AIAA Infotech@Aerospace Conference
and
AIAA Unmanned...Unlimited Conference
6 - 9 April 2009, Seattle, Washington

AIAA 2009-1882

NASA Study on Flight Software Complexity

Daniel L. Dvorak*
Jet Propulsion Laboratory, California Institute of Technology, 4800 Oak Grove Dr. Pasadena, CA 91109

In 2007 the NASA Office of Chief Engineer commissioned a multi-center study to bring forth technical and managerial strategies to address risks from growth in size and complexity of flight software 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. This report examines complexity throughout the engineering lifecycle—from requirements definition through design, development, verification, and operations—and presents sixteen findings and associated recommendations.

Nomenclature

APL	=	Applied Physics Laboratory
ERTS	=	embedded real-time software
GSFC	=	Goddard Space Flight Center
JPL	=	Jet Propulsion Laboratory
JSC	=	Johnson Space Center
MSFC	=	Marshall Space Flight Center
NASA	=	National Aeronautics and Space Administration
NESC	=	NASA Engineering and Safety Center
NPR	=	NASA Procedural Requirement
OCE	=	Office of Chief Engineer

I. Introduction

SOFTWARE continues to play a growing role in NASA's space missions, in both human and robotic missions. There are many kinds of mission software, with most of it classified as either "flight software" or "ground software," reflecting the location where it executes. Although ground software is larger by far, flight software is considered higher risk because it interacts directly with spacecraft hardware, controlling virtually all of the onboard systems in real time at various levels of automation. Over a span of four decades, the amount of flight software per mission has grown rapidly, along with an increase in the number of problems attributed to it, as seen in both pre-launch and post-launch activities. In 2007 the NASA Office of Chief Engineer (OCE) initiated a study to examine the growth in flight software size and complexity and recommend ways to better manage complexity. The study involved five space flight centers: NASA Goddard Space Flight Center, NASA Jet Propulsion Laboratory, NASA Johnson Space Center, NASA Marshall Space Flight Center, and Applied Physics Laboratory of Johns Hopkins University.

The study adopted a simple definition of "complexity" as 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 also in upstream activities (requirements and design) and in downstream activities (testing and operations). The study made a distinction between *essential functionality*, which comes from vetted requirements and is therefore necessary, and *incidental complexity* that arises from decisions about architecture, design and

* Principal Engineer, Systems & Software Division, Mail Stop 301-270, AIAA Member.

implementation. The latter can be reduced by making wise decisions, and is therefore the target of most recommendations.

Flight software is an example 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, and estimates for Orion[†] flight software exceed one million lines of code. The trend is expected to continue because of the increasing capability of flight computers and the engineering advantages of situating new functionality in software or firmware rather than hardware. Flight software has become the "complexity sponge" for space systems, making it an enabler of progress as well as a cause for concern. 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 the average GM car has grown from 100 thousand lines of code in 1970 to a projected 100 million lines of code in 2010.

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 and idioms that are hard to understand or analyze. Testing can be unnecessarily complex when testability is not properly considered in software design. Finally, all the complexities that have not been handled earlier complicate the job of operations. Although some kinds of operational complexity are deliberate or unavoidable, other kinds are an unconscious 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.

Findings in each of these areas led to over a dozen recommendations in the areas of systems engineering, software architecture, testing, and project management. This report is a condensed version of the final report[2] submitted to NASA OCE in March 2009. This report first examines the growth in flight software *size* Section II) and *complexity* (Section III), recognizing that they are not well correlated; a small program can be complex and a large program can be simple. Section IV describes the findings and recommendations, and Section V reports on an important area where software architecture and fault protection intersect, resulting in incidental complexity. Finally, Section VI offers a historical perspective on software complexity dating back to the 1968 NATO conference on software engineering.

