Software Fault Injection: Growing 'Safer' Systems

Jeffrey Voas
Reliable Software Technologies
Suite 250, 21515 Ridgetop Circle
Sterling, VA 20166
(W) (703) 404-9293 (F) (703) 404-9295
jmvoas@RSTcorp.com

Abstract—This paper describes software fault injection and what types of anomalies fault injection should simulate. We highlight the benefits that software fault injection could have provided to several infamous software error-related incidents. And, we explain how fault injection can be used to "grow" safer systems.

Table of Contents

1. Introduction

If we knew what the future held, then we could know precisely how good today's software is. More precisely, we would know when in the future it will fail, how often it will fail, and under what conditions it will fail. Armed with that information, it would be trivial to confidently assess the "goodness" of the code.

To us, the future is a *black box*. We do not know what faults will be triggered, and we do not know how severe the failures will be. All that can be done from a development standpoint today is to fix *known*, existing faults; but since we cannot get out all of the faults, this situation is unsatisfying.

Fortunately, we are not limited to a development standpoint, and can estimate the quality of the code under predicted anomalous circumstances that could arise tomorrow. What this provides is a means for predicting the quality of the code under simulations of undesirable circumstances. The inability to predict future behavior has long been a problem for software quality metrics. However, fault injection is a completely different breed of metric, which, because of its more complex nature, can partially address this problem.

Software fault injection includes a family of techniques that instrument the original code with mechanisms (more code) that either modify the existing syntax or force the state of the code to be modified when the software

executes. Either way, the code or its behavior is somehow changed. It is this process of modifying events that makes it ideal for predicting what might happen if future events get disturbed.

2. Fault Injection Basics: What to Inject

We now describe the basic issues related to using software fault injection to grow "safer" systems.

Development processes, even formal, are not solutions for *measuring* software quality; they only seek to build software with quality behavior. We champion fault injection, a process that doesn't tell you how good the code is *per se*, but instead provides a worst case prediction for how badly the code might *behave* in the future (as opposed to how bad it might *be*). Recall that correct code can even have "bad days" and not work as desired due to external influences on it. For example, your workstation might be in good working order, but if the network server is not functioning, you may find that the workstation is so unusable it might as well be broken too. This worst case assessment is not a *direct* prediction of the future, but instead an *indirect* foreshadowing of how bad the future could get.[1]

If software does not experience any problems during execution, then it cannot behave badly; only when it encounters problems that corrupt its program state can things go awry. An *anomalous operational scenario* (or anomaly) is an event that creates a data state during software execution that alters the output of the software such that the output is undesirable. The number of different anomalies that software can experience during its lifetime is almost always infinite and unknown. The anomalies that we are interested in here are those that can arise from code defects (caused by programming errors or design defects), human factor errors, or other external failures (from hardware upon which the software depends for inputs or from other software). The combinations of anomalies from instances of these different problem sources is intractable.

It is becoming common for software engineering standards to enumerate certain classes of problems that must not occur. The argument *for* doing so is that you can't protect against problems until you know what the problems might be. Here, problems are usually defined in one of two ways: (1) a class of failure that must not occur, or (2) a fault class that can occur but the fault class must be shown to only cause acceptable outputs.

IEEE standards have been criticized as "ad hoc and unintegrated" because they have been developed over time by many different people in a piece-wise manner [4]. In fact, with the exception of the ISO-9000s and the SEI's CMM, unless regulatory standards are applicable for licensing, most organizations are ignorant to what standards are current. If it were not bad enough that software standards do not ensure that good code will be developed, it is also true that you cannot count on using reliability models or direct mean-time-to-failure (MTTF) measurement for risk assessment. Correct software does not guarantee risk-free operation. Standards are at best vague goals from which to start, but even their satisfaction does not prove an absence of risk. Demonstrating that software outputs are what we want (or at least can tolerate) requires showing that even if bad events happen during computation, un-

[1] Software testing might be able to give a direct foreshadowing of how bad the future could get, if operational profiles and anomaly frequencies are available, but this is debatable.

desirable outputs will not, and standards, correct software, and reliability models do not provide this demonstration.

Fortunately, this can be partially accomplished via fault injection methods, since we can observe how *well-behaved* software is in even the most disheartening of circumstances and then look for ways to improve the software's behavior. But even software fault injection methods suffer from an intractability problem (in terms of how many different anomalous scenarios can be instrumented) which is similar to the exhaustive testing problem:[2]

> Since the space of potential faults is infinite, the space of anomalies to inject is also infinite. What do we choose to inject rather than ignore?

