Predicting How Badly "Good" Software can Behave

Jeffrey Voas*, Frank Charron, Gary McGraw
Reliable Software Technologies Corporation

Keith Miller
University of Illinois at Springfield

Michael Friedman
Hughes Electronics Corporation

ABSTRACT

This paper presents a fault-injection methodology that predicts how software will behave when: (1) components of the software fail, (2) hardware components external to the software fail, (3) human factor errors occur and bad input is provided to the software, and (4) the software is executing in unlikely operational modes. Because of the enterprise-critical nature of many of today's software systems, it is vital that these system are robust enough to handle problems that originate externally as well as the expected problems that will arise from internal defects. Also, this paper presents four cases studies that highlight the benefit of this analysis for both safety-critical systems and non-safety critical systems.

KEYWORDS

fault-injection, software failure tolerance, fault tolerance, input distribution, software testing, software quality

1 Introduction

Software failure is the nightmare of the Information Age. Vast software engineering resources are ubiquitously spent throughout all industries to avert this nightmare. However, in the end it is impossible to guarantee that software is perfect, nor is it possible to predict and eliminate every possible gliche that could attack software from the outside world as it executes. Thus it is critical to devise ways to exploit the fact that software faults are reproducible and the fact that software is inherently malleable in order to reduce the risk of catastrophic failure of computer-based systems. This paper describes an automated system that makes use of software malleability and fault reproducibility to help identify weaknesses in software that could cause

*Voas is correspondence author at RST Corp., Suite 250, 21515 Ridgetop Circle, Sterling, VA 20165.

catastrophic disasters and to help pinpoint locations in the code where these weaknesses lie. The automated software analysis environment is called the *PiSCES Safety Net* (*PSN*).[1]

The *PiSCES Safety Net* is an implementation of a software analysis technique called "extended propagation analysis", or *EPA* [7, 15]. *EPA* uses the injection of artificial faults (of both software and hardware varieties) to test software's tolerance to unusual events. Artificial faults simulate problems occurring both internally and externally to the software. Although there is extensive literature on the use of artificial faults in software testing [13, 12], the commercial adoption of such techniques has been slow, due to both practical and theoretical limitations. Though *EPA* is certainly related to previous work in artificial fault injection, it is distinct in two important ways:

1. *EPA* does not worry about correctness. Instead, it worries about software output behavior — the avoidance of dangerous and sometimes life-threatening outcomes.

2. Because correctness is not its focus, *EPA* does not require an oracle. After the user identifies undesirable outcomes, the analysis of whether a program can produce such is almost completely automated.

EPA takes as its input a program, an operational profile, and a description of unsafe (or unacceptable) outcomes. *EPA* returns as its output places in the program where it has discovered potential weaknesses based on the fault-injection experiments that it employs. This paper describes the theory behind *EPA* and discusses the results of four real-world case studies. In the four case studies, *EPA* was applied to four distinct software control applications, two of which are safety-critical. The results of these case studies suggest that *EPA* is a valuable technique for anyone concerned with the quality of the outputs that software produces.

1.1 Extended Propagation Analysis Differs from Traditional Measures of Fault-Tolerance

Traditionally, *fault tolerance* refers to building subsystems from redundant components that are placed in parallel to ensure higher quality systems [2]. Unlike traditional fault tolerance, *EPA* not only *seeks* to aid developers in increasing quality, but it also *measures* the quality of software outputs. In order to measure "how good" software outputs are, we introduce the notion of "failure tolerance." Software is *failure-tolerant* if and only if:

1. the program is able to compute an *acceptable* result even if the program itself suffers from incorrect logic, and

2. the program, whether correct or incorrect, is able to compute an *acceptable result* even if the program itself receives *corrupted* or *malicious* incoming data during execution.

