken by the control system are devoted to acquiring improved knowledge. The change in behavior sought of the system under control in such cases might be merely to alter the measurements it provides, or at the opposite extreme, could require a substantial system reconfiguration to enable a fault diagnosis. Whatever the case, such actions are motivated by objectives to reduce uncertainty. This becomes fair game for goals, once we realize that uncertainty in state knowledge is part of the state history subject to constraint.

It is essential to note with this refinement that, whereas one’s direct intent may be focused on the system under control, assessment of success or failure is never better than the control system’s knowledge of state. This does not create a new problem; it merely acknowledges one that already exists in any real system. There is an inevitable trade between certainty of knowing about success or failure and the precision of objectives. For instance, in the objective to maintain a safe system state, one must choose between a looser definition of safety and a higher likelihood of fault alarms. Thus, pulling uncertainty under the oversight of goals not only enables management of state knowledge quality, as part of the control process, but also makes the practical limitations of control transparent. As always, transparency is preferable.

A key principle of goal-based control is to be specific and precise. As we have seen, the way objectives are commonly defined can leave certain aspects of behavior undefined, implicit, or conditional. Even the pointing goal above involves a little omission, where rotation around the boresight has been ignored. Such slips may make someone unhappy, but infractions that are far more serious are possible. Failure to appreciate this can lead to dangerous vulnerabilities.

[16] Of course, this leaves open the potential to meet goals merely by faking state knowledge in some way, such that it is a dishonest interpretation of the available evidence and behavior models, but we already have a principle against that.

There are reciprocating aspects to this. First, an objective is not transparent if a control system produces behavior other than that expressed in its objective, while claiming otherwise. This much is clear. Conversely, though, when specifying an objective, anything short of expressing full intent is also not transparent. For example, if one intends a latch valve to be open for the duration of a propulsion activity, then a directive to open the valve does not express the real objective. A valve control system could certainly act on such a directive, but not knowing the full intent, it would have no subsequent basis for rejecting competing directives, or for determining whether the actual intent of the directive had been met, or for taking any corrective action if it had not.

Real transparency therefore amounts to a sort of contract between the issuer of objectives and the control system achieving them, where objectives are expressed in a way that makes success or failure mutually obvious to both parties (i.e., where objective are goals). Unfortunately, this is frequently overlooked. In such cases, delivered behavior can be different from what is expected, and the difference will not necessarily be apparent. Closed loop behaviors that are quirky, prone to surprises, hard to model, unreliable, or otherwise opaque about their capabilities are of questionable benefit. In fact, this is a primary concern behind reservations over increasing autonomy in space systems (including fault protection), which essentially amounts to permitting greater closed loop control on the vehicle. Operators are more comfortable dealing with the arcane details and peculiarities of a transparent open loop system, than with the broken promises of an opaque closed loop system. Resolving this dilemma requires careful attention to making objectives say what they mean, and mean what they say.

Even where intent is clear though, systems still generally fall short in making overt connections between setting objectives and checking for their success. For example, pointing objectives rarely include an overt criterion for how well pointing must be done, while fault monitors looking for excessive pointing control errors rarely have any idea of their relevance to current pointing activities. Similar problems occur for tasks where the only indication that there might even be an objective is a fault monitor (e.g., “too much” thruster firing). In still other cases, direction is given to systems with no explicit objective at all regarding the states under control, placing objectives instead on the state of the control system itself (e.g., “cruise mode”). In all such cases, responsibility for meeting objectives has been divided, and connections between objectives and their success criteria, where they exist, are generally implicit — hidden assumptions that are vulnerable to abuse. Intent is opaque, so predictably, coordination issues arise.

Consider what might happen, for instance, if control transients at objective transitions became problematic for separate monitors of excessive control error. If gadgets, such as persistence filters, are added in the error monitoring system to ride out such predictable nominal events, the system becomes vulnerable to *real* problems that might occur during transitions, and it is overly desensitized to errors generally. On the other hand, if control error monitors are given access to control system internals (such as deadbands, modes, or gains) and other information they may need to be more discriminating, the result is a proliferation of *ad hoc* interfaces, an Achilles heel for any design. Thus, separating monitors for excessive control error from the control functions responsible for regulating that error cannot be defended because it is simpler. Hidden assumptions, opaque intent, potential inconsistencies, fault coverage gaps, coordination problems, tangled interfaces, and so on are not simpler. They are merely symptoms of a complexity problem.