To get a better feeling for what anomalies must be considered, we will first introduce notation. For now, we will assume that no modifications to program P will occur; when we relax this assumption, the spaces that we are about to define must also change, which suggests that the fault injection methods may need to be continually reperformed.

Let Γ represent the space of *all* anomalous events that *could* affect P's output during P's life-time; it is not necessarily the case that all members of Γ *will*. Let A denote the portion of Γ representing those anomalous events caused by the program logic errors in P, and let B denote potential anomalous events caused by human factor errors or external failures. A represents a finite and unknown space of events, and B represents an "effectively" infinite and unknown space of events.

Since A represents those defects already in P, to determine their impact on P's behavior, you simply need to execute P; i.e., there is no need to use fault injection methods to simulate them.[3] However for B, fault injection methods will be attractive for studying how those types of events might affect P.

We can further divide B into two spaces: b' and b''. b' represents those problems that *will* occur in P's future, and b'' represents those problems that could occur in the future but will not. Both b' and b'' are unknown before (and after) the fault injection analysis, hence for any member of B, we cannot know if it is in b' or b''. Note that b' is finite and b'' is probably infinite.

It would be ideal for the fault injection methods to simulate b', but to do this requires either incredible luck or omnipotence. Even if you were lucky enough, since b' is unknown, you would never know that you had succeeded.

The impact of the anomalies in A will be statistically determinable by testing P. For a member of A to have a negative impact on the quality of the software's outputs, all three conditions of the *fault/failure* model [8] (*propagation*, *infection*, and *execution*) must occur. If they do not occur, or they occur with little frequency, then the negative im-

[2] Dijkstra showed that testing could not reveal the absence of errors unless it was exhaustive, and fault injection suffers a similar fate: it can show output behaviors that are undesirable for the anomaly injections tried, but for all of those classes of anomalies not tried, we cannot know if the resulting output behaviors would be undesirable or not. Hence the results of fault injection mean a lot if undesirable outputs are observed during the analysis and little if no undesirable outputs are observed.

[3] This is not completely true, because faults of small size are likely to have an impact on P during testing, and hence simply running P is not likely to make their impacts observable. And so it is often interesting to inject faults into such systems (even though we know they are incorrect) to ascertain how likely it is that faults could be hiding (from testing). This is the basis for Voas's testability model [5].

pact to quality will be greatly reduced. Testing will not, however, be capable of predicting the impact of those members outside of A (including members of B).

For B, traditional testing will be of no help. But what can be done is to simulate as many members as are known (without ever knowing whether we are sampling from b' or b''), and then sample as many other members from Γ as resources allow for. The reason that some of B's members may be known is that we can identify certain failure modes for the humans, external software, and external hardware systems that feed information into P. And if you restrict B to only anomalies that can be passed by component interfaces, fault injection can scale to large Commercial-Off-The-Shelf software packages [6].

So in summary, to "grow" safer software, we need to convince ourselves that A does not contribute to hazardous output, and inject members of B to convince ourselves of the same. For brevity, we will not describe the actual instrumentation processes that we have employed for fault injection. An explanation of these can be found in [5, 7], which is publically available as a technical report (`ftp://rstcorp.com/pub/papers/ieee-gem.ps`). In that report, we also provide several real-world case studies that underwent fault injection analysis, one for a medical device, and the other for a train control system.

3. Infamous Disasters

Section 3 is devoted to two infamous mechanical disasters, the Ariane 5 and Therac-25, and what software fault injection could have done to thwart the results. Our analysis is based on the best information available from the groups that diagnosed and publicized what the software problems were.

Ariane 5

On June 4, 1996, the maiden flight (Flight 501) of the Ariane 5 launcher ended in disaster. Only about 40 seconds after initiation of the flight sequence, at an altitude of approximately 3700m, the launcher veered off of the expected flight path, at which point it exploded. A team of investigators from the Ariane 5 project teams was assembled to investigate the failure. The Director General of the European Space Agency and the Chairman of the French Centre National d'Etudes Spatiales (CNES) set up an Inquiry Board to investigate the problem further; we have used the published results from this board in our analysis [2]. We will now talk about the accident and the role that fault injection could have played in preventing this disaster.

