American Astronautical Society
Rocky Mountain Section

AAS 08-031

GN&C Fault Protection Fundamentals

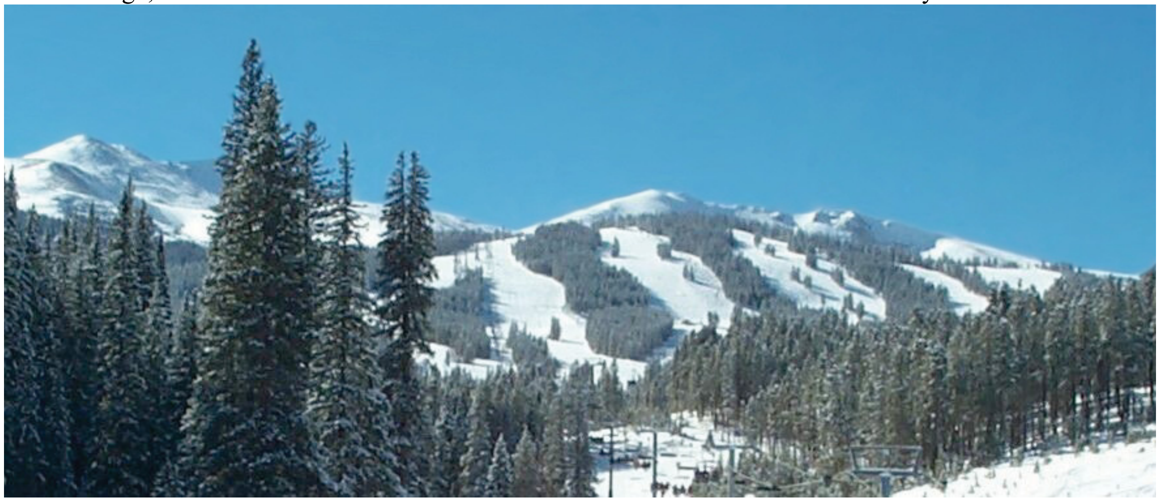
Robert D. Rasmussen

Jet Propulsion Laboratory, California Institute of Technology

31st ANNUAL AAS GUIDANCE AND CONTROL CONFERENCE

February 1-6, 2008
Breckenridge, Colorado

Sponsored by
Rocky Mountain Section



AAS Publications Office, P.O. Box 28130 - San Diego, California 92198

AAS 08-031

GN&C FAULT PROTECTION FUNDAMENTALS

Robert D. Rasmussen[*]

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.

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.

[*] Jet Propulsion Laboratory, California Institute of Technology

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[*] elements themselves — the integrated control functions, active in a live system, that make the difference between failure and success, when things go wrong.

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

[*] 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.

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.* 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 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 1 (next page).

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.

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

* 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.

- 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 1 Issues Routinely Complicating Fault Protection 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.

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.

CORE CONCEPTS

The conceptual roots of modern GN&C lie deep in systems theory*, 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.

Basic Ideas and Terminology

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.

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

* 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.

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.

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.

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.

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*.[1]

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.

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.

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.

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.

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.

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.

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[2], 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"[*] 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.

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

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 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.[3] 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.

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.[4,5,6]

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.

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.

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.