II. Flight Software Growth

A. Flight Software Described

As its name suggests, flight software executes onboard a spacecraft[‡], 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: attitude control; entry, descent, and landing; surface mobility; science observations; environmental control; instrument control; and communications. Flight software also provides numerous 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. Four characteristics distinguish flight software from most of the familiar commercial software used in daily engineering and management work[1]:

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.

[†] "Orion" is the name of the crew exploration vehicle, part of NASA's Constellation Program for returning to the Moon.

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

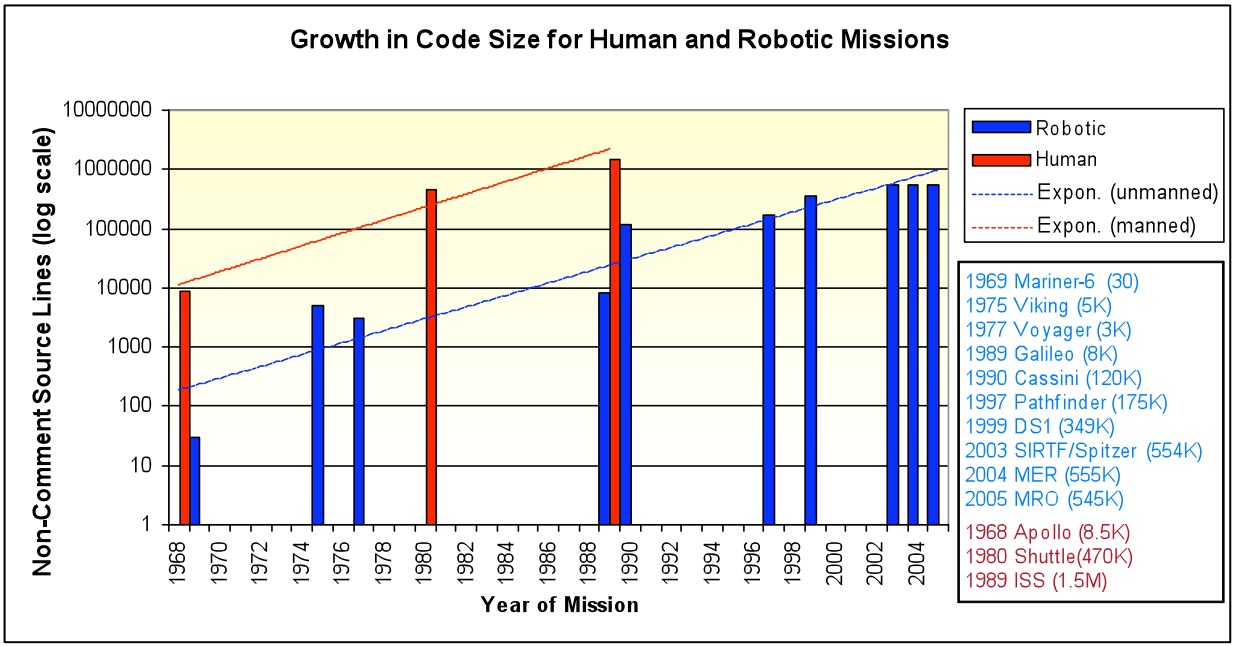

Figure 2. History of flight software growth in human and robotic missions.

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 radiation-hardened 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.
4. Flight software performs real-time processing, meaning that it must satisfy various timing constraints involving periodic deadlines in control loops, timely execution of timed events, and fast reaction time to asynchronous events. Real-time software that performs the right action, but too late, is the same as being wrong.

These 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).

B. Growth of NASA Flight Software

One objective of this study was to measure the growth in flight software size and identify trends within NASA. Figure 2 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 chart is typically the year of launch or completion of primary software. Line counts are either from best available source or direct line counts. 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 Norman 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"[3].

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, based on data from GSFC, MSFC, and APL. The lack of a growth trend at GSFC 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.