The Flight Control System of the Ariane 5 is of a fairly standard design. The positioning of the launcher and its movements in space are monitored and measured by an Inertial Reference System (SRI). The SRI has its own computer, in which angles and velocities are computed based on incoming data from sensors. Data calculated by the SRI is then sent through a databus to the On-Board Computer (OBC), which executes the flight program.

The Ariane 5's design contains redundancy. It had two SRIs operating in parallel, with identical hardware and software. One SRI was active, and the other was in standby "hot swap" mode. Also, the Ariane 5 had two OBC's, and certain parts of the Flight Control System were duplicated. Interestingly, the software design of the SRI for the Ariane 5 was very similar to that of the Ar-

iane 4's design.

We will now take the reader through the sequence of events that led to the disaster. But before we do that, the reader needs to know that with respect to the SRIs, there was a specification requirement that stated that in the event of any kind of software exception, "the failure should be indicated on the databus, the failure context should be stored in an EEPROM memory, and finally the SRI processor should be shut down." Hence any software exception was expected to shut down the SRI computer upon which it occurred. This might seem like a strange requirement, but since there were two SRIs, it was allowed.

In the software on each SRI, there was an internal alignment function called Horizontal Bias (BH) that returned a 64-bit float. For Ariane 5, the value returned by BH was much higher than had been expected, probably because the early part of the flight of Ariane 5 differs from that of Ariane 4 and much of the SRI design for Ariane 5 was taken from Ariane 4, and thus this resulted in much higher horizontal velocity values. After liftoff, SRI 1's software reported an Operand Error while trying to convert the BH value from a 64-bit float to a 16-bit signed integer. Because the value was too large, the operation failed, thus sending off an exception. This led to the immediate shutdown of SRI 1.

Almost immediately thereafter (72 milliseconds to be exact), SRI 2 tried to perform the same conversion, and it too failed. When SRI 2 failed, however, it sent incorrect attitude data to the OBC, which was controlling the flight. Part of the data that the OBC received from SRI 2 appears to have contained diagnostic bit patterns, as opposed to proper flight data. This forced the OBC to change the angle of attack such that the high aerodynamic loads caused separation of the boosters from the main stage, in turn triggering the self-destruct system of the Ariane 5. So the immediate cause of the disaster was a software exception, but that is not the full story.

The question then arises as to why a type conversion was allowed to occur when an exception was specified to shut down the computer. Part of the reason for this was that the computers on Ariane 5 were not to exceed a maximum workload of more than 80% of capacity, and hence protecting all conversion calculations might have contributed to exceeding this threshold. During design, the engineers had identified seven variables that were at risk for leading to an Operand Error, one of which was the BH. Four of those seven had been protected, but BH and two others were decided not to need protection.

The reasoning was as follows: for the variables that were left unprotected, it had been decided that either it was physically impossible or there was a large enough margin of safety built into the system that would accommodate them resulting in an Operand Error. Hence, BH and two other variables were left unprotected. And it appears that from what was known from Ariane 4, this thinking was plausible, and hence we speculate that the engineers probably thought that for BH in Ariane 5, an Operand Error was physically impossible. But as we know now, this rationale was incorrect.

Where fault injection could have played a major role in avoiding this disaster is obvious. First, if the reasoning used for not protecting BH was that there was an adequate margin of safety already built into the system, then fault injection could have quickly shown otherwise. All that would have been required was a perturbation function that would have thrown the 64-bit float

to a high level, and the exception would have been triggered. Given that both SRIs run the same code, that would have meant that both SRIs would have failed, and this information would have been immediately available for the designers.

Secondly, recall that the OBC received corrupted data via the databus; that data appeared to have diagnostic bit patterns in it as opposed to correct flight data. Fault injection can easily simulate corrupt data going into the OBC's software, and we have no way of knowing whether this was done during testing of the OBC. Whether or not simulated corruptions employed by fault injection would have simulated diagnostic bit patterns would have depended on the classes of corruptions tried when the fault injection was employed. We assume that it would have been known to the designers that when SRI2 failed, this was the type of information that would have been on the databus, particularly because of the requirement: "the failure should be indicated on the databus, the failure context should be stored in an EEPROM memory."