As with state determination, state control is often modularized, each module controlling a sub-element of the system under control. In such composite systems, it may be possible to exercise control on the appropriate sub-element only indirectly via chains of influence from one sub-element to another, each with its own control module. For instance, drivers affect thrusters, which affect attitude. Therefore, attitude control requires thruster control, which requires driver control. Even if directly controllable, a sub-element may be disturbed by other such chains of influence (e.g., a gravity gradient). However, engineered systems generally have some identifiable structure in their composition that tips the balance in favor of meeting objectives.

The hierarchy of functional decomposition is immediately recognizable in this picture. Therefore, there is a tendency in control systems for the composition of state control modules to be treated as a hierarchy, some modules and their objectives deemed as “higher level” than others. Thus, one might hear that a science observation is a high-level objective, while pointing, actuator power, and so on are successively lower levels of control. Similarly, one might hear of a high-level mode overlaying lower level modes.

The problem with this point of view is that these relationships are not always the operative ones under all circumstances. For example, if battery energy drops to dangerous levels, a function near the bottom of the stack suddenly takes the top position, and science instruments are often the first to be switched off — a dramatic inversion of levels. Many such examples can be found in any system, and upon closer examination, it becomes clear that there is really no One True Hierarchy of control. Rather, there are a large number of intertwined dependencies with shifting objectives and priorities driven by need and circumstance. Trees of dependencies, rooted at important objectives, are misinterpreted as hierarchical levels of control, rather than as just part of a larger topology of a coordinated control. This topology is driven, not just by relationships established in the original functional decomposition, but by other physical and logical interactions in the system under control, such as shared resources, interference, fault propagation, broad environmental influences, and so on. These interactions each obey their own notions of hierarchy, such that the overlaid network of effects among the states of the system under control becomes rich and intricate. Trying to approach what is not hierarchical with a strictly hierarchical control scheme leads to inevitable dissonance. Systems resort to multimode hierarchies depending on the circumstances (e.g., normal command sequencing versus safing), where dependencies are allowed to shift. However, this is ultimately quite difficult to manage.

Flattening the state control structure provides a means to deal with these issues. With control modules as peers, none are defined in terms of others. Instead, they interact cooperatively, each telling appropriately selected others what they need and what they can do. This interaction, of course, is through goals. For instance, an attitude control module, knowing that thrusters affect attitude in a certain way, can influence the behavior of thrusters to provide just the effects needed. Moreover, it can request this influence directly, by placing goals on thruster behavior with the thruster control module. This permits it to deal with these effects without having to know anything about how the required behaviors will be achieved. It merely asks for the behavior it needs.

Given these relationships, each module need concern itself only with immediate state effects. Therefore, the topology of interconnections among state control modules can be mapped directly to the topology of modeled state effects in the system under control. There is a similar mapping to the state determination module topology. This transparent relationship between the structure of the control system and that of the model of the system under control is a recurring theme explored further in the discussion below.

This is not quite the whole story, since one module may place a goal on another that cannot be achieved. This may be because the goal is not within the capability of the system; it may be that ripple effects violate other goals elsewhere; or it may be because yet another module has requested a conflicting goal. It is also possible that an accepted goal will fail. These are all different aspects of the failure handling requirements of a control system, because in each case some goal will not be achieved.

Whether or not such issues can be coordinated locally and reactively will depend on the nature of the problem. Often a more deliberative approach is needed. Nonetheless, the expression of *all* such issues in terms of transparent goals means that they can all be handled in terms of modeled behavior of the system under control. Goals describe desired behavior of the system under control, which can be evaluated through models of that behavior to determine which combinations of goals are feasible. As always, all control system features appeal transparently to the system under control.

Moreover, the universal application of transparent goals within a goal-based architecture enables the normalization of all competing goals to a state control module. As constraints on the history of state, goals may be directly compared and merged (both being essentially set operations), in order to detect conflicting intents and meet shared intent, respectively. This makes prediction, planning, and coordination easier to do with confidence and efficiency, and hence easier to automate, if desired. This is normally *much* harder with arbitrary command-based systems.

This is a summary of principles derived from goal-based control:

- Control objectives should be specified only as goals, which define success via constraints on the control system's knowledge of state history for the system under control.
- Goals should express explicitly, completely, and unambiguously the full intent of the issuer against the state of the system under control (not the state of control system).
- The quality of state determination should also be managed via goals, where constraints on uncertainty motivate appropriate control decisions (e.g., measurement collection or diagnostic actions).
- All system objectives relevant to the success or failure of a system should be overtly expressed, and each check for failure in a system should be directly associated with a goal that defines the success criterion.
- If state control is modularized, state control modules should be collaborating peers in order to avoid a dominant control hierarchy.