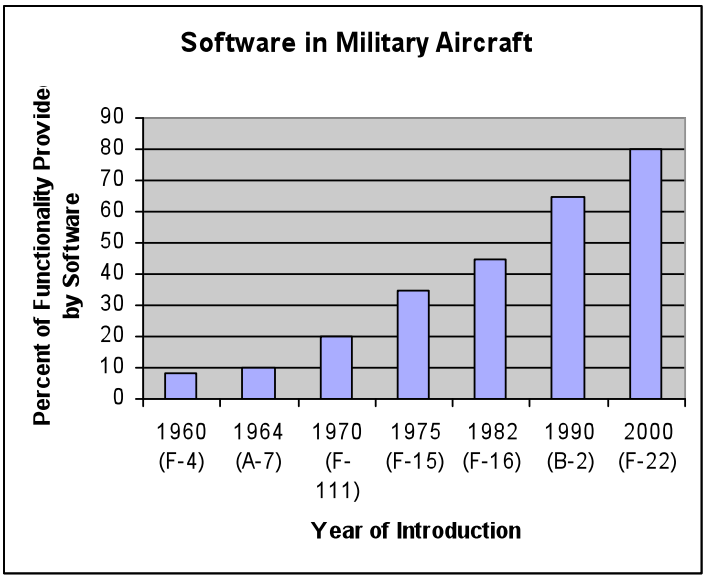

Figure 3. Growth in functionality provided by software to pilots of military aircraft.

C. Growth of Embedded 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[4]. 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 3. The F-22A is reported to have 2.5 million lines of code.

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. In 1970 the average General Motors automobile had a hundred thousand lines of code, but now a premium-class automobile has close to 100 million lines of code running on 70 to 100 microprocessor-based electronic control units. Although not all of the code is not critical to the car's operation, it is indicative of the continuing trend with embedded systems.[13]

D. 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."[5]

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. The prime example in this category is 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[6].

To help further understand the growth in functionality assigned to flight software, as well as its continuation into the future, Table 1 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 1. 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

III. 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 examines complexity from different perspectives in an attempt to highlight the properties of a system that make it hard to understand.

A. 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"[7]. 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.

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.

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*[§]. "Essential functionality" comes from requirements and, like well-written requirements, specifies 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.

[§] 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".

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 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 Recommendations C and A, 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.

Interdependent Variables

In a study of failures in complex systems, Dietrich Dörner[8], 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*.

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[9] 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 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 4. 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.