But it appears that the Ariane 5 design team were more convinced that the BH variable could not have been as large as it was. In the findings of the Inquiry Board, there was no evidence that actual expected trajectory data was used to analyze the behavior of unprotected variables, and even more alarming is the fact that it was agreed early on to not include Ariane 5 trajectory data in the SRI requirements and specification. What this means then is that the domain of trajectory data for the expected environment of the SRI software was not included as a part of the SRI specification, even though this specification appears to have been partially reused from Ariane 4. Whether or not that information did make it into the test plans for the software is not known to us, but it appears that even if it did, the test cases that would have demonstrated this problem were not used.

The lessons here are many fold, since there was a series of mistakes that contributed to the software defect. The key lessons will now be summarized. First, when code is reused in a new environment, you must consider the environment thoroughly before deciding what can or cannot happen in the new environment. In the case of Ariane 5, it appears that because the value of BH was not so large in Ariane 4 to cause an exception, it was assumed that this would not be a problem for Ariane 5. Since trajectory data was not a part of the specification nor requirements, it would have been very hard to argue that BH could indeed have been so great as to cause the Operand Error.

Secondly, fault injection determines levels of safety. In the report from the Inquiry Board, it was stated that members of the project team had simply decided that safety was great enough to leave three variables unprotected. Since the engineers were not considering Ariane 5 trajectory data as part of the specification or requirements, they had little evidence upon which to base that claim. Fault injection could have easily mitigated that belief as either true or false, and with only a few perturbation trials.

Therac-25

Our next case study is for the Therac-25 accidents.

Therac 25 was a medical linear accelerator that sent electrons at high speeds into tumors with the goal of destroying tumors with minimal impact to the surrounding tissue. The Therac-25 was first released for commercial sale in 1982. Eleven Therac-25s were installed. Five were installed in the US, and

six were installed in Canada. Six massive overdoses to patients occurred between 1985 and 1987, three of which resulted in death.

Like the Ariane 5, the Therac-25 was based on a predecessor, the Therac-20. But unlike the Ariane 5, which was heavily based on Ariane 4's software design, the Therac-20 had little or no software; instead, the Therac-25's software was intended to replace some of the hardware functionality of the Therac-20. For example, the Therac-20 had hardware interlocks that checked to ensure that illegal overdoses of radiation could not be administered. The Therac-25 did not have hardware interlocks, and did not even have equivalent software interlocks to perform this functionality.

Leveson and Turner have provided an in-depth diagnosis of the history of problems that plagued the Therac-25, and thus we will only reprint a tiny portion of that information here [1]. Instead, we will focus only on the main software problem, and see what, if anything, fault injection analysis would have done in predicting this problem before the Therac-25 was released in 1982.

To make our reasoning understandable to the reader, we need to at least show the pseudo-code that Leveson and Turner published in their paper in IEEE Computer [1]. This is not the Therac-25's code, but instead is a pseudo-code representation of the main control-flow features of the software. To make the ensuing discussion easier to follow, we have added line numbers to the code.

```
1. Datent:
2. if mode/energy specified then
3.   begin
4.    calculate table index
5.    repeat
6.       fetch parameter
7.       output parameter
8.       point to next parameter
9.    until all parameters set
10.   call Magnet
11.   if data entry is complete then
                set Tphase to 3
12.   if data entry is not complete
                then
13.      if reset command entered
                then set Tphase to 0
14.  return

15. Magnet:
16.     Set bending magnet flag
17.     repeat
18.       Set next magnet
19.       Call Ptime
20.       if mode/energy has changed,
                then exit
21.    until all magnets are set
22.   return

23. Ptime:
24.      repeat
25.        if bending magnet flag
                     is set then
26.           if editing taking
                     place then
27.              if mode/energy has
                   changed then exit
28.    until hysteresis
            delay has expired
29.    Clear bending magnet flag
30.    return
```

The above software controls the hardware and configures it to administer the correct energy levels based on operator inputs into Therac-25. After an operator has input the dosage levels in **Datent** (data entry), the function **Magnet** is responsible for physically getting the bending magnets into place for administering the dosage. This process takes approximately 8 seconds to set each magnet, and so while a magnet is physically

being set, the software calls another function **Ptime**, that sits and waits in a repeat loop for a time interval of hysteresis delay (8 seconds) to see if the operator modifies the prescription that was entered. If the operator does so, and **Ptime** detects that a different energy level and mode was entered, then the software's logic forces control flow to immediately exit **Ptime**, then exit **Magnet**, and then exit **Datent**. This basically amounts to restarting the system from scratch.

The key points to note in the code are as follows. Notice that in the **Ptime** function, the software will only look to see if editing has taken place (line 26) if the bending magnet flag is set. Notice also that at the end of the **Ptime** function, this flag is cleared (line 29). And the only place that the bending magnet is set is on line 16 at the beginning of the **Magnet** function.

The problem then is as follows: at the end of the first iteration of **Ptime**, which is executing when the first magnet is physically being set by function **Magnet**, the magnet bending flag gets cleared, and that flag cannot ever be reset, because the only place that it gets set just happens to be outside of the repeat loop in **Magnet**. The inner condition in **Ptime** is the only place where modified operator input is checked for. Hence since the **Ptime** repeat loop executes for about 8 seconds for each magnet, the only time that the Therac-25 will consider modified input data from the operator is during those moments when the first magnet is being set. After that time, when succeeding magnets are being set, Therac-25 will ignore any modifications to the energy and mode parameters from the operator console.

At this point, you might be wondering why this problem was not "tested out" of the code. After all, there are several different classes of test cases that should be able to "shake out" this bug. Recognize that if test cases are used that never test the edit prescription condition on line 26 (meaning make it evaluate to TRUE), then this bug cannot be detected via testing. And if test cases are not used that exercise this condition on a second or later calling of **Ptime**, the bug cannot be detected. Hence to detect this bug during testing will require time-dependent input cases, and possibly many of them, because it may not be the case that all incorrect commands to the Therac-25 resulted in a miscalculation of the dosage to the level that it would be observable in the output dosage commands from the system.

This brings up an interesting point: time-dependent inputs. Time-dependent inputs are data events that are entered at very specific points in time during execution of the software. For example, stress testing is a form of testing where many work requests attack a system simultaneously to see if the system can handle greater workloads. Recognize that trying to do 100 jobs in one hour is very different than trying to do 1 job per hour over a 100 hour period of time. Note that the amount of work being requested may be the same in these two scenarios, but, the timing of when the requests come will likely impact the ability of the system to not fail.

To observe the effect of the Therac-25 problem via fault injection, i.e., to be able to know a priori that this software could result in an unacceptable output dosage level, we will need the following: (1) test cases that include the scenario where an operator is editing a prescription while the bending magnets are being set (but we will not require that these test cases need the editing to occur at any specific time), (2) a hazard assertion that acts as the hardware interlock did on the Therac-20, and (3) fault injection applied to the result of line 16, i.e.,

clear the flag setting. Note that these three entities are nothing unusual; i.e., these are the normal parameters that we would employ for any fault injection-based analysis for this software. The test cases that we need must exist from coverage and feature testing, otherwise the inner code in **Ptime** could not have been exercised. The fault injection approach that we use always perturbs flags at the places where they are set or unset. And for this particular system, the hazardous event that we would not want the software to be capable of enabling would naturally be a radiation overdose.

When you do this, the hazard assertion will at some point be triggered. How many test cases (trials) it will take before the dosage calculated by the software is out of the acceptable range is unknown to us, but as demonstrated tragically in the field, this event will, at some point, occur. Had this been applied before the Therac-25 went into service, this anomaly should have been detected. Of course these tragedies could have been avoided also by simply using a hardware interlock similar to the one on the Therac-20. The assumption that software added so much more accuracy than the Therac-20 had (making the need for an interlock minimal) obviously proved to be a incorrect.

4. Growing Safer Systems

Many people do not realize how the results from fault injection can be rolled back into the code development process to "grow safer software." Over time, the goal for software is for it to get more robust, and fault injection provides a means for determining if this is happening. Without product measurement, development processes cannot be adequately judged for their value-added. Recognize that the results from fault injection are sometimes estimated frequencies. These frequencies are for some distinct part of the code, and they tell how often an undesirable output event occurred when that part of code was instrumented with fault injection software. If a large estimated frequency is observed, then you know what part of the code the overall behavior of the code is most *sensitive* to.

Fault injection provides an analysis of how tolerant the physical system is with respect to the software and vice-versa. If the hardware cannot be made more failure-tolerant, then the software must be. If the software cannot be made more failure-tolerant, then the hardware must be. It is information about the "give and take" between the different components that allow us to augment our software with the right level of robustness for the physical environment that it will reside in and control.

