Goal-Based Fault Tolerance for Space Systems Using the Mission Data System[1]

Robert D. Rasmussen
Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive, Pasadena, CA 91109-8099
(818) 354-2861
robert.d.rasmussen@jpl.nasa.gov

Abstract—In anticipating *in situ* exploration and other circumstances with environmental uncertainty, the present model for space system fault tolerance breaks down. The perplexities of fault-tolerant behavior, once confined to infrequent episodes, must now extend to the entire operational model. To address this dilemma we need a unified approach to robust behavior that includes fault tolerance as an intrinsic feature. This requires an approach capable of measuring operators' intent in the light of present circumstances, so that actions are derived by reasoning, not by edict. The Mission Data System (MDS), presently under development by NASA is one realization of this paradigm — part of a larger effort to provide multi-mission flight and ground software for the next generation of deep space systems. This paper describes the MDS approach to fault tolerance, contrasting it with past efforts, and offering motivation for the approach as a general recipe for similar efforts.

TABLE OF CONTENTS

1. INTRODUCTION

Fault tolerance and its attendant operational complexity have always been problems for space missions, requiring a tense balance between tight ground control and flight system autonomy. Ability to predict and certify every action has been the hallmark of most operational models, with reaction to faults considered as a necessary but disruptive and potentially dangerous interference. While very expensive, this approach, nevertheless, has been generally manageable — until now.

In anticipating *in situ* exploration and other circumstances with significant environmental uncertainty, the present model breaks down. Operation in dynamic or unpredictable situations will become common. Thus the perplexities of fault-tolerant behavior, once confined (with luck) to infrequent episodes, must now extend to the entire operational model. What's more, these behaviors must remain affordable — or better yet, reduce costs below the present norm.

This is one of several objectives undertaken by the Mission Data System (MDS), a unified flight, ground, and test architecture for the next generation of NASA's deep space systems presently under development at the Jet Propulsion Laboratory. MDS will be adaptable to a wide range of space systems, including interplanetary and orbital missions, small body explorers, surface rovers, aerobots, formation flying interferometers, observatories, and so on. All of these systems, in one way or another, can benefit enormously from a system that reacts to its circumstances and behaves appropriately (though not necessarily predictably) to meet its operators' intent. This is the essence of fault-tolerant behavior, extended to the general issue of performance and survival in an uncertain environment.

To accomplish this, MDS has been composed of a full set of object-oriented software frameworks, and of equal importance, a set of systems engineering tools and methodologies. The frameworks provide a basis for reusable, reliable software grounded both in solid software engineering principles and in a set of architectural themes [1] tailored to achieve robust behavior. The systems engineering approach supports an expression of system design and operational considerations that is natural to designers, yet directly interpretable in software. It is this essential merger of software and systems engineering that MDS claims as its fundamental contribution, not just to fault tolerance, but to the whole problem of safer, more reliable, and lower cost space systems.

2. SPACECRAFT FAULT TOLERANCE TODAY

Spacecraft fault tolerance is not a single entity. It arises from characteristics of the hardware, such as intrinsic

[1] 0-7803-6599-2/01/$10.00 © 2001 IEEE

reliability, redundancy, provisions for fault masking or containment, physical robustness against adverse conditions, interlocks on critical functions, adequate margins, and so on. It also depends on various schemes to detect problems and react to them soon enough to preserve system safety and functionality. Such protection systems are themselves generally divided into layers.

At the top is adherence to some operational model, which one trusts will guide the system through its tasks with tolerable risk. To date, this has been for the most part a timed sequence of commands — predictable events at predictable times. Conditional behavior in such sequences, while not unheard of, is generally the rare exception. Such behavior is more typically restricted to a supplemental layer below the sequence, implemented in state machines or similar embodiments of sequential behavior, but always within a strict envelope of performance that planning efforts can predict. There will be much more to say on this matter below.

Barring unpredictable events outside the envelope of nominal variations, conventional time-based sequencing serves the system well, giving operators just what they expect. In fact, much of the operational model revolves around the detailed verification of "predicts". For anything unusual, however, the system must take care of itself. High level "fault protection" software handles the bulk of this responsibility, but it depends on lower level software to assure an operational computing platform, which in turn relies on protection in the hardware itself for power and other fundamentals.

In this complex collection, many competing factors are at work. For instance, missions are often characterized by one–time opportunities or by a heavy workload that demands very high availability to accomplish mission objectives. This often dictates a compressed, carefully optimized sequence of actions, where each activity is generally hard won against concerns over tight margins. Therefore, any contemplation of uncertainty, any lack of guarantee that these planning conquests won't be squandered by a system that cannot promise each and every item its turn, is regarded with deep suspicion. Nevertheless, there *are* no guarantees — only the disquieting certainty that everything will go exactly as planned, if nothing unplanned happens. This is but a hollow victory, for the potential of tripping into a fault response threatens every step.

