PREDICTING SOFTWARE FAULTS

Jay Naphas

Federal Aviation Administration/AST-300, 800 Independence Avenue SW, Washington, DC 20591, USA, Jay.Naphas@faa.gov

ABSTRACT

At first glance, software faults appear to occur randomly. On deeper inspection, faults seem to possess a devious intelligence, hiding in the system to elude software testers and pouncing on unsuspecting users at the worst possible moments. These observations hint at the idea that we need a central organizing theory of software safety that makes testable predictions about the potential software faults in a system from known or knowable information. This paper proposes a new theory to meet this demand, and presents both the derivation of the theory and a set of predicted observations to verify it.

1. DERIVING THE THEORY

All software faults are manifestations of errors in mental models; this is the proposed central organizing theory of software safety. Mental models are developed by communication processes, both among people and between humans and computers. Mental models are stored in the human mind between communication steps and in software as it is written. Errors in software may or may not be severe enough to demand correction, but the errors exist, in every case, as a result of errors in mental models. From the theory, we predict that the most effective prevention and mitigation measures for software faults will be derived from psychology and sociology.

Further, software error mitigations that ensure the construction of accurate mental models and communicate those models with high fidelity will be the most effective, and organizations that implement such measures will experience relatively few software faults. These predictions are testable, and examples from accident history attest to their accuracy.

At this point, it is important to note the role of computer hardware in software safety. The mental model stored as software must include an understanding of the mechanical limits of the hardware on which the software executes. Any defect in that understanding, or any difference between the as-built computer hardware and the programmer's assumptions about that hardware, will introduce potential errors into the software. Technical understanding of computer hardware, and indeed of system hardware generally, is a necessary, but not sufficient, part of the mental modeling processes that produce software.

The central organizing theory of software safety also predicts that lower software error rates correspond to effective communication techniques within and between every stage of the software development process, and indeed at every stage of the system development process. Further, communication effectiveness is necessarily limited by the capabilities of the human mind and the computer hardware; the implications of this constraint are assessed in this paper and applied to complex software development. What emerges from this paper is a theory that organizes our thinking about software safety, and that makes experimentally verifiable predictions about the types of software faults that can emerge from any specific development effort.

To derive the theory, we must first limit the faults under consideration, and software safety must include all those situations in which software has a safety role. The software safety domain is defined here as a continuum from faults with minimal, peripheral software involvement to faults that manifest entirely in the software. At one end of the software safety domain, we find computer hardware faults for which software may be a possible mitigation; single-event upsets in memory are one example. These are hardware problems for which software can provide an effective remedy, but for which software is not intrinsically necessary.

The software-only end of the spectrum is more difficult to bound, and it is this boundary condition that gives us the first insight into the form of the theory. A true "software-only" fault would have to manifest in the software of its own accord. This definition rules out human error, hardware flaws, and design defects; design, after all, is a human action. The fact that all software is brought into existence by human action, whether by directly coding or using other software to deliberately generate more code (as with a compiler or autocode generator), implies that, in the matter of finding a true root cause, no "software-only" end exists. Faults that manifest in software do so, without exception, because a human put the faults there.

It is important, at this point, to note that much software is now written using other software. In modern coding, it is therefore critical to understand the mental model stored in the software that you use to create other software. The theory predicts that there is an opportunity for mental model conflict, and therefore software error, of a size and form dictated by the disparity between the actual mental model stored in the coding support software and the programmer's understanding of that model. In other words, the programmer must understand the software they use to create software, as well as the software they're creating, in order to accurately translate their mental model into code.

The fact that software faults are universally of human origin leads to the theory of software fault causation when combined with one other important fact: software is a set of instructions that a computer executes over time, not a component that exists in space. A fault in software is thus analogous to a missed step in a procedure, a computation performed with invalid numbers, or a mathematical error in a calculation, such as 2+2=5. These are errors that represent flaws in the mental model preserved in the software; they are errors in design. Software faults are therefore the manifestations, during execution, of the flaws in the models from which the software was written. Those models are formed in the human mind, and are thus formed by psychological and sociological processes.