Many people believe that software analysis techniques, because they are applied at the tail-end of the life-cycle, are too late to be cost-effective. I do not recall how many times I have had people "brush me off" because the methods I advocate are code-based. Typical responses go something like this: "once you have the code, if there are problems, then it is too late." Nothing could be further from the truth! It is never too late, just costlier.

As an example, reconsider the Therac-25 bug. Suppose that the problem was caused by a design error, meaning that the code is correct with respect to the design. If a formal analysis of the code is made with respect to the design, the defect would not be caught, since the code would have mimiced the design. In comparison, the code-based analysis we detailed tripped the defect, regardless of whether it was a design problem. The moral here is simple: code-based analysis techniques are not applied too late to be

valuable, and they can play a role that even formal methods applied early on cannot.

5. Conclusions

Until the late 1980s, creating reliable software systems was the driving force during software development. The process that was being employed then was fairly naive: (1) write, (2) test, (3) debug, (4) fix, and (5) go back to (2). The acknowledgement that high reliability would not be achievable via this process caused correct outputs to be deemphasized and "acceptable" outputs became the goal [3]. Getting the correct output is less important than giving harmless, incorrect outputs.

Software fault injection aids in knowing whether software meets this imperative. Software fault injection can also be used as an *indirect* measure of the quality of a physical system (in which software resides). Embedding microprocessors and software into mechanical systems has become commonplace. The types of mechanical systems that we envision software fault injection providing the greatest benefit to will consist of three main components: (1) a software control system, (2) actuators that perform the desired physical function, and (3) sensors that feed information to the software. So although software fault injection assesses the quality of the software, for software embedded in mechanical systems, that measurement is with respect to the mechanical system, and in that sense, software fault injection is a system design tool and an indirect measure of the complete mechanical system's quality.

The key to a successful first experience with software fault injection methods is properly interpreting the results. When satisfactory outputs are observed after injection occurs, it is very tempting to overclaim that the software is incapable of ever producing undesirable outputs. Nothing could be further from the truth! The correct interpretation is that the software is *only* known to be resilient to those anomalies tried; however it is likely that it is also resilient to many other anomalies. Probably the greatest benefit occurs when the software does not tolerate injected anomalies. The interpretation here is less likely to be overstated: *your software is unsafe.*

References

[1] N.G. Leveson and C.L. Turner. An Investigation of the Therac-25 Accidents. *IEEE Computer*, 26(7):18–41, July 1993.

[2] Prof. J. L. Lions. Ariane 5 flight 501 failure: Report of the inquiry board. Paris, July 19, 1996.

[3] D. R. Miller. Making Statistical Inferences About Software Reliability. Technical report, NASA Contractor Report 4197, December 1988.

[4] J. W. Moore and R. Rada. Organizational Badge Collecting. *CACM*, 39(8):17–21, August 1996.

[5] J. Voas. *PIE*: A Dynamic Failure-Based Technique. *IEEE Trans. on Software Eng.*, 18(8):717–727, August 1992.

[6] J. Voas, F. Charron, and K. Miller. Robust Software Interfaces: Can COTS-based Systems be Trusted Without Them? In *Proc. of 15th Int'l. Conf. on Computer Safety, Reliability, and Security (SAFECOMP'96)*, Vienna, Austria, October 1996. Springer-Verlag.

[7] J. Voas, F. Charron, G. McGraw, K. Miller, and M. Friedman. Predicting How Badly 'Good' Software can Behave. *IEEE*

Software, To appear in early 1997.

Jeffrey Voas *is the Vice-President of Reliable Software Technologies (RST) and is currently the principal investigator on research initiatives for ARPA, National Institute of Standards and Technology, and National Science Foundation. He has published over 75 journal and conference papers in the areas of software testability, software reliability, debugging, safety, fault-tolerance, design, and computer security. Before co-founding RST, Voas completed a two-year post- doctoral fellowship sponsored by the National Research Council at the National Aeronautics and Space Administration's Langley Research Center. Voas has served as a program committee member for numerous conferences, and will serve as Conference Chair for COMPASS'97. Voas has coauthored a text entitled Software Assessment: Reliability, Safety, Testability (John Wiley & Sons, 1995 ISBN 0-471-01009-X). In 1994, the Journal of Systems and Software ranked Voas 6th among the 15 top scholars in Systems and Software Engineering. Voas's current research interests include: information security metrics, software dependability metrics, and information warfare tactics. Voas is a member of IEEE and received a Ph.D. in computer science from the College of William & Mary in 1990.*