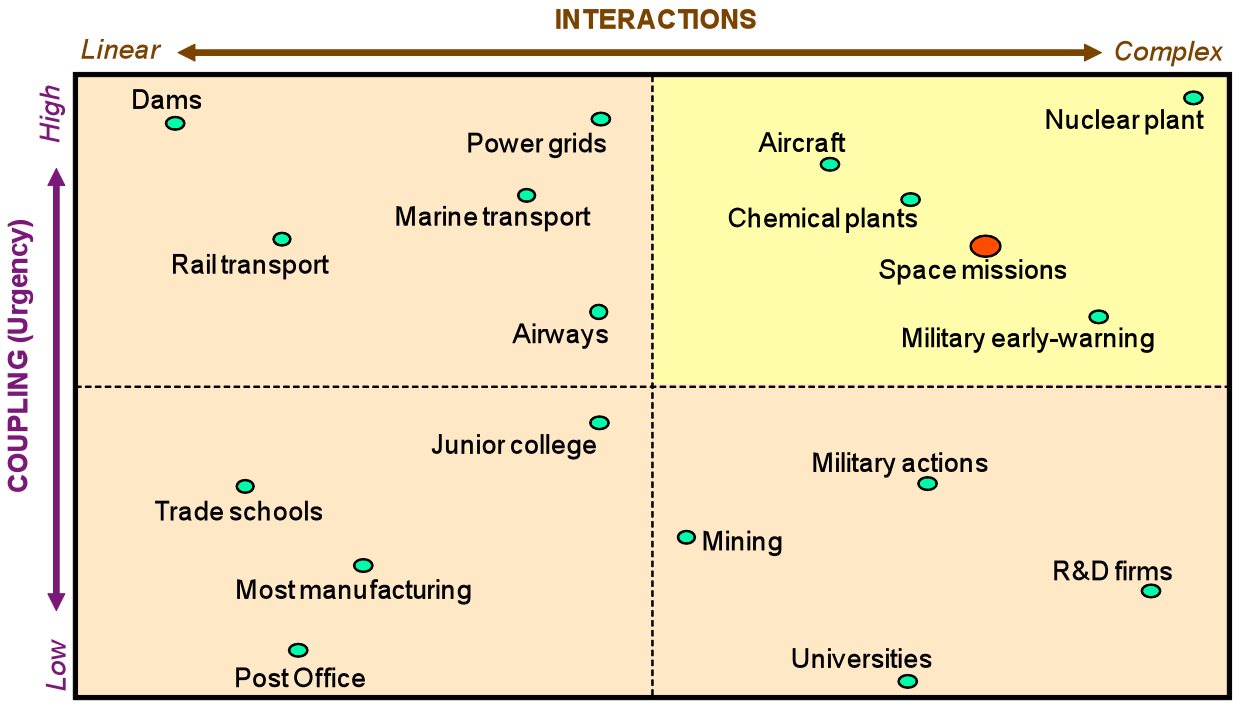

Figure 4. Systems that must manage complex interactions and high coupling are more prone to accidents. Space missions are among these high-risk systems.

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'"[10].

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 that 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.[11]

B. Software Complexity Metrics

In order to present a picture of growth of flight software within NASA, Section II 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 2 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, to alert themselves to code that should be considered for refactoring. We did not discover any projects that mandate such metrics.

Table 2. 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

It's important to note that these complexity metrics only apply to source code and therefore cannot alert projects earlier in the engineering lifecycle when the only available artifacts are requirements and architecture and high-level design. Research is needed to identify this kind of early-design complexity metric, as suggested in Recommendation P.

IV. Findings and Recommendations

This study produced 16 recommendations, each of which is described separately in the following subsections. Completely after the fact, the recommendations were grouped into five categories shown in Table 3. 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.[12]

Table 3. Recommendations grouped into five categories.

Category	Sec.	Recommendation
Complexity Awareness	A	Raise awareness of downstream complexity
Project Management	B	Emphasize requirements rationale
	C	Emphasize trade studies
	G	Involve operations engineers early and often
	J	Stimulate technology infusion
	P	Collect and use software metrics
Architecture	D	More early analysis and architecting
	0	Establish architecture reviews
	F	Nurture software architects
	I	Invest in reference architecture
Fault Protection	L	Standardize fault protection terminology
	M	Establish fault protection proposal review
	N	Develop fault protection education
	O	Research fault containment techniques
Verification	H	Analyze COTS software for verification complexity
	K	Use static analysis tools and code compliance checkers

A. Raise Awareness of Downstream Complexity

Finding

Although engineers and scientists don't deliberately make things more complex than necessary, it sometimes occurs 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 spawns a cascade of increased complexity in analysis, design, development, testing, and operations. Similarly, a hardware engineer might choose a hardware interface design that unknowingly imposes a hard-to-meet timing requirement on avionics software; a systems engineer might omit a simple capability from flight software that unknowingly 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.

Recommendation

Raise awareness about complexity through educational materials that provide easy-to-understand examples from previous missions. Such material could be provided on a NASA-internal web site that everyone can see and contribute other examples. This could be part of the NASA Lessons Learned Information System, categorized under the keyword "complexity". The full report to OCE contains a "Complexity Primer" as a beginning example. 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 (Recommendation C) and the need for more up-front analysis and architecting (Recommendation D). 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.

B. Emphasize Requirements Rationale

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.

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. NASA Procedural Requirements for Systems Engineering Processes and Requirements (NPR 7123.1A)[14] 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)[15] *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.

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.