- The topology of interactions among state determination and state control modules should reflect the topology of interactions among their assigned states in the system under control, dealing in either case only with immediate state effects.
- Whether anticipated during planning or discovered during execution, monitoring for goal achievement and reacting to non-achievement should be considered an essential part of the overall control task.
- Reasoning about control objectives should be accomplished entirely in terms of goals, which are fully modelable entities in a form that supports normalized prediction, planning, and coordination operations.

A goal-based architecture introduces one essential new concept to the fundamentals of control identified so far:

> **Goal** — a control objective expressed as a verifiable constraint on knowledge of the history of the state of the system under control

5.3 Model-Based Architecture

As described above, in order to reason about which goals a control system can achieve and how these should be coordinated among state control modules, knowledge of the behavior of the system under control is needed. As constraints on behavior, goals must be consistent with plausible behavior. This is part of the overall task of devising appropriate state control actions, which also depends on models of the system under control. Similar considerations apply for state determination functions, which use models to interpret and reconcile evidence and to project estimates into the future. In fact, everything fundamental about a control system depends in some way on models of behavior for the system under control. Therefore, any hope that we might understand the system as a whole hinges on the clarity, consistency, validity, and proper use of these models. This is hard to ensure if these models are not transparently inspectable, wherever they affect the design or its operation. Thus, the third step in endowing a control system with cognizance is to expose and consolidate all models across a control system. We refer to an architecture, where the model behind each implementation choice is clear, and all appeals to that model can be shown to be consistent, as a **model-based architecture**. A model-based architecture is necessarily state-based, because models used in control must always be in terms of state. A model-based architecture is also naturally complementary to a goal-based architecture, because only when objectives are transparent can their achievement be fully modeled.

Common Points of Confusion

Every control system design is based on models in some way, even if that model happens to reside only vaguely in someone’s thoughts. The objective of a model-based architecture, as defined here, is to make all such models transparent.

This does not mean that all reasoning accomplished in a model-based architecture requires the explicit execution or manipulation of models performed by generic reasoning engines. An architecture can be model-based while still relying on traditional algorithmic approaches, as long as the relationship between the model and the implementation is clear and verifiable.

We have already seen several instances of common practice that violate principles of modeling transparency, including indiscriminate use of raw evidence, careless extrapolation, opaque timers, neglect of modeling uncertainty, gadgets (e.g., reset or disable) to deal with potential modeling issues, and so on. In general, models tend to be divided, scattered, and implicit, with only an *ad hoc* connection to the structure of the system.

Consider for instance the split between the way normal and abnormal behaviors are monitored. Models play an essential role in acquiring state knowledge by providing expectations against which evidence of state can be compared. Assuming a model is right, then any departure from expectations is an indication that a knowledge correction of some sort is in order. Adjustments are typically small, to accommodate measurement and disturbance noise, but may need to be large if the only way to align observations with a model is to hypothesize a discrete change of state, such as a modal change, or a fault. When models are viewed in this way, it becomes clear that there is no transparent way to separate models of normal and abnormal behavior, and hence to divide normal state determination from fault diagnosis. Nonetheless, this is not only the norm, but often defended as a separation of concerns — a perilous misunderstanding of control fundamentals.

This division of responsibilities frequently arises from the careless conflation of the two distinct sorts of error one sees in a control system. Large control errors indicate a problem in meeting objectives, whereas large expectation errors suggest a problem in state knowledge. The first calls for a correction in control, such as a fault response, while the latter calls for a correction in knowledge, such as the diagnosis of a fault. The former is *not* a fault diagnosis, and the latter is *not* an objective failure. However, when both types of excessive error are treated the same, they start looking different from everything else and will then tend to be split off from the rest of the implementation. Thus, the division of responsibility for objectives in control functions described above (nominal behavior in one place — faulty behavior in another) has an analog in the division of responsibility for knowledge in estimation functions. In both cases, models of behavior have been broken and scattered, resulting in a loss of model transparency.

Similar problems of transparency arise in the connections among control system modules. In order for a control system module to do its job well, it helps to close loops as well within the modules it depends upon, making those modules transparent in all the respects described here. However, since such dependent behaviors affect what a control system module can do, models of these dependencies should be reflected in the behaviors that a dependent module pledges in turn to others. Unless this is accomplished in an open and disciplined manner that allows integrity to be preserved and verified, subtle errors can creep into implementations that are hard to spot.

Consider, for example, an objective to pressurize or vent of a propulsion system. Doing this properly depends on adequate settling of pressure transients, but the relationship between such an objective and the valve actions taken to accomplish it in typical systems is frequently implied only through command sequence timing (e.g., open a valve, wait “long enough”, close the valve). Even in systems with conditional timing (e.g., open a valve, wait for pressure to reach some threshold, close the valve), this dependency is often still implicit, the functional connection between the actions and the objective having been made elsewhere (e.g., in an uplink sequencing system), but beyond the awareness of the control system responsible for the objective’s accomplishment. Nothing in the normal way the propulsion activity is done carries any information about the effects of altered timing or actions in the sequence Therefore, any interruption that puts fault protection in control, perhaps for totally unrelated reasons, suddenly exposes the missing connections. What must a fault response do then to safely secure propulsion capability, once the interruption is handled? Simply resuming the sequence is problematic, since the assumed timing no longer applies (e.g., the interruption lasted several minutes) or assumed conditions may have changed (e.g., fault protection closed the valve as a precaution); and restarting may not work, because initial conditions are different (e.g., maybe unsafe pressures). Making the objective of the activity transparent would at least provide a basis for recognizing an unsafe condition, but gaining insight and concocting a reasonable response has to be done with no help from normal procedures. They were designed separately, assuming only nominal behavior. Thus, the result yet again is an *ad hoc* growth of scattered models, functions, and interfaces, which may not always get things right.

Adding to such problems is the typical absence of good architectural mechanisms to negotiate potentially competing objectives from multiple sources, as in compositions of state control modules like those described above. Neglecting to articulate and negotiate potential conflicts, such as wanting to minimize thruster firing while achieving good antenna pointing, both involving objectives on attitude, will result in disappointing someone — usually the one without the explicitly stated objective. Mechanisms for accomplishing such coordination routinely include liberal manipulation of modes, timers, enables and disables, priority or threshold tunings, locks or serialization of responses, and the like in most implementations. However, these are all indirect, low-level, devices for managing software programs, not explicit, high-level architectural features for coordinating system objectives. The latter require an anticipation of the objective failures, and that invokes models.

Problems caused by such a deficit of structure are even more apparent in the presence of shifting system capabilities. Whether this happens during development and integration, as system understanding evolves, or during flight, as failure or serious degradation accumulates (e.g., from wear, radiation, or other stresses), the resiliency of the system will depend primarily on how easily it can accommodate substantial changes in system behavior (and hence in the models of this behavior). Such flexibility is a widely recognized hallmark of good engineering design. Indeed, the need for flexibility is the very reason such functions are captured in software — as we have seen.

Despite this, however, most designs focus on a narrow definition of normal behavior. Ordinary variations are tolerated within a set range of objectives. However, even fault tolerance functions (as with system management in general), which frequently require the rest of the system to adjust to major changes in the available objectives, is structurally prepared to accommodate such changes in only a few specialized cases, such as in safing modes or critical activities. Any situation that takes the system outside this predefined range, and any significant capability change generally, is viewed as something outside the architecture altogether and requiring a redesign. The altered system is not the one for which the undertaking was prepared, so for every potential alteration of available objectives in one place, there must be corresponding accommodations among all issuers of these objectives. These accommodations may result in further changes of behavior, and so on, rippling through the system until the effect is contained. Therefore, a design structured around one strict repertoire of normal behaviors (and a few safing behaviors) will have difficulty managing such changes.

The systems in real danger are those where propagating effects of change have not been adequately considered or characterized, or where an assessment of this coverage is obfuscated by a design that fails to make these effects transparent. In other words, there is not an adequate model of the effects, and typically, the control system does not overtly reflect such models or operate in an easily modelable fashion. Otherwise, the effects could be understood. Unfortunately, as with negotiation mechanisms, most deployed systems show little overt architectural support for this notion. Instead, one typically finds minimal mechanistic support for change (e.g., ICDs, `*.h` and `make` files…), accompanied by a plethora of *ad hoc* mitigations. Without the supporting structure of a principled architecture looking after this basic notion of closed loop control, the job gets a lot harder.

A more forgiving and flexible approach to this problem is to make the control system design more overtly responsive to a uniform model of system interaction, guided by models of behavior. The network of information exchange within a goal-based architecture is all about behavior, both in the content of interchange (because goals explicitly express the desired behavior of the system under control) and in the topology of interconnections (because it reflects the topology of effects within the system under control, which objectives seek to exploit). Therefore, the information required to accommodate model changes is already transparently available. If the designs of the modules also reflect similar transparency to behavior models, then the impacts of change are even more readily apparent.[17]

Another common mistake arising from neglect of principles for transparent modeling is the dispersal of essential behavioral information among complex parameter sets. No doubt buried there, for both designers and operators to sort out, are important relationships among objectives and behaviors. However, not only can their implications to actual behavior be opaque, but there is also typically no structure provided to ensure consistency of objectives with represented behaviors, consistency of the parameters among themselves, or consistency of the implied behaviors with actual behavior. Various mitigating processes can be tossed over this problem, but the haphazard treatment that created the problem in the first place leaves quite a mess under the carpet. The behaviors that emerge can be quite unpleasant.

In fact, parameters are simply parts of models or derivatives of models and should never stand alone. The models themselves are what matter, since this is where behavior is captured. Parameters merely designate an instance within the range of possible behaviors. When parameters appear within this context, the relationships among them become apparent, and consistency and validity are easier to ensure.

The principles for model-based architecture are recapitulated here:

- Models used for or in control system designs should be thoroughly validated.
- Models should be expressed only in terms of the states of the system under control.
- The relationship between the model of a system under control and its control system implementation should be clear and verifiable.

[17] In principle, it may even be possible in many cases to make the design entirely generic, aside from the models, such that updating the models accomplishes the required design changes as well. Such technologies already exist in specialized cases.

- There should be no inconsistency among the models behind a control system design (ideally accomplished by avoiding duplication or overlaps).
- Control system functions should not be divided in such a manner that models of behavior are also split (e.g., between normal and abnormal behaviors).
- The achievability of an objective should reflect the modeled behavior of the system, given the achievability of the subordinate objectives.
- The composite achievability of contemporaneous objectives should reflect the modeled behavior of the system interactions and be resolved through overt negotiation mechanisms.
- The content and topology of interactions among control modules should reflect the modeled behavior and topology of interactions within the system under control.
- Parameters describing the system under control should only be described and managed within the context of models.

A model-based architecture requires adding no new concepts to the fundamentals of control identified so far.

6 Postscript

The descriptions above have given only the barest of overviews for state-, goal-, and model-based Architectural approaches to cognizant control. If taken seriously, the ideas outlined here can be formalized, additional principles of their application can be added, resulting patterns can be captured in concrete reusable software frameworks, and a great deal of specialization and support can be built on this base. To do this topic justice though requires far more exposition than is appropriate here.

Another reason for not going further, however, is that beyond the fundamentals outlined here, there are many additional layers with many variations that can be added, as concepts are specialized and new abstractions are added to the mix. Only with these additions do the fundamentals become recognizably part of an actual flight software program. The purpose here has not been to suggest that these many details are secondary. This is hardly the case, since these details will comprise the vast majority of the design effort. Rather the purpose has been to argue that this multitude of details, without a rock solid foundation in fundamentals, is prone to collapse. The most important of these fundamentals in spacecraft software is control, and to the extent control can be accomplished within a cognizant framework of state-based, goal-based, and model-based architecture, there is some hope for continuing progress against growing complexity.

At this point, however, it is easy to imagine vigorous protests regarding many other issues of complexity have been neglected in this account. What about the complexities of large, multi-threaded, real-time programs and the management of computing resources? What about the complexities of requirements development and associated costing and planning? What about the complexities of collaborative, iterative software development? What about the complexities of verifying and validating complex software? What about the complexities of coordination and interoperability among systems of systems? What about the complexities of operator interactions with these system, whether ground operators or astronauts?

The short answer to these questions is to reiterate the appeal to patterns, as described previously for the management of complexity in general. The patterns for control sketched above address directly many system level issues of complex interactive behavior and the emergent characteristics of mismanaged complexity that hound present designs. However, the mere presence of orderly structures for dealing with these issues has numerous side benefits that help confront other complexity problems, as well. Some of these are summarized, as follows.