Given this predicament, designing system fault tolerance for space applications has been difficult. It *has* steadily improved over the years. However, as currently practiced, it remains a daunting, protracted labor for each new project to bring to a level of maturity and reliability that is adequate to the task.

This is especially so when critical mission activities are at stake. If time or resources are constrained, then recovery actions inevitably conflict with the urgent need to push forward to a successful end — without ground intervention.

Even in less stressful conditions, where it is possible (though generally undesirable) to forego normal operations for the sake of system safety, there is a struggle between assuring adequate fault coverage and satisfying the misgivings of operators over finicky or unruly fault tolerance systems. False alarms can be highly disruptive to operations, but failure to respond adequately to genuine faults is potentially calamitous. Thus, when the only options are failure to perform versus failure to survive, the choices available are frequently disappointing.

Operators also fear outright incorrect, or even dangerous, behavior anytime they relinquish control to automatic functions, and they find it irksome to realize that their judgement is most facile in situations where the system needs it least. That is, that which they trust the least to manage routine operation is that which they *must* trust when things get complicated. This is a fundamental irony in present techniques, clearly indicating that we have taken a blind alley in the evolution of fault tolerance.

Until now, these struggles have ordinarily resulted in a satisfactory but wary compromise for all interests, but not without great cost. Moreover, new, more challenging planetary missions threaten to push even this expensive concession out of reach. In essence, space systems are moving into regimes where the unexpected will be routine, so the line between fault tolerance and generally robust behavior blurs. Attempting to carry old fault tolerance paradigms into the broader functioning of the system is simply impractical.

3. A Brief Overview Of MDS

Fortunately, there is a resolution to this dilemma, but it requires the unification of fault tolerance with all other aspects of system operation in a single operating paradigm. Furthermore, any realization of this paradigm must allow overt expression of the operator's intent. It must be explicitly cognizant of system state and the integral situation of failure modes within this context. Also, guidance of the system must be further informed by models of behavior and operational constraints that provide the basis for reasoning about alternative courses of action.

State Variables[2]

At the core of MDS, then, is the notion of a state variable — an object of a formal class within which all software knowledge of some external system state is captured. The collective state of the whole system, including the

[2] It is important to observe the distinction in this context between a software state variable and the state of the software itself. State variables are software objects that capture knowledge of the state of the *external* system. The software itself may have internal modal behavior or other state, but state variables do *not* refer to this software state. That is, state variables are not introspective.

environment and relevant states of other interacting entities, is captured in a set of state variables. A typical system may require hundreds of them in order to express all knowledge pertinent to its function. The key, however, is that this knowledge is maintained in an internally consistent form and all software actions derive solely from this information.

State variables span a wide range, from obvious items like attitude and device states to less obvious minutiae like mass properties and calibration parameters. They attend to system resources, such as power or propellant. They keep track of what data have been collected from science instruments and what the next downlink opportunity might be. They follow the moons and planets of the solar system. They can even represent the dusty winds on Mars — whatever the system needs to know.

Need is the governing factor on complexity. All state variables represent state in a manner befitting the needs of the system. Thus, in the final analysis, one finds that the information provided would have been available in some form anyway. A benefit of overtly formalizing state knowledge within the MDS architecture is that it leaves no room for questioning what indeed the factors are that govern system behavior, or whether or not they collectively make sense — a surprisingly difficult thing to discover in most other systems.

MDS also encourages a statement of this knowledge in clear and direct terms, rather than in myriad flags, counters, parameters, and the like, scattered indiscriminately through the code. For instance, one might give up retrying a command, not because some counter has incremented to the fatal limit, but rather because the commanded device has been ascertained to be in an unresponsive state. Thus, state variables tend to be much more descriptive. (Of course, this begs the question of how the state value is determined. This is described below.)

Knowledge Uncertainty

Another aspect of state knowledge promoted by the MDS architecture is that all knowledge is suspect. Every state variable, therefore, is required to declare its degree of certainty in any knowledge it provides. In fact, a state variable is never without an answer, even when it has no knowledge. It merely declares its knowledge to be completely uncertain.

The inclusion of uncertainty is essential for two reasons. First, such information is usually crucial in making proper decisions. What is appropriate when a value is well known may be totally inappropriate if the same “best” value is highly uncertain. One might not risk firing an engine, for instance, even if the most likely values of pressure and temperature were safe, if the envelope of uncertainty extended beyond the safe limits.

The other reason for including uncertainty in state representations is that uncertainty is often the principle target of intent in a system rather than the value of state itself. The operation of sensors, for instance, is motivated foremost by the need for useful (i.e., relatively certain) state knowledge, and only secondarily by how this information will be used.

Estimation and Control

These two aspects of state knowledge result in two basic aspects of system functionality: estimation and control. Estimation deals with the determination of state knowledge and is motivated by how good this knowledge must be. Control deals with the manipulation of state, as represented in state knowledge, and is motivated by what value this knowledge must attain.