Indeed, when software faults manifest into accidents, it often seems as though "the machine had a mind of its own," and this is not a trivial observation. In such situations, the software truly "believes" something about the world that is patently false, and that belief has caused it to take unsafe actions or inactions. The difference between reality and the software's understanding of the world is the proximate cause of such accidents. The root cause of all software errors, however, and the cause that we can address with the theory of software safety, remains with the humans, for they inescapably give the software all of its beliefs, both true and false.

There are theoretically infinite beliefs that can be used to write software, and an equally infinite variety of false beliefs that serve the same purpose. The problem of false belief has resulted in the development and application of Common Ground Theory [1] to improve the safety of robotic systems; by checking the robot's beliefs against the operator's beliefs directly, the two perform tasks faster and safer [2]. A false belief may be that it doesn't matter which propellant gets to the combustion chamber first, or that 3 meters plus 1 foot equals 4 meters, or that 129 can be stored in an 8-bit signed integer, or that all values between -10 and +10 are permissible denominators, or that the compiler checks for counter synchronization, but the practical implication is that an erroneous belief was used by a programmer to write the software. That is the root cause of every software fault.

This conclusion is not to be used to blame the programmer, because the flaw in the mental model, or the false belief, very frequently enters the system during requirements development. Creating a complex mental model of a system that no single person can fully understand demands communication, for communication is the only means available to increase the mental capacity of a person or group beyond the bounds set by psychophysiology. Communication is therefore critical for the development of good requirements, because each system designer must be able to check their changes to the design against a common model of how the system will operate. Communication errors are frequently the cause of requirement errors, and requirement errors are passed to programmers who dutifully write them into system behaviors that pass tests. At the same time, the system tests are frequently written from the requirements that contain the initial error, which makes the primary mitigation measure for software error (testing) far less effective when this is done.

Communication errors, in this paper, include errors of both commission and omission, both deliberate and inadvertent, and both within groups of humans and between humans and computers. Further, the working definition of "communication" here is: "the process by which a mental model is transferred from one storage medium (biological or electronic) to another." Psychological limitations are the characteristics of the human mind that bound the size and complexity of the concepts it can actively manipulate and store at any one time. This further complicates the communication problem, because there is the potential to lose information both due to the method of communication and the mind's method of storing and recalling it.

This is where the central organizing theory of software safety finds its greatest utility: focusing our mitigation measures on the very complex problems presented by the development of complex mental models. Mitigation measures for software faults must maintain the fidelity of the mental model of a system throughout its development, and provide the means to directly check the mental model of the system for errors. Armed with this test, we can evaluate the effectiveness of changes to the design process in terms of storing and checking mental models prior to making changes, and consciously design system development processes that transfer and store mental models with a minimum of loss or error. That is precisely what will reduce the prevalence of software error.

The theory has one further implication. The design of a system is precisely analogous to the design of software, so we postulate that the safest designs will result from design processes that produce and store accurate mental models. The theory can thus be expanded to read: all errors in software and system design are manifestations of errors in mental models. We can therefore evaluate design processes, as well as programming practices, based on their ability to develop, communicate, and store mental models. The implications of this theory for reducing the initial error prevalence in new systems will be explored more thoroughly in a forthcoming paper.

2. SPECIFIC PREDICTIONS

To be useful, a theory must make specific, testable predictions about the nature of future observations. Several such predictions arise directly from the theory of software safety posited in this paper. To understand these predictions, we must first characterize exactly what is meant by "communication error" or "modeling error," as those are the two sources of software error predicted by the theory.

A communication error is a failure of a pair of communicators to transfer complete and accurate mental model information between the sender and the receiver, which results in an incomplete or inaccurate mental model in the receiver. This could be due to any or several of the following, or others not listed:

- Failure of the communicator to recall and convey all of the relevant details stored in his or her mental model.
- Failure of the receiver to check the new mental model for inconsistencies.
- Failure of the communicator to verify that the receiver has stored and can recall the components of the mental model during communication.
- Failure of the communicator to check his or her mental model against that of the receiver after communicating it.