- **Real-Time Software —** The approach outlined here recasts all control functions within a uniform, relatively flat framework of cooperating modules, where a model of their interactions is part of the fabric of the architecture. This allows most real-time software issues to be gathered in a principled manner into the regular structure of control. There they can be managed far more rigorously than is often the case in large real-time designs, where interactions tend to be more irregular, execution management more distributed and *ad hoc*, and overt ties between the structure of the control problem and the mechanics of its execution are generally missing.

- **Requirements —** As described earlier, requirement gaps, growth, and instability aggravate software complexity problems, but poor requirements and complex software are in many ways both victims of a common, more fundamental problem: lack of understanding of the application at hand, and inadequate means of communicating what understanding there is between systems and software engineers. Cognizant control principles and supporting architecture address this issue by providing a disciplined framework within which to more systematically and thoroughly understand and specify the interactive behavior and control objectives of a system, and to relate this understanding directly to the structure and performance of the software. Because of the more seamless model-driven relationship between systems and software engineering afforded by commonly held patterns of understanding and implementation (in contrast to present document-driven methods, requiring heavy interpretation, translation, and extrapolation), this can be accomplished in an iterative, incremental manner that is both intrinsic to the systems engineering effort and profits from the assets software brings to system development. It also naturally aligns with broader initiatives in model-based engineering, which are finding increasing application in modern project developments as a means to deal with complexity.
- **Costing and Planning —** Estimates of software development cost and schedule routinely fall far short of reality. This is compensated in present processes largely through fudge factors learned through painful experience, such that one rarely finds an early accounting that predicts the scope and scale of the final product in any detail, except where heritage is clear and unchallenged. The architectural approach described here makes it easier to penetrate more quickly and thoroughly into both the required functionality of the software *and* the structure best suited to meeting these needs. This more direct and detailed accounting can result in more accurate and complete cost estimates, better metrics during development, earlier anticipation of growth, and easier accommodation of change.
- **Collaborative, Iterative Development —** Emergent behaviors in complex software tend to arise, not so much at the level of individual contributions to a design, but rather in the way such contributions interact as a system. Since most interactions in flight software relate to control issues, an architectural framework of principled, transparent control provides the means to engineer these interactions in a manner that leads to system level understanding that can be maintained and exploited throughout development.
- **Verification and Validation —** There are two reasons one might have confidence in a software system to perform its functions robustly and reliably. Both are required, and neither has to do with exhaustive testing, which is fundamentally impossible. Rather, one gains confidence 1) when the design of the overall system adheres to solid principles of good, understandable behavior, and 2) when assumptions, models, and other information that inform the design have been confirmed through appropriate trials. One provides the theory of operation and the other a demonstration of its veracity. Through the principles of transparency described here, cognizant control is a major contributor to providing the first of these criteria and laying the foundation for the second.
- **Interoperability —** When the topic of interoperability arises, discussions often turn reflexively to mechanistic topics of data exchange. What matters most for confident interaction, however, is not the medium of exchange. Rather, it is the appropriateness of the information exchanged to the needs of interaction, and the mutual understanding among the

communicating parties regarding that information. Each entity in the communication needs to clearly understand the relevant behaviors and capabilities of its partners, unambiguously convey its objectives regarding that behavior, and understand the shared situation within which they are interacting. These are precisely the issues addressed through a cognizant approach to control, which is readily extended to interoperating systems.

- **Operator Interactions** — Operators play far more elaborate roles in their relationship to software control functions than is generally given credit. This is because no operator can ever be in total control. Besides their relationship to other humans, operators may be director or peer, judge or consultant, sensor or actuator, or even the object of control in some circumstances, relative to their software counterparts in the overall control structure. In fact, operators often change roles or fill several roles simultaneously. Even among people, it is easy to misunderstand nuanced communication, where words alone do not convey full meaning. Shared understanding and experience and other queues are necessary to communicate with clarity. This is why operations environments have become as structured and formal as they are. Extending this relationship to software has been problematic, with most of the burden for correctness falling on operators. More complex systems make this increasingly difficult. The value to this concern of a framework for understanding and characterizing control is to overlay a *shared* pattern on the roles and relationships of both software *and* operators that is grounded in fundamental principles of control. Thus, when an operator acts, it can be with full appreciation of the particular role associated with that act, as it will be understood by the software — something that can lead ultimately to a much more intelligent conversation between humans and machines.

In these examples it becomes clear that a foundation for understanding complexity in behavior and control has broad benefits to managing complexity overall. It is not the whole story, but clearly, nothing is more important than getting these control fundamentals right. If this is outside the box, then the box must be inside out.