C. Emphasize Trade Studies

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*[16] at Defense Acquisition University lists trade studies as one of the practices having the most evidence. 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."[17]

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.

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 Recommendation A, namely, lack of awareness of downstream complexity. Also, as Recommendation G 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.

D. More Early Analysis and Architecting

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.

Finding

The study team found evidence of weak software architecture, which 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.[18] 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. As the famous computer scientist

Alan Kay said, "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.

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[19], reflecting industrial experience over a wide range of software projects, supports the value of applying resources to architecting. As Figure 5 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. As the figure shows, larger systems require larger investments in architecture in order 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 Recommendation I for investing in "reference architectures." 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.

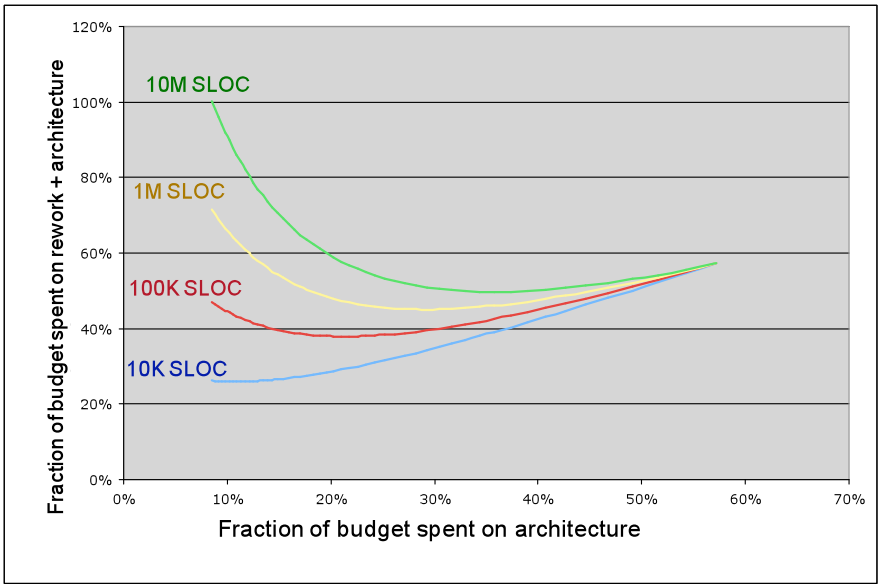

Figure 5. Equations from the COCOMO II model for software cost estimation show that the "sweet spot" for investing in software architecture increases with software size.

E. Establish Architecture Reviews

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.[20]

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[16] 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.[21] 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.

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*[20] provides an excellent description that includes principles, participants, process, artifacts, lessons learned, and organizational benefits. 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"[22] and "Peer Review Inspection Checklists."[23] Also, NASA could strengthen NPR 7123 regarding *when* to assess software architecture.

F. Nurture Software Architects

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.[8] 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.

Finding

The two preceding recommendations (D and E) 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 course on software architecture. During a span of 10 years, the program graduated several hundred students, with students completing a project component 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.

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.

G. 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.

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 was 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. Although each operational workaround is typically a small increase in complexity, 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.

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".[24]

H. Analyze COTS Software for Verification Complexity

Finding

Commercial off-the-shelf (COTS) software is sometimes a mixed blessing: it 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.

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.

I. Invest in Reference Architectures

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".[25] 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.

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."[26] 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 E) since it will shape the architecture of many missions.

J. Stimulate Technology Infusion

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.

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.

K. Use Static Analysis and Code Compliance Checkers

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.[27] 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.

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.

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.

L. Standardize Fault Protection Terminology

Finding

A recurring observation during this study and in the NASA Fault Management Workshop[31] was the lack of consistency in fault protection terminology and its use in software frameworks. 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. 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 (e.g., "macro"). Further, the industry lacks good reference material from which to assess the suitability of different fault protection approaches.

Recommendation

NASA should work with the aerospace industry to develop and publish a NASA Fault Protection Handbook or Standards Document that provides a: (a) standard lexicon for fault protection and (b) 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.