There are a wide variety of factors that influence the likelihood of communication error. These factors include the following:

- Interpersonal conflicts that overtly hinder communication, such as open hostility, evident or perceived bias, or other breakdowns in communication between sets of individuals.
- Low-fidelity communication methods, such as the use of long text descriptions in place of easily comprehensible diagrams, email in place of meetings, or verbal exchanges without adequate preservation of their content.
- Structural impediments to communication, such as geographic separation of development team members, when not mitigated by communication technologies.
- Assumptions regarding the knowledge bases of others, such as educational background or prior experience, that if inaccurate may lead a communicator to believe that the receiver either already knows some key piece of mental model information (such as the units in which a program is coded) or that the receiver knows too little (leading to a bored or dismissive receiver attitude). Such assumptions may lead to interpersonal conflicts.
- Social pressures, both to appear knowledgeable and subordinate, may influence the information given by a communicator and the thoroughness of the receiver's checking of that information.

A modeling error, by contrast, is an error that occurs during a person's internal mental manipulation of the model. The preconditions for modeling errors exist in the human mind continuously, and change with experience, making modeling errors difficult to prevent. Human working memory is limited biologically, and this limitation leads to many modeling errors of the "oops, I forgot something" variety. Other sources of modeling error include, but aren't limited to:

- Attention errors, where an aspect of a mental model is thought to be less important than others, and under-analyzed as a result.
- Interruptions. Mental model manipulation is a demanding task, and requires continuous focus to prevent components of the model from being displaced from working memory by a distraction, whether momentary or persistent.
- Extraneous information, brought in from the manipulator's prior knowledge or experience, can influence the importance given to components of a mental model, or alter them entirely.
- Preconceived notions about the system may lead to inaccurate perception of the system or its anticipated operating environment, which in turn distorts a person's mental model of the system, both when they first learn about the system and during their manipulation of the mental model of the system.

The two sources of software error, characterized above, lead directly to testable predictions about the form and prevalence of software errors. In the following discussion of communication problems, it is important to acknowledge the challenges associated with diagnosing communication problems in an organization in real time. To overcome this limitation, we can focus our efforts on setting up an organization for good communication from the start, rather than relying on finding and fixing problems along the way. Good communication is predicated on trust, familiarity, and skill in conveying information.

The most evident prediction derived from the theory of software safety concerns software testing. Development efforts where the testers report feeling "disconnected" from the process, or where tester feedback is minimal, or where testing is limited to tests designed by system or subsystem designers, will experience an increased proportion of errors in general. It has long been known, colloquially, that software systems tend to be safer when the testers are highly engaged and given plentiful resources, and the theory explains why this has been observed. Constructing and checking mental models takes time, and testers with adequate time and resources to perform those functions are more likely to find errors. In particular, testers given the time and latitude to devise and run innovative tests of a system tend to find errors that would pass through the scripted test process undetected. Defining "adequate time and resources" remains a challenge, and the definitions will necessarily vary with other traits of the development effort, but the essential point is that some combination of time and resources in the testing phase will produce a system that has been verified across the predictable limits of the operating environment, and that the mental model stored as the software must be verified across the environment in which it will be used.

Development efforts that feature communication problems within the system designer group will tend to have more errors generally. In particular, two specific types of errors tend to result from such a development effort:

- Resource allocation errors (memory leaks, throughput deficiencies, etc) due to changes in requirements as the design matures and prior mistakes are corrected.
- Requirement errors (wrong units, mode confusion, etc) due to conflicting requirements.

Development efforts that feature communication problems between system designers and programmers will also tend to have more errors in general, and similar specific errors. There are, however, some differences worth noting:

- Resource allocation errors (memory leaks, throughput deficiencies, etc) will be concentrated in specific software modules, instead of distributed across the system.
- Requirement errors (wrong units, mode confusion, etc) will also be concentrated in specific modules, and these concentrations will correlate with specific programmers.