In MDS, these functions are cleanly delineated. Nothing about the intended value of state is allowed to influence the estimated value or its uncertainty. Likewise, none of the evidence (measurements and observed commands) collected for state estimation is used directly by state controllers. They react only to the estimates themselves. This separation helps assure the honesty and expressiveness of state knowledge, and the consistency of action among controllers. Moreover, it improves the modularity and reusability of software components.

Goals

Estimators and controllers in MDS are also formal classes. Together they (and a couple others not described here) comprise a set of so-called goal achievers. This is because all actions in such a system are directed by goals, *not* commands.

The distinction between commands and goals is essential and marks one of the greatest departures of MDS from conventional approaches. A command is momentary. Any lasting effect it may have is due to functions or behaviors expressed elsewhere in the system. Thus, one cannot tell from a command alone whether it contradicts the intent of its predecessors.

Suppose one commands a device to some state, for instance, with the intent to leave it in that particular state for at least five minutes. It is possible that no change of state is commanded after this until necessitated by a much later activity, even though it wouldn’t matter after five minutes, as far as the operators were concerned. Yet there is nothing in the sequence of commands itself that declares such intent. Moreover, there is nothing in the timing of commands that indicates what flexibility might be possible in these times. A fault response, needing to change this state, would have no way to tell if it is creating a problem or how long it would have to wait to avoid one.

These issues are at the root of incompatibility between command sequences and fault protection. Absent any way to discover intent or flexibility, fault protection is reduced to assuming the worst, and that usually means terminating the sequence.

MDS resolves this dilemma by expressing intent explicitly as constraints: constraints on the value of state (or its uncertainty!) over a time interval, and constraints on the instants in time that demarcate these intervals. A constraint is a condition on value, expressible in concrete terms (such as with an inequality) that must be true.

State constraints with their associated time intervals are called goals. A collection of goals and temporal constraints comprise a goal network that unambiguously expresses operator intent in a declarative manner that supports reasoning and that affords the system flexibility in response, which is unattainable in conventional space systems. Whether or not two goals are in conflict is simply a matter of comparing their state constraints and the temporal constraints on their time intervals. Where conflicts do arise, the system is free to try alternative actions and timing that meet the constraints, because the constraints generally lay down a broad set of possibilities rather than a specific sequence of actions and times.

Closing the Loop

Commands are derived from these constraints, given any discrepancies between desired state and current state knowledge. Goal achievers, as described above, are the means to effect these commands. Thus commands do appear, but only indirectly as a byproduct of goals, not as the first order means of system direction.

The specific set of commands selected is determined by circumstances, given the goals, so commands are not fully predictable. What *is* predictable, at least to the same extent that timed commands have been, is that the constraints will be met. In reality, however, a goal-driven system is much more likely to produce the desired results because of its ability to try alternative actions and timing. Conventional systems, faced with problems, will usually just abandon sequenced actions and wait for help.

Expressiveness

Constraints may seem, at first, like a limiting way to direct a system. One might ask, for example, how one could use constraints to direct a series of science observations. One can't simply say that the goal is to *do* the observations. That, after all, is just a command, not a state constraint. So how is it done? There is a simple answer that, nevertheless, often generates surprise when first encountered. One merely determines what one wants to be different about the system after the observations from what existed before. The difference in this case is clearly that the desired data exists in storage afterward. If it already existed, the observations would be unnecessary. This is the change of state desired, so we represent the contents of data storage with a state variable, and impose a goal (i.e., state constraint) upon the system to be in a state where the required observation data has been stored.

Another seemingly peculiar situation arises when considering resource allocation and similar operational situations where only indirect control is possible. One wants to describe, not what set of values a state is allowed to take, but rather what variability a state must be guaranteed at a minimum. At first, this seems to violate the notion of a constraint, but after some reflection, it is clear that both ordinary constraints and resource allocations have something to say about what states are allowed. Ordinary constraints merely bound this allowed set from above, while allocations bound it from below! With this subtle generalization, the entire question of resource management is also subsumed within the vocabulary of goals.[3]

Data transport, navigation, system safety, and so on can all be managed in a similar manner. Thus we see that the language of goals is not only very expressive, but also that even the highest level of system operations can be directed in a closed loop fashion. The power to handle both faults and normal activities within the same operational model emerges from this ability.

Elaboration

Of course, there are a few steps between expressing a high level constraint and actually performing the low-level actions that bring it about. This process in MDS is called elaboration. A set of rules recursively expands the high level goal into a goal network that is merged with the outcome of previous elaboration. Because constraints can be compared for conflict, everything can be arranged to meet all the constraints unless it is logically impossible (or computationally intractable) to do so. When unresolvable conflicts arise, additional rules determine the precedence of goals in order to find an acceptable subset that *can* be resolved in a timely manner. This is described in more detail below.