M. Establish Fault Protection Proposal Review

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.

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 (Recommendation L) can be used as a reference to aid in the assessment and provide some consistency.

N. Develop Fault Protection Education

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.

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.

O. Research Fault Containment Techniques

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?

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. The recommendation here is to fund research to apply this approach on the most suitable areas of flight software.

P. Collect and Use Software Metrics

Background

Any 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 (Section III.A) and devoted most of our time 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?

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 recommends "software measurement programs at multiple levels should be established to meet measurement objectives" with respect to planning and cost estimation, progress tracking,

[**] The practice known as "test as you fly and fly as you test" means that a flight system should be tested in the way it is planned to be operated, and it should be operated within the envelop of what has been tested. This conservative practice guards against unanticipated behavior in software and hardware.

software requirements volatility, software quality, and process improvement. However, these recommendations are not specific and not mandated.

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.

V. Fault Protection Architecture

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. Recommendations L, M, N, and O highlight several broad areas for improvement.

The conventional approach treats fault protection software as an add-on to a sequencing system designed for nominal execution. The problem with the add-on approach is that it introduces unnecessary complexity. A conventional 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 with less complexity by deeply integrating fault protection into the nominal system? This study was directed to answer these questions, and did so in a publication by Robert Rasmussen.[28] The paper goes back to fundamentals—the systems theory of control—to explain 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 with nominal control, thereby simplifying engineering and testing. The paper is very readable and highly instructive about how systems engineering and software architecture go hand-in-hand. The paper offers a prime example of the importance of architectural thinking and the value of architectural principles and patterns to shape software design.

VI. Historical Perspective

Software complexity 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 attended, all concerned professionally with software, either as users, manufacturers, or teachers at universities. 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 paper expressed alarm about "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. 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. What was once considered 'complex' became routine because we learned better ways, but our ambitions have also grown, creating new challenges that today seem 'complex'.

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."[29]

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:

> "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."[30]

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.

VII. Conclusion

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), including 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.

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 protection was given special attention because it is among the most complex software to design and verify.

This study generated 16 recommendations, some of which are relatively simple and some of which require investment and training. 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 often built on heritage software that, in some cases, 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 shape both 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 architectures, and steps to grow and promote software architects.

Acknowledgments

This study was designed to draw upon expertise at five institutions responsible for space flight software: Goddard Space Flight Center, Jet Propulsion Laboratory, Johnson Space Center, Marshall Space Flight Center, and the Applied Physics Laboratory of Johns Hopkins University. Approximately 45 engineers and managers participated in discussions held at these institutions, giving the study a broad perspective on "complexity" and corroboration of common issues.

GSFC contributors include Harold "Lou" Hallock, Kequan Luu, Manuel Maldonado, Jane Marquart, David McComas, Jonathon Wilmot. JPL contributors include Kevin Barltrop, John "Brad" Burt, Len Day, Daniel Dvorak, Lorraine Fesq, Gerard Holzmann, Cin-Young Lee, Robert Rasmussen, William "Kirk" Reinholtz, Glenn Reeves. JSC contributors include Mike Brieden, Brian Butcher, Rick Coblentz, Pedro Martinez, Carl Soderland, Emily Strickler. MSFC contributors include Darrell Bailey, Pat Benson, Terry Brown, Keith Cornett, Robert "Tim" Crumbley, Helen Housch, Steve Purinton, Steven "Gregg" Taylor, and Helen "Leann" Thomas. APL contributors include David Artis, Brian Bauer, George Cancro, Debbie Clancy, Bob Davis, Jacob Firer, Adrian Hill, Chris Kupiarz, Roland Lang, Horace Malcom, Aseem Raval, Mark Reid, Steve Williams, and Dan Wilson.

I offer special thanks to my primary technical contacts at the 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 biweekly teleconferences and organized visits at their respective centers where much of the key information was collected that led to the 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).

I also thank John Kelly, OCE Program Executive for Software Engineering, and Tim Crumbley, co-chair of the NASA Software Working Group. Both served as technical advisors and offered constructive feedback and helpful suggestions. I am also indebted to three 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). Each had prepared an analysis of flight software complexity that seeded this study, and I gained many insights from discussions with them. In addition, Kevin Barltrop (Autonomy and Fault Protection Group) deserves special recognition for leading the entire examination of complexity in fault protection and authoring the appendix on that subject in the final report.

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. Chi Lin, Manager of JPL's Engineering Development Office, served as program manager for this study and provided enthusiastic support for the study and its recommendations.

Finally, I wish to thank NASA's Office of Chief Engineer and the individuals who initiated and funded this work under the Technical Excellence Initiative. NASA Directorate Chief Engineers Frank Bauer (ESMD), Stan Fishkind (SOMD), Ken Ledbetter (SMD), and George Xenofos (ESMD Deputy Chief Engineer) identified the importance of

a study on this topic, so without their leadership this study would not have occurred. Adam West managed the study and was always helpful in answering my questions and offering guidance.

Part of the work described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, and at Applied Physics Laboratory, John Hopkins University, under a contract with the National Aeronautics and Space Administration.

References

[1]Aldiwan, W. and Wang, J., "Flight Software 101: Ten Things JPL Engineers Should Know about Flight Software," Flight Software and Data Systems Section, JPL/Caltech (unpublished).

[2]Dvorak, D., editor, "NASA Study on Flight Software Complexity," Final Report submitted to NASA Office of Chief Engineer, March 2009, http://oce.nasa.gov

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

[4]Gillette, W., Vice President of Engineering, Boeing Commercial Airplanes. Presentation at 17th Digital Avionics Systems Conference, November 1998.

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

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

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

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

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

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

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

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

[13]Charette, R. N., "This Car Runs on Code," IEEE Spectrum, Feb. 2009.

[14]NPR 7123.1A, NASA Systems Engineering Processes and Requirements, http://nodis3.gsfc.nasa.gov/lib_docs.cfm?range=7___

[15]NASA Systems Engineering Handbook, NASA/SP-2007-6105 Rev1, December 2007, http://education.ksc.nasa.gov/esmdspacegrant/Documents/NASA%20SP-2007-6105%20Rev%201%20Final%2031Dec2007.pdf

[16]Defense Acquisition University, "Best Practices Clearinghouse," https://bpch.dau.mil

[17]Ferguson, J., "Crouching Dragon, Hidden Software: Software in DoD Weapon Systems," *IEEE Software*, vol. 18, no. 4, pp. 105-107, July/Aug. 2001.

[18]Shaw, M. and Garlan, D., *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

[19]Center for Systems and Software Engineering, University of Southern California, COCOMO II, http://sunset.usc.edu/csse/research/COCOMOII/cocomo_main.html

[20]Maranzano, J., Rozsypal, S., Zimmerman, G., Warnken, G., Wirth, P., and Weiss, D., "Architecture Reviews: Practice and Experience," *IEEE Software,* March/April 2005.

[21]Clements, P., Kazman, R., and Klein, M., *Evaluating Software Architectures: Methods and Case Studies*, October 2001, Addison-Wesley.

[22]Weiss, K., "Software Architecture Review Process," internal document at JPL/Caltech (unpublished), 2008.

[23]Schuler, P., "Peer Review Inspection Checklists," internal document at NASA Langley Research Center (unpublished).

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

[25]Reed, P., "Reference Architecture: The Best of Best Practices," September 2002, http://www.ibm.com/developerworks/rational/library/2774.html

[26]Wilmot, J., "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

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

[28]Rasmussen, R. D., "GN&C Fault Protection Fundamentals," Proceedings of 31st Annual AAS Guidance and Control Conference, AAS 08-031, American Astronautical Society, Feb. 2008.

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

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

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