

# A Software Analysis Technique For Quantifying Reliability in High-Risk Medical Devices

Jeffrey Voas  
RST\*

Keith Miller<sup>†</sup>  
Department of Computer Science  
The College of William & Mary

Jeffery Payne  
RST

## ABSTRACT

*Embedded micro-processors are becoming increasingly important in medical treatments. These devices are embedded in CAT scanners, radiation therapy devices, MRIs, and are used in the control of drug production. Embedded systems are becoming more sophisticated, and the programs that control these system have increasing demands for precision and reliability. Recently, the FDA has been under increased pressure by Congress to end the paralysis within the agency in granting pre-market approval to the so-called "high-risk" devices. Complaints have been aimed at the FDA's Center for Devices and Radiological Health, which has become reluctant to approve devices. This fear, it is claimed, is due to several past controversies including the silicone-breast implants and the Pfizer Inc. heart valve. As evidence that there really is a slow-down in approval, in fiscal 1986, 72 approvals were granted, in fiscal 1991, 27 approvals were granted, and in fiscal 1992, only 12 pre-market high-risk approvals were granted [1].*

*In this paper, we will briefly present a software engineering technique called sensitivity analysis that aims at producing software that is less likely to hide faults. When sensitivity analysis indicates that faults are not likely to hide from testing, we can more comfortably rely on the results of testing. Sensitivity analysis has been automated, and preliminary results from the automated analysis suggest that the technique will have significant advantages for the development, validation, and regulation of medical devices.*

## 1 Introduction

When software controls a safety critical device, the most advanced software engineering techniques should be brought to bear to decrease the probability of catastrophic failure. This need is most obvious where human lives are immediately at risk: defense, aerospace, and medicine. The goal of software engineering is to establish methods that efficiently produce higher quality software; software that does not present us with danger. *Producing* highly reliable software for medical applications is not sufficient; we must also be able to

---

\*Reliable Software Technologies Corporation, Penthouse Suite, 1001 N. Highland, Arlington, VA 22201, (703) 276-1219.

<sup>†</sup>Supported by NASA Grant NAG-1-884.

*demonstrate* this high reliability. Unfortunately, the reliabilities desired for life critical applications cannot be demonstrated using traditional testing regimes [3].

We begin by briefly describing the problem of testing critical software to high levels of reliability. The general problem of assessing reliability to very high precision is not specific to medical applications, but it is particularly acute for life-critical medical software.

Randomly generated testing is an established method of estimating software reliability [11, 14]. Unfortunately, as software applications have required higher and higher reliabilities, practical difficulties with software testing have become increasingly problematic. These practical problems are particularly acute in life-critical applications, where requirements of  $10^{-7}$  failures per hour of system reliability translate into a probability of failure (**pof**) of perhaps  $10^{-9}$  or less for each individual execution of the software [10]. In this paper, we will refer to software with reliability requirements of this magnitude as *ultra-reliable* software.

The probability of failure of a program is conditioned on an input distribution. An input distribution is a probability density function that describes for each legal input the probability that the input will occur during the use of the software. Given an input distribution, the **pof** is the probability that a random input drawn from that distribution will cause the program to output an incorrect response to that input.

Even if ultra-reliable software can be in theory *achieved*, we cannot comfortably depend on this achievement unless we can establish that reliability for a particular piece of software in a convincing, systematic, and scientific manner. As pointed out in [3], black-box testing is impractical for establishing these very high reliabilities. In general, by executing  $T$  randomly selected tests, we can estimate a probability of failure in the neighborhood of  $1/T$  when none of the tests reveals a failure [13]. If the required reliability is in the ultra-reliable range, statistical testing would require years of testing before it could establish a reasonable confidence in this reliability, even with the most sophisticated hardware. Based on these impracticalities, some researchers believe that very high reliabilities can not be quantified using statistical methods [2, 3]. However, our research in sensitivity analysis has convinced us that their pessimism is premature.

## 2 Our Solution to The Testing Problem

We agree that statistical software testing, on its own, cannot establish ultra-reliabilities. However, software testing is not the only statistical technique possible for assessing software reliability. In this paper, we discuss a statistical technique *complementary to* testing, “software sensitivity analysis.” (We often use the shorter term *sensitivity analysis* with the software implied.) In conjunction with testing, sensitivity analysis allows us to estimate reliability to a much higher precision than was possible with testing alone.