Data Management and Transport

One final note about the general MDS architecture is in order before turning to the specific issue of fault tolerance. Like almost everything else in MDS, data management and data transport are entwined in the notion of state. Besides current state knowledge, both past and future (i.e., experience and plans) are maintained in a timeline for each state variable. Factors that govern the persistence (e.g., through reset), quality (e.g., after compression), and transport (e.g., priority) of this data are all associated with the state variables through policy mechanisms that are also subject to direction by goals.

Notice also that in none of this discussion has the distinction been made about where goal elaboration or achievement occurs — flight or ground. In fact, these functions may be distributed, as dictated by needs and feasibility, across both

[3] Moreover, this enables a vital mechanism (not described here) for the link between functions handled directly by goal networks and those delegated to subordinate goal achievers.

locations (and any other collection of systems). This is made possible by the sharing of state variable timelines in both places and by communication across the link of goals, measurements, and other common entities that also flow within each realm. This is accomplished by data management and transport capabilities dedicated to this end. In fact, most engineering activity on the link, in both directions, may be properly thought of as merely bridging communication between two parts of the same system. Thus, MDS is truly a unified architecture.

4. MDS FAULT PROTECTION

With this description in mind, it is now possible to show how fault protection fits naturally within the larger framework of MDS, not as an additional set of functions, but rather as a logical extension of the core ideas. As with most things in MDS, we start with the notion of state.

Failure Modes

A failure is a possible condition (state) of a device in which it does not perform as designed. Failure modes are sets of failures that share a common cause or symptom. These failure modes, together with the conditions in which the device *does* perform as designed comprise the total set of possible states for that device. Therefore, MDS treats failure modes and other anomalous conditions as just another possible set of system states. All such potential states must be included among the possible values of the collection of state variables defined for the system. State variables are augmented or added, as necessary, to make this possible. Thus, one might say, for instance, that a valve is either open or closed and working normally, or that it is stuck in one of these two positions — four possible values for one state variable, two of which are failure modes.

In these state variable representations, failure modes need not be described explicitly. Rather, any portrayal that captures essential observable behavior is sufficient. In particular, what is most important is the circumscription of normal or acceptable behavior, such that departures can be ascribed to anomalous states. This allows the system to categorize any deviant behavior, even if it arises from unknown failure modes. One may know, for instance that a linearly dependent set of gyros has a serious problem if a parity test among them is violated without ever having a clue about the source of the error. The state variable can categorize this as "not right" even before any more particular diagnosis is made.

Failure categories are chosen to discriminate mainly by their effect, rather than by their cause, unless knowing the cause determines what corrective actions might be possible. For example, a valve may be stuck for a variety of mechanical or electronic reasons, but one needn't know which, if it is deemed permanent. On the other hand, it is usually important to know whether a valve is stuck open or stuck closed, and not just that it is stuck.

Fault Monitoring

State determination collectively and dynamically chooses an estimated current state for each state variable. These estimates are chosen to best fit the observed evidence (measurements and commands), given state–based models of behavior maintained by state determination processes.

State-based models include behavior of sensors, data chains, and commanding mechanisms. Therefore, the potential for corrupted measurements and commands, or interrupted data flow is taken into account by state determination.

Fault detection occurs whenever observations do not adequately match modeled behavior if nominal states are assumed. That is, if it is necessary to alter the estimated state to one of the fault states in order to better explain the observed evidence, then a fault is assumed to have occurred.

There may be more that one possible explanation when a fault is detected. In this case, the ambiguity is reflected in the uncertainty associated with every estimate.

To illustrate, consider a situation where a command has been issued to open a valve, but the status measurement from the valve indicates that it is still closed. This is a departure from modeled normal behavior, but it is consistent with three different failure models: a failure to deliver the command (or failed valve driver), a stuck closed valve, or a faulty valve position sensor. Barring further information, the estimated states would be "possibly broken command chain", "unknown valve state", and "possibly failed valve position sensor". That is, each state variable would report some degree of uncertainty, each with the possibility of a failure.[4]

If, subsequently, additional information arose that the valve was indeed open (perhaps due to no observed pressure difference across the valve), then the estimated states would resolve to "command chain okay", "valve open", and "definitely failed valve position sensor" as the only modeled behavior fitting all observations without assuming two simultaneous but independent failures.

Note that no decision regarding potential action is made in any of this determination. This is consistent with the separation of state estimation from state control.

Note also that the representations of state are quite explicit, whereas in conventional designs one will often see such conditions appear only as momentary expressions evaluated in conditional code branches. Separation of state estimation from state control fosters the explicit form, which greatly

[4] Of course, much more quantitative descriptions are possible, based on Bayesian methods or other approaches. However, the simple enumerated possibilities, used here for illustration, may also be used, if appropriate.

clarifies the assumptions going into the design, making them open for easier inspection.

Finally, note that the failure values of state are totally within the same representation as normal values. Fault detection is not a separate function from normal state estimation, nor is it ever possible for one part of the system to react to a detected failure, while other parts go about business as though everything were normal.