Notably, in both of the previous two predictions, the length of the requirements development and design phase is not the key factor; the characteristics of communication drive the form and prevalence of the errors that result from the development effort. In fact, short requirements development and design stages tend to result in more errors discovered during coding and testing. This is due to incomplete communication in and between the design and requirements stages; complete communication takes time, and that time is well spent up front to prevent problems during testing.

It is also noteworthy that diagnosing communication problems is a non-trivial challenge. Almost by definition, communication problems are difficult to detect because communication problems cause the loss of information; without information, it is difficult to detect a problem in the information. This is a challenge worth meeting by training people in communication skills.

Development efforts that feature communication problems between programmers and their computers will tend to encounter problems in the debugging and testing phases. Such efforts are fortunately rare, as this has been the focus of much of the discipline of computer science since its inception. The evidence that an effort is experiencing this problem would be requirements that pass every test for consistency and completeness, including critiques by programmers. Should such a problem be discovered, improved coding interfaces may be needed, or programmer qualifications may need to be revisited.

The theory of software safety also makes the testable prediction that the development effort that produces the most complete mental model of the system, and transfers it accurately throughout development, will result in the safest system. On a decision-by-decision basis, this implies that aspects of the development method can be evaluated for the degree to which they produce and communicate complete and accurate mental models. Safe software development is the result of evaluating the development processes on the specific basis of ensuring mental model fidelity.

3. PRECONDITIONS FOR SAFE SOFTWARE

The theory that all software faults are defects in mental models produces useful information about the structure of a safe software development process. Specifically, there is a set of readily accessible preconditions on which safe software necessarily depends. These preconditions include:

- Testers must have the time, resources, and authority to test their full mental model of the system against the as-built system.
- Programmers must have open lines of communication with system designers, architects, and end users, whenever possible.
- System designers must be able to consult programmers during initial system design. This will allow them to intelligently allocate tasks between hardware and software, as well as verify feasibility of requirements.
- System designers must be trained to discern and communicate the assumptions they make about the way a system will be used.
- Testers must be encouraged to find innovative tests of a system that challenge it to respond to novel situations.
- Predicted or representative end users must be consulted during system design to verify that the requirements implemented by the system designers meet the requirements of the end users.

These preconditions are offered as examples of imperatives derived from the theory. Software development efforts share these needs regardless of the purpose of the software, and the assurance of such preconditions will drive errors out of any software development effort.

4. CONCLUSION

The central organizing theory of software safety proposed in this paper is that software faults are, without exception, manifestations of errors in mental models that result from psychological and sociological processes. This theory gives the space safety community a powerful new insight into the prediction and mitigation of software faults, a predictive power that is absolutely vital in an arena where “fly-fix-fly” is prohibitively expensive and ethically untenable. The theory, or at least hints of it, has been long suspected by many in the software community, but formal recognition is required to drive beneficial changes in the way software is developed.

With a formal theory, we can now derive specific methods for software safety that directly address the root causes of software errors. Further, we can now know, in advance, which safety method will produce better results by comparing the fidelities of their mental modeling processes. It is this predictive power, the ability to know software has been developed safely before it flies, that will be of great benefit to the space safety community in the decades to come.

Finally, the theory is broad enough to cover all design processes, not only software development. The critical fact is that, in both software development and system design, the errors in the mental model transfer into the operation of the resulting system. The mental model becomes the physical system and the software that runs it, and we can and must get that mental model right. Speaking freely, communicating clearly, and questioning extensively will prevent errors in our designs and in our software.

This paper has named the “enemy” in software safety: mental model errors are the cause of all software and design errors. We can now select the design processes and programming practices that produce accurate mental models, and know that our systems will be safer as a result.

5. REFERENCES

1. Clark, H. H. (1996). *Using Language*. Cambridge University Press.

2. Stubbs, K. (2008). *Robot Proxy Grounding*. Doctoral Thesis, Carnegie Mellon University, Pittsburgh, PA.