Sensitivity analysis uses program mutation, data state mutation, and repeated executions according to the test distribution to predict a program’s minimum fault size [12]. *Fault’s size* is the probability that an input selected at random from the test distribution will cause the program to result in failure from that fault. The minimum fault size that is predicted by sensitivity analysis is the smallest probability of failure likely to be induced by any programming error resident in the code. Note that a predicted fault size does not assert that a fault is in code, but rather asserts that *if* any faults are in the code, they will be of no smaller than the prediction. A fault that is “small” is very difficult

to detect during testing, so we prefer large faults, which are likely to be detected early during testing.

A major reason that testing is difficult is that we need to determine for each input what the required output is before we can know if the software has “passed” a test. During testing whoever (the human tester, for example) or whatever (perhaps a previous version of a program being converted to a new machine) provides the correct answer is called the testing *oracle*. The oracle is often the most expensive part of testing, in terms of time. Moreover, when the oracle is a human, then mistakes by the oracle a major source of inaccuracy in the testing. Sensitivity analysis does not use an oracle, and can therefore be completely automated in a way that testing most often cannot be automated.

Sensitivity analysis and testing complement each other. Testing establishes an upper limit on the likely probability of failure; sensitivity analysis establishes a lower limit on the probability of failures that are likely to occur. Together, these estimates can be used to establish confidence that software does not contain any faults.

Software sensitivity analysis is based on separating software failure into three phases: execution of a software fault, creation of an incorrect data state, and propagation of this incorrect data state to a discernible output. This three part model of software failure [8] will be referred to as *PIE*, for Propagation, Infection, and Execution. PIE has been applied to research in program design [9], software metrics [7], and debugging [6]. In our research we apply PIE to finding a realistic minimum probability of failure estimate when random testing has discovered no errors.

In the rest of this section we give a brief outline of the three phases of sensitivity analysis. For more details, see [12], where these descriptions are elaborated. To simplify explanations, we will describe each phase separately, but in a production analysis system, processing for the phases would overlap. As with the analysis of random testing, the accuracy of the sensitivity analysis depends in part on a good estimate of the input distribution that will drive the software when it is in use.

Before a fault can cause a failure, it must execute. In this paper we will concentrate on faults that can be isolated at a single *location* in a program. A location can be defined as a single high level language statement, one machine code instruction, or some intermediate amount of computation. Our experiments thus far have defined a location as a piece of source code that can change the data state (including input and output files and the program counter). Thus an assignment statement and an **if** statement define a location, and a statement **read(a,b)** defines two locations. The probability of execution for each location is determined by repeated executions of the code with inputs selected at random from the input distribution. An automated system controls the executions and the bookkeeping.

If a location contains a fault, and if the location is executed, the data state of the execution may or may not be changed adversely by the fault. If the fault *does* change the data state into a data state that is incorrect for this input, we say the data state is *infected*. To estimate the probability of infection, the second phase of sensitivity analysis performs a series of syntactic mutations on each location [5]. After each mutation, the program is re-executed with random inputs; each time the location in question is executed, the data state is immediately compared with the data state of the original (unmutated) program at that same point in the execution. If the state differs, infection has taken place.

The third phase of the analysis estimates propagation. Again the location in question

is monitored during random tests. After the location is executed, the resulting data state is changed by assigning a random value to one data item using a predetermined distribution. (Research is ongoing as to the best distribution to use for this random selection. Current experiments use an equally likely distribution over the range of values for this variable during random testing.) After the data state is changed, the program continues executing until an output results. The output that results from the changed data state is compared to the output that would have resulted without the change. If the outputs differ, propagation has occurred.

Each phase produces a probability estimate based on the number of trials divided by the number of events (either execution, infection, or propagation). For a random test to reveal a fault, execution, infection, and propagation must occur to result in a failure. Thus the product of these estimates yields an estimate of the probability of failure that would result if this location had a fault.

Sensitivity analysis is a new empirical technique. Preliminary results concerning the accuracy of the estimates given by sensitivity analysis have been encouraging [5, 12, 6] and are summarized in Section 4.

Both statistical testing and sensitivity analysis gather information about potential probability of failure values for a program. The two techniques generate this information in distinct ways: random testing treats the program as a black box but sensitivity analysis examines the source code location by location; random testing requires an oracle to determine correctness but sensitivity analysis requires no oracle because it does not judge correctness; random testing includes analysis of the possibility of no faults but sensitivity analysis focuses on the assumption that one fault exists. In summary, the two techniques give independent predictions about the probability of failure.

### 3 *PiSCES*

*PiSCES* is RST Corporation's commercially leased tool for performing PIE on applications written in C. New front-ends to *PiSCES* for applications written in ADA and C++ are planned. Applications written in PASCAL or FORTRAN-77 can be automatically translated to C and then analyzed by *PiSCES*. However, care must be taken to assure that the translation occurs correctly. There are peculiarities that can occur that are machine/language specific, and these instances require additional attention, including custom programming.

*PiSCES* is written in C++, and provides the user with several different back-ends for the way in which the raw numbers that the PIE algorithms produce is displayed. For example, there is a sensitivity analysis back-end on *PiSCES* that performs the failure rate prediction. The prediction can be produced for the entire program, a module, or a single statement. Also, there is an option to only produce the raw numbers generated during analysis; as with the failure rates, the raw numbers can be generated for the program, a module, or a statement.

## 4 Results

Several studies on the effectiveness and preciseness of PIE have been performed; we will summarize the results of three of these experiments here.

[12] showed that for battlefield simulation of approximately 2000 lines, PIE produced predictions of the failure rate had a correlation coefficient of 0.97 with the true failure rate. [5] showed how to apply PIE's algorithms to the task of ranking test inputs (similar to what is done in mutation testing), meaning determining how good particular inputs were at revealing faults. In this experiment, out of 1,000,000 inputs, 10,000 were found that were thought to be the best, 10,000 were found that were thought to be the worst. In experiments with these two sets of inputs, it turned out that those thought best caught almost twice as many faults as those thought to be the worst. [4] applied *PiSCES* to several versions of NASA's RSDIMU programs. The programs we used had actual faults, and the true failure rate was estimated. Also, a prediction of the minimum non-zero failure probability was found with *PiSCES*. In these experiments, we underpredicted the actual failure rate; Thus we were able to accurately predict how much testing would have been needed to catch the actual faults in the code.

## Concluding Remarks

It is premature to claim that sensitivity analysis will dramatically sharpen probability of failure predictions for life critical medical applications being written today. However, it appears that sensitivity analysis provides information about the likely size of software faults, and that sensitivity information can be combined with testing to facilitate more precise software quality assessments. Stringent testing standards can set a lower bound on the likely size of a software fault in a particular piece of software; it appears that sensitivity analysis can set an upper bound on the likely size of any software fault in that same piece of software. When the software involved controls life critical medical devices, this added information takes on added significance.

## References

- [1] B. INGERSOLL. FDA Attacked for Holding up Medical Devices, *Wall Street Journal*, 1992.
- [2] R. BUTLER AND G. FINELLI. The infeasibility of experimental quantification of life-critical software reliability. In *Proceedings of SIGSOFT '91: Software for Critical Systems*, pages 66–76, New Orleans, LA, December 1991.
- [3] D. R. MILLER. Making Statistical Inferences About Software Reliability. Technical report, NASA Contractor Report 4197, December 1988.
- [4] J. VOAS, J. PAYNE, C. MICHAEL AND K. MILLER. Experimental Evidence of Sensitivity Analysis Predicting Minimum Failure Probabilities. In *Proc. of Eighth Annual Conference on Computer Assurance.*, pages 123–133, National Institute of Standards and Technology, Gaithersburg, MD, June 1993.
- [5] J. VOAS AND K. MILLER. The Revealing Power of a Test Case. *J. of Software Testing, Verification, and Reliability*, 2(1):25–42, May 1992.
- [6] J. VOAS AND K. MILLER. Applying A Dynamic Testability Technique To Debugging Certain Classes of Software Faults. *The Software Quality Journal*, 2:61–75, 1993.

- [7] J. VOAS AND K. MILLER. Semantic Metrics for Software Testability. *J. of Systems and Software*, 20:207–216, March 1993.
- [8] L. J. MORELL. Theoretical Insights into Fault-Based Testing. *Second Workshop on Software Testing, Validation, and Analysis*, pages 45–62, July 1988.
- [9] J. VOAS, K. MILLER, AND J. PAYNE. Designing Programs that are Less Likely to Hide Faults. *J. of Systems and Software*, January 1993.
- [10] I. PETERSON. Software failure: counting up the risks. *Science News*, 140(24):140–141, December 1991.
- [11] T. A. THAYER, M. LIPOW, AND E. C. NELSON. *Software Reliability*(TRW Series on Software Technology, Vol. 2). New York: North Holland, 1978.
- [12] J. VOAS. PIE: A Dynamic Failure-Based Technique. *IEEE Trans. on Software Engineering*, 18(8):717–727, August 1992.
- [13] K. MILLER, L. MORELL, R. NOONAN, S. PARK, D. NICOL, B. MURRILL, AND J. VOAS. Estimating the Probability of Failure When Testing Reveals No Failures. *IEEE Trans. on Software Engineering*, 18(1):33–44, January 1992.
- [14] S. N. WEISS AND E. J. WEYUKER. An extended domain-based model of software reliability. *IEEE Trans. on Software Engineering*, 14(10):1512–1524, October 1988.