Flight Rules

Goals specify constraints imposed on the state of the system. Most goals are in service of operational objectives. Others, however, embody flight rules and constraints. One might require, for example, that an instrument never be pointed close to the sun while the spacecraft is inside Jupiter's orbit, or that power margin never be allowed to fall below 10 Watts, or that high voltage power supplies remain off during launch vehicle ascent. Each is a constraint on system state that must be assured, and each may have a prescribed interval of applicability, just like any routine operational objective. That is, flight rules and constraints are just business as usual. Thus, in MDS, this important aspect of operations is neither outside normal activities nor imposed upon them as a separate step. Rather it is simply a coequal part within the total process of expressing operator intent.

Goals are achieved by controlling their associated states (when possible) through goal achievers, and through a process of goal elaboration, which creates additional subgoals designed to enable or facilitate the achievement of the original goals. Elaboration relates subgoals to parent goals and to one another in such a way that dependencies among them are enforced. These dependencies include access to all necessary resources via allocations. Elaboration also imposes additional temporal constraints, as necessary to properly order actions and motivate completion at appropriate times.

Most subgoals support goal achievement, either through further elaboration into subgoals on states that affect the target state, or by directly commanding controllers of the target state. In addition, there are often supplementary subgoals on knowledge quality. A case in point is a goal to maintain some temperature within a narrow range. If knowledge of the temperature were uncertain by more than this range, then there would be little hope of meeting the control objective. In any event, the goal would be declared unsuccessful, because success cannot be determined.

Subgoals on knowledge quality assure that states are sufficiently well known to permit declaration of success by the originating goals. The essential role of these knowledge goals is to configure the appropriate sensors for the task through further elaboration of their own. A supplemental subgoal on temperature knowledge uncertainty results in steps to configure sensors that make the required accuracy possible, so in the example above one would find this a necessary part of the elaboration.

Knowledge goals are an important component of fault tolerance in MDS, because actions to disambiguate fault indications generally arise from goals to have *un*ambiguous knowledge. This is discussed further below.

Flexibility

As described, MDS gains much of its strength by specifying intent flexibly, in order to give the system room to explore alternative actions. Therefore, an important aspect of goal network designs is that they be flexible, to the extent possible, in the constraints they impose. For instance, it is often the case that temporary outages or delays in service are acceptable. (As a case in point, the outage of gyro data for a few seconds is rarely a serious issue.) Moreover, many performance thresholds are soft. Why should an error of 2.000 milliradians be acceptable, for instance, while 2.001 milliradians is not — just because someone set a threshold at this value? In both conventional designs and in MDS this is handled by more sophisticated tests. Consequently, constraints might consider both the severity and persistence of deviation, allowing for a variety of acceptable excursions.

What MDS adds is the ability to make such criteria much more dependent on the context of other activities. Thus the constraint on some state that determines whether it is deemed faulty can be readjusted with every elaboration — never more tightly than necessary just because some worst case had to accommodated.

Safety

The most perseverant and flexible of goals is that responsible for general system safety. It is the source, through elaboration, of the more particular constraints against system hazards that populate every goal network. Such hazard avoidance goals might, for instance, specify safe temperature ranges outside what might be necessary for good performance; or they might require that at least one receiver chain be operational at all times, even when no uplink is expected; or that pressure regulators be isolated when propellant flow rates are low, whether or not a leak is detected.

The safety goal that hosts these hazard avoidance goals would never declare failure, since to do so would involve conditions in which the software would likely not be running in the first place. Thus, although it may set limits of tolerability, it nevertheless suffers the arbitrarily long persistence of danger without giving up. It is the goal, therefore, that always has more alternative elaborations to explore and which is always able to trump other goals in the system with higher priority, as required, in the attempt to regain the upper hand and restore the system to safe operation (commonly referred to as "safing").

Fault Responses

Goals monitor their associated states for adherence to the constraints. When a fault occurs, the system state will generally change to one in which one or more state constraints is either now or soon will be violated. This can be either because the state is believed to have strayed from its allowed range, or because the knowledge of the state has degraded such that it is no longer possible to tell for sure whether or not the constraint is satisfied. Either possibility is problematic.

State determination should recognize and report this change, at which time goals monitoring these constraints will respond. This response may be to attempt an alternate method of achievement, if one exists and there is time to attempt it. Otherwise, the affected goal *and all its subgoals* are abandoned, and an immediate declaration of failure is made to a parent goal where responses of greater scope are possible. All goals share this failure behavior. Responses are escalated in this way to the appropriate level, while simultaneously simplifying the context of the response. This has important consequences for the ability of the system to respond in a timely and uncluttered manner.

In selecting an alternate achievement method, responses may need first to perform actions that resolve ambiguous estimates. For instance, the system configuration may have to be altered in such a way that additional, appropriately discriminating information becomes available. Suppose, for instance, that a sun sensor ought to have seen the sun, but didn't. Is the sensor at fault, or is it simply not pointing where the attitude estimate says it is? Turning on a second sun sensor will tell for sure. As described before, such responses are generally motivated by subgoals on knowledge quality.

One type of alternate achievement method is the invocation of block redundancy. This is possible when a goal specifying the availability and health of some resource does not explicitly constrain which redundant element may be used to fulfill the goal. In elaborating this goal, a choice may be made among the remaining healthy resources, and this is accomplished via a subgoal for a *specific* element. Failure of this element results in failure of the subgoal, but not of its less particular parent, which may then re-select among the remaining healthy redundant elements via a new subgoal.

Alternate achievement methods need not always be so final relative to the deviant element. Re-commanding, reset, or other actions against the originally selected element may be sufficient to clear a fault condition. These, too, may be manifested as subgoals. For instance, it may be necessary to cycle power. This could be accomplished by a pair of subgoals. However, in localized cases, responses are often delegated to controllers, which can respond directly and more quickly to the observed state and adjust their actions accordingly.

In all of these alternate method responses, allowance for the time necessary to perform the switch must be granted via flexible goal specifications, as described above. Similarly, where block redundancy has not been provided, or no redundant element remains, alternate methods of achievement with degraded capabilities may still be available for selection, if goals are flexible.

The patterns described here repeat over and over throughout the design. Unlike conventional approaches, however, each application can be considered in relative isolation — all interactions that typically bedevil fault responses being handled through the normal coordination functions provided automatically to every goal network. One may anticipate, therefore, a much more rapid convergence to a robust design.

Hazard Avoidance

Some faulty units may be a hazard to the system in their discovered state. However, a typical goal is happy in its elaboration simply to acquire an alternate set of capabilities that supports its objective. Otherwise, it just gives up. Either way it typically leaves the faulty unit to its own ends. To keep things safe and tidy, there are, among the goals in the system, goals to avoid hazardous states, as described previously. Such goals are generally either passive, since in a healthy system normal goal elaboration in conjunction with resource management (via allocation goals) would seldom select such states, or they merely enforce conservative behavior, stepping in with cautious direction only when no other factors dictate a particular configuration.

If a hazard violation were caused by a fault, however, the affected hazard avoidance goal would be threatened, and with no other line of defense from active fault recovery. In such situations, if allotted sufficient flexibility, the hazard goal itself would respond to the situation, for instance by isolating or disabling the faulty element. A simple case is illustrated by a device in a battery-powered system with an internal short circuit that leaves it inoperable. The main response may have turned on a backup unit, but the faulty unit remains a hazard both for drawing excessive current and for reducing energy margin, even though both might still be within allocation. One would rely on hazard goals to turn off any unnecessary or faulty devices. Hazard avoidance goals would also take steps to recover margin.

Suppose, however, that nothing could be done to recover a safe margin. Perhaps other functions using energy are simply too critical to perturb further. They may have been able to resume after the fault, but hazard avoidance has failed. In a worst case like this, when their responses are insufficient, it is ultimately the responsibility of the system safety goal, as parent, to attend to failures of hazard avoidance goals, imposing more extreme responses to attempt recovery. There is no appeal beyond this level until ground intervention is possible, so this is where one decides out of desperation, for instance, whether to pursue a

hopeless goal into likely oblivion, or to let the opportunity pass with some hope at least of reporting what went wrong.

Goal Networks

Fault responses will often conflict with other activities, including other fault responses. Since all system objectives are expressed as goals, this conflict manifests itself as conflicting constraints. That is, two or more goals may attempt to establish different constraints on the same state variable at the same time, and no single value for state can satisfy them all. A typical situation will find a goal present to support some operational objective in conflict with a goal to establish a safe condition or to restore some resource.

Conflicts such as this may be resolvable by postponing or rearranging the order of activities, possibly retrying activities that were interrupted. This depends on how much temporal flexibility is provided in the goals. In such cases, normal activities would resume as soon as the fault was cleared, completing normally, albeit delayed.

Otherwise, the conflict is not resolvable unless some goals are removed. Only those goals immediately affected by a conflict would be in jeopardy. Later goals, after the situation has been resolved, would be left in place unless they depended in some way on dropped goals. In this way, most planned operations should proceed, depending on the severity of the incident.

Priority determines which goals win in such conflicts. Priorities of subgoals derive from their parents, so problematic situations, such as two high priority goals each losing key low level subgoals so that neither can succeed, are avoidable.

Priorities will be set by adapters to suit each mission's needs. A representative pattern would have immediate safety goals at the fore, followed by resource preservation goals, critical mission goals, communicability goals, and normal operational goals in decreasing priority order. Of course, there is room for variability in such a scheme. There may be some critical activities, for instance, that need to be completed even it ultimately dooms the system by exhausting some resource. What's important architecturally, therefore, is not the particular order, but rather the fact that it is adjustable. In MDS, this control can be exercised freely as a function of mission phase or activity.

Recovery

As a system begins to recover, it is likely to find that conditions are substantially altered from those in place just prior to the fault. In conventional designs this is made tolerable only in the most critical cases, where massive effort is expended on specialized sequences and fault responses unique the situation. A typical system would handle this by establishing a handful of "marked" sequence states to which the system can return and from which resumption of the timed command sequence is possible. This may repeat previous commands so actions not rolled back must be made tolerant of re-commanding. In addition, because the sequence is tardy from the fault interruption, it must be designed to allow compression — basically by running as fast as possible until it catches up. However, because this is not guaranteed to honor all timing constraints, conditional delays must be sprinkled (with conservative parsimony) through the sequence. And, no matter what the actual fault had been, there would never be anything but the original sequence of commands to resume, compelling the system always to return to the same narrow path.

Plotting such a universal path to success is difficult. Yet despite all the "predictability" of this cherished approach, fault protection is *still* saddled with recovery from almost arbitrary starting conditions. Moreover, it must arrive, not at the safest state nor at the best state to proceed necessarily, but rather at one of the marked sequence states. And all this must be accomplished with virtually no architectural support: no resource management, no conflict avoidance … just lots and lots of code. Details vary across systems, obviously, but the general character of the problem remains.

Thus, in the attempt to reconcile these competing operational models, we end up with the worst of both. Getting this right is so complicated and so fragile that it is reserved only for dread cases. It is so bad, in fact, that the temptation (and often the resulting design) is simply to turn fault protection off and go through the episode on blind faith in good luck. How ironic that fault protection should be the enemy!

In MDS surviving goals after fault resolution are still in force. Through elaboration they can rebuild all of the activities required to re-establish normal operation, just as they did in the first place. Depending on how widely the effects of the fault spread, this may result in a sweeping set of activities involving the whole spacecraft. Since most of this activity is driven by normal elaboration, which is designed to achieve goals from any reasonable initial condition, fault responses, as such, tend to be fairly simple. They deal with immediate local consequences of the fault, while normal architectural mechanisms of elaboration, coordination, resource management, conflict resolution, and so on, orchestrate the recovery process and make it straightforward to specify.

The result won't be the sequence that would have occurred originally, but rather will be a different one adapted to the new situation — without backtracking or repetition, and with all the state and timing constraints preserved. Since this occurs seamlessly within the normal workings of the system used in day-to-day operation, it naturally supports resumption of activities under *any* circumstances, not just those few special cases that would ordinarily have merited all the extra work. Even the most mundane activities can be made recoverable.

Really, the only consideration one need make in MDS for critical situations is to assure that preemptive actions are

taken to enable contingencies should the need arise. If there are backup items that need time to warm up or prepare for operation, for example, then making sure they're ready ahead of time allows the system to switch, if necessary, and still meet deadlines. One might leave more margin in consumable resources, as well, just in case fault responses use a little extra. Though these are simple things to arrange, handling such contingency planning is not yet a totally automated part of the MDS architecture. Some day it will be. For now, it is a small burden compared to any conventional alternative.

Reporting

Another aspect of recovery is notification of the ground that something has happened that requires attention. It may first be necessary just to gain the ground's attention, if for example the spacecraft is being operated using beacon tones[5] for this purpose. Such actions would normally be driven by goals that raise the proper semaphore whenever there is something to report and downlink is not otherwise scheduled. This is expressible as a state constraint, because data management and transport states are included as part of the system representation in state variables.

When ground intervention becomes possible, one of the first objectives will be to gain insight into the preceding events and their residual effects. Most of the interesting data will have been captured in state variable timelines, showing the explicitly stated beliefs to which the system reacted, and the steps (as goals) it took in responding to the situation. Other data, such as measurements, may also be available, if policies to preserve such data are imposed by fault responses. In addition, message logs will record low level events, and special reports can be generated (as dictated by adapters) to augment normal reporting.

Clean Up

If further action from the ground is required, it may be to perform additional actions to clarify what happened, to refine the assessment of fault states made by the system or the models used in making these assessments, to make adjustments in existing safety and hazard constraints, to impose additional constraints on the use of faulty equipment, to adjust margins in resource usage, to identify new alternate ways for the system to accomplish its goals, and so on. Only a few of these operations require corrections to code. The remainder can be carried out via goals on the system, updates to state knowledge, or other standard actions within the MDS architecture.

[5] Beacon tones are simple modulations of the downlink signal amounting to the steady transmission of a single symbol. This is quickly and easily detectable by small ground stations, which would then notify a larger station of the need for attention.

Other Facets of Fault Protection

The discussion so far has dealt mainly with the functional aspects of fault protection. Not discussed here are low level issues such as recovery from resets, reliable startup processes, management of computing resources like buses and mass storage, arbitration among redundant computers for system control, computer swaps, computer failure detection and fail safe mechanisms, software assertion mechanisms, software fault handling, software update mechanisms, redundant data storage, soft error scrubbing, the preservation and reliable distribution of clock time, and many other lower level topics. These have not been neglected by MDS. They are merely outside the scope of this paper, especially since some of them depend significantly on the underlying computing system architecture. MDS is doing an adaptation for all these functions on a candidate platform with an eye for standardizing upward looking interfaces in order to facilitate ports to other platforms.

5. Systems Engineering Better Fault Tolerance

As a final note, it is fitting to return to the role of systems engineering in MDS that was advanced at the beginning of this paper. Clearly, systems like MDS are dependent on sound representations of state and their behavior, and every effort has been made to ease the translation from common systems engineering terminology into software expressions. The adoption of a state- and model-based approach to systems engineering, consequently, would be a boon to any system using MDS. This is hardly the only reason to adopt these methods, though. The rigor and formality of modeling reaps benefits from the moment a project is conceived until the last bit of data arrives home.

One sees this as soon as one attempts to model something about the system. If this modeling is hard to do because of complex coupling, or if state boundaries aren't easily delineated, or if models seem to be full of exceptional cases, or if abstracting high level behavior is problematic, or if it is difficult to express intent against the mechanisms available, then there is ample reason to believe that the underlying system design is flawed. The same principles that facilitate modeling are those that guide good systems engineering.

Moreover, models turn out to be an excellent way to specify a system design in the first place. Not only do they encourage clean, integrated design throughout the process by highlighting issues as they appear, but they are directly applicable to the verification of the ultimate design, serving both as a proof against which the system is judged and as a simulated environment within which tests can be performed. The value continues into operations, with models providing a basis for planning and analysis. The fact, then, that both flight and ground software happens to find these same models handy in order to reason about the system in a variety of ways is just frosting on the cake!

The relevance of all this to fault tolerance is simply that a clean, well-integrated system is going to be more reliable, and if it does fail, the system's ability to deal with it is going to be greatly enhanced. The concern, often raised, that a model-based system like MDS is only as good as its models, misses the point. All systems are dependent at the very least on the models in the designers' heads. Models are *always* a factor in *any* design. What MDS provides is the discipline to write these models down, get everyone to agree on a single consistent set, and then apply this knowledge directly, rather than encrypting it implicitly into code from which the original model has long since been scrubbed away, as in most conventional designs. Besides, in MDS, if a model turns out to be wrong, the consequences will likely be less severe, because reactions across the system will at least be consistent. Moreover, it will be easy to see what to fix.

Another concern one hears is that all this modeling will be expensive. But compared to what? Given the high cost of conventional fault tolerance (not even counting critical sequences), reduced operability from inflexible and complicated sequencing, lower reliability from lack of full conflict and resource management, lost data from canceled sequences, engineering crises from poorly understood designs, and so on, models evidently have a lot to offer.

6. CONCLUSIONS

The Mission Data System offers a 21st century approach to fault tolerance in space systems. By adopting states and models as its core concepts, incorporating these directly into key architectural frameworks, applying this architecture uniformly across flight and ground systems, and wrapping the result in closed-loop, goal-based control, MDS hopes to achieve unprecedented reliability and ease of use.

The system engineering methods that support this architecture also promise a new era for the design of space systems. If this proves to be the case, there will be no turning back.

7. ACKNOWLEDGEMENTS

The research and design described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

Many of the ideas described in this paper first began to take shape during the development of fault protection for the Cassini spacecraft [2]. The fault protection ideas in MDS owe much to the hard work of this fine team.

A major refinement of these concepts was tested as part of the New Millennium Autonomy Architecture Prototype (NewMAAP) and its follow-on flight demonstration, which flew successfully on NASA's Deep Space 1 mission as the Remote Agent eXperiment (RAX) [3].

Finally, the work reported here would not be possible without be the collective effort of the entire MDS team to whom the author is intensely grateful.

REFERENCES

[1] D. Dvorak, R. Rasmussen, G. Reeves, A. Sacks, "Software Architecture Themes in JPL's Mission Data System," *Proceedings of 2000 IEEE Aerospace Conference*, March 2000.

[2] G. M. Brown, D. Bernard, R. Rasmussen, "Attitude and Articulation Control for the Cassini Spacecraft: A Fault Tolerant Overview, " *Proceedings of the 14th AIAA/IEEE Digital Avionics System Conference*, November 1995

[3] B. Pell, D. Bernard, S. Chien, E. Gat, N Muscettola, P. Nayak, M. Wagner, B. Williams, "An Autonomous Spacecraft Agent Prototype," *Proceedings of the First Annual Workshop on Intelligent Agents*, Marina Del Rey, CA, 1997.

Robert Rasmussen *is a principal engineer in the Information Technologies and Software Systems division of the Jet Propulsion Laboratory, California Institute of Technology, where he is the Division Technologist and the Mission Data System architect. He holds a BS, MS, and Ph.D. in Electrical Engineering from Iowa State University. He has extensive experience in spacecraft attitude control and computer systems, test and flight operations, and automation and autonomy — particularly in the area of spacecraft fault tolerance. Most recently, he was cognizant engineer for the Attitude and Articulation Control Subsystem on the Cassini mission to Saturn.*