This paper presents a simple metric for assessing how failure-tolerant a software system is, where a score of 1.0 means totally tolerant and a score of 0.0 means complete intolerance. Obviously, the most critical aspect in this definition is "acceptable result", which is simply an output result that is in the range of acceptable values for a given input. We will not concern ourselves on minute details concerning acceptability; the user can define acceptability to the nth decimal place if his or her computer has enough precision. In *EPA*, acceptable

[1]The *PiSCES Safety Net* is one module of the PiSCES Software Analysis Toolkit®[2].

results are defined directly by the user from the requirements of other subsystems that the software sends its outputs to. Thus acceptable results can be made to exhibit characteristics such as correctness, timeliness, and/or safety.

2 Extended Propagation Analysis: Theory and Implementation

EPA works by instrumenting source code with functions that cause corrupted internal states. In principle, this is similar to a predecessor technique known as *error seeding* [9]. As such, *EPA* is meant to be applied late in the software life-cycle, near the time when code certification occurs. *EPA* requires certain information in order to do its job: (1) inputs that some code can execute on, and (2) system safety requirements that explicitly define the software outputs that are unacceptable. Ideally, these safety requirements will include information about the failure-modes of external components that feed input to the software. (For example, *EPA* can make use of the fact that when some particular sensor fails it returns a constant signal with the value -1.)

The *PSN* implementation of *EPA* is the most recent (and advanced) version of this technology; it contains the most recent performance optimizations that have been learned from applying earlier prototypes. *PSN* is designed to work as a command-line program that is activated at compile-time (for example within a `makefile`). Whenever the software is compiled, *PSN* instruments the code so that it can perform extended propagation analysis. EPA results are dynamically gathered as the software executes.

As *EPA* runs, it alters data-state values during program execution (using fault-injection) and collects information concerning whether program outputs are *unacceptable* given the corrupted run. *EPA* considers an output "unacceptable" if it is: incorrect, non-existent (meaning that the code fails to halt in a reasonable amount of time), unsafe, etc. As we will further explain below, *EPA* is a natural extension of the basic hypotheses on which *fault-tree analysis* is based. By contrast to the usual approach, however, *EPA* employs fault-injection to play "what-if" scenarios, much like a Failure Mode Effect and Criticality Analysis [4]. *EPA*'s dynamic approach is computationally expensive in terms of cpu time. Fortunately *EPA* can be almost fully automated, leading to a software safety analysis technique that exploits readily-available computational power instead of relying on human intuition. *EPA* still requires some manual set-up effort that involves telling *EPA*: (1) where in the code to perform fault-injection, and (2) what constitutes as an unacceptable output.

For your software to be failure-tolerant, two classes of problems must be protected against: software faults that you enter into the code and external hardware or external software failures (from someone else's code) that release erroneous inputs into your software. *EPA* can simulate both classes of problem in order to observe the impact of either type of failure. For example, *EPA* can determine what may happen should a a hardware sensor fail *in addition to* determining what the effects of software failure may be. *EPA* can also simulate the failure of a call from your program to a system utility. The simulation of these types of events enables *EPA* to predict future impacts to critical output variables. In this way, *EPA* helps the user identify a large set of software events that could lead to a catastrophe.

2.1 The Extended Propagation Analysis Theoretical Model

We now explain *EPA*. We begin with some formal definitions. Let P denote our program, x denote a program input value, Δ denote the set of all possible, valid inputs to P, Q denote the probability distribution of Δ, l denote a program location in P, and i denote a particular execution (or what we term an "iteration") of location l caused by input x. Finally, let $\mathcal{A}_{lPix}$ denote the data state produced after executing location l on the i^{th} execution from input x.

A *data state error* is an incorrect variable↔value pairing in a data state, where correctness is determined by an assertion placed between program locations (*i.e.*, statements).[3] We also refer to a data state error as an *infection*, and use these two terms interchangeably below. If a data state error exists, the data state and variable with the incorrect value at that point is said to be *infected*. A data state may have more than one infected variable. *Propagation* of a data state error occurs when a data state error has some noticeable effect on the output.

It is important to group data states into sets with similar properties. Let ν_{xl} represent the number of times that location l is executed by input x. All of the data states created by x immediately after l is executed are denoted by:

$$\mathcal{A}_{lPx} = \{\mathcal{A}_{lPix} \ \mid \ 0 \leq i \leq \nu_{xl}\}$$

Sets of these sets can be created for all inputs in Δ as follows:

$$\alpha_{lP\Delta} = \{\mathcal{A}_{lPx} \ \mid \ x \in \Delta\}$$

A *simulated infection* occurs when a variable of some data state (having either an initial value or an undefined value) is forced to some new value. In a sense, a simulated infection is merely a simulated data-state error. The relationship between simulated infections and potentially disastrous program states is essential to the success of *EPA*. *EPA* employs simulated infections to enable the user to observe the impact of different classes of "corrupted states" some of which may be "bad states". When a system gets into a "bad state" during execution, it would be ideal if the next event to occur were a recovery from the bad state into a correct state. If this is impossible, then it would be desirable at least to demonstrate that the "bad state" has no hazardous consequences.

Simulated infections mimic the impact of programmer faults and external software or hardware failures on program data states. The infections are simulated by a set of perturbation functions. The process of injecting a simulated infection into the data state of an executing program is termed *perturbing* the data state. A *perturbation function* is a mathematical function that takes in a data state as an incoming parameter, changes the data state according to certain parameters, and produces as its output a different data state. A data state that has been changed is said to have been *perturbed*. Users of this technique have complete control over the perturbation functions and can define their own perturbation functions or use some default perturbation functions that we have published elsewhere [5].

EPA detects when particular types of output events or program data-state events occur by making use of a user-defined logical predicate called $PRED$. $PRED$ either relates specific output variables (and

[3]This is admittedly tenuous, since for any specification there exists an infinite number of correct programs that implement that specification. Furthermore, after each statement in some implemented version of the specification (*i.e.*, a particular program) we are required to place an assertion if we are to expose data state errors. As a practical matter, this will never happen.

combinations thereof) to acceptable value ranges or it relates program data-state variables (used at particular times during code execution) to value ranges. For example, if *EPA* is used for assessing whether a program is likely to produce hazardous outputs, then $PRED$ should be a predicate describing different types of software hazards. Or $PRED$ might be "attached" to a program statement in order to track a certain data-state variable to make sure that the variable never exceeds some value. $PRED$ will likely contain information about (program input) x that can be used to determine the acceptability of outputs or data-states.[4] The following algorithm calculates the proportion of outputs that satisfy $PRED$ after simulated infections are injected at location l:

Algorithm 1:

1. Set **count** to 0.
2. Randomly select an input x according to input distribution Q (which will hopefully be the operational distribution or at least the test distribution of P), and execute P on x until $\mathcal{A}_{lP1x}$ exists as the current state of P or until P halts or P times-out. Set $\mathcal{Z}$ to $\mathcal{A}_{lP1x}$,
3. Alter the sampled value of program variable a found in $\mathcal{Z}$ with a simulated infection creating a different program state, $\breve{\mathcal{Z}}$, and execute the succeeding code with state $\breve{\mathcal{Z}}$. If l is executed more than once for this execution of P, alter a in each state created immediately after l.
4. If P's output satisfies $PRED$, increment **count**.
5. Repeat steps 2-4 n times.
6. Divide **count** by n yielding ψ_{alPQ}. Note that $1 - \psi_{alPQ}$ is the measure of failure tolerance that we assign to location l.

An interesting question often arises as to why a is altered in each state after l in Step 3. a gets corrupted on the first iteration, so why corrupt it repeatedly? The reason that this is done is because we are simulating the execution of some unknown fault classes that might continually corrupt the state at this location. Also, it seems implausible that true faults would only corrupt on the first iteration and never again. Of even more interest here is the fact that the manner of altering on multiple iterations is not necessarily consistent, and is instead sometimes random. Interestingly enough, preliminary evidence suggests that this does not bias the results in a manner inconsistent with how real faults behave.

The hope for the results from this algorithm is that ψ_{alPQ} is 0.0, meaning the failure tolerance $(1-\psi_{alPQ})$ is 1.0. Whenever this is not the case, the perturbation functions used during Step 3 will have revealed the possibility of a weakness in the code.

2.2 Rare Input Events

A classic problem in measuring the quality of software's behavior is that it is sometimes difficult to determine what input distribution best represents how some software will be used in practice. You would expect that given two subtlely different but almost identical distributions, q' and q'' , the impact of the differences on

[4] Currently, $PRED$ is defined for single-point outputs, and not for a sequence of output events; work to remove this limitation is ongoing.

the quality of the behavior would be small. What would you expect if q' and q'' were complete opposites, meaning that q'' is the compliment of q'? Would you expect small differences on the quality of the behavior? Or would you expect enormous differences? This question intrigued us to the point of building a distribution inversion capability to quantitatively begin answering it.

We will now summarize a hypothesis that we had from the outset regarding the effect on software's behavior when it operates in a mode best described as the complete opposite of the operational profile:

> Suppose Q is the *operational profile*, n is a relatively small number, and $\mid \Delta \mid$ is large. In this case we hypothesize that not only do there exist many test cases that are not being used with the *EPA* algorithms (or being used during testing according to Q for that matter) but also the likelihood of any of the rarest (most unlikely) test cases getting selected is extremely small. To compensate, one should force the selection of some of the lowest probability events in Q in order to assess how robust the code is in the rarer scenarios.

(This is slightly different than the well-known testing strategy known as *boundary-value* testing, because boundary-value testing picks test cases that appear near the minimum and maximum values allowed, whereas this approach picks rare test cases, that might not at all be near the minimums nor maximums.)

During testing according to some distribution Q, the majority of the test cases selected are from the subset of Δ made up of cases that are the most likely to be selected. This is true even if testing an enormous number of times. As more and more samples are picked according to the assumed operational profile and run, the potential of repeating particular test cases (or at least running test cases that do not exercise any novel functionality or forge new paths) becomes greater. Running more and more test cases according to the same distribution does improve the chances of selecting an "infrequent test case" (from an area of the input space which we term the "ultra-rare" region), but only slightly. In some cases (*e.g.*, if the distribution has a small variance), selecting members of the ultra-rare region may require sampling an intractable number of test cases. When an operational profile is inverted in order to sample "rare" test cases, the "rareness" of these cases stems solely from the infrequency with which they are sampled during operational use. This does not mean that such cases are necessarily a small subset of Δ. In fact, it is possible that the "rare" test cases represent a substantial portion of Δ.

This suggests that a marriage between EPA and the ultra-rare test cases would produce meaningful data concerning software's failure tolerance during unusual operational events.[5] When the software does not perform robustly during experiments with unusual inputs, chances are that it will not perform acceptably if unusual events occur during deployment. Likewise, if the software does behave robustly, this suggests that the software is probably sufficiently hardened against anomalous events. Such indicators are not guarantees of future failure-tolerant behavior of the code, but they clearly suggest that such behavior will occur.

3 Practical Results

The information provided by *EPA* identifies locations in your source code from which particular types of software failure could originate if states become infected. One important application of our algorithms is to

[5]One caveat here: there are operational distributions that when inverted produce the original distribution. Hence this idea of inverting a distribution will only be applicable to operational distributions with spikes (*i.e.*, large variance in the probabilities).

identify where specific software hazards can originate, so that safety improvements can be undertaken. If there exists any $1-\psi_{alPQ}$ that is < 1.0, it immediately suggests that location l needs improvement. That is, such a location is a candidate for an assertion that insures that some particular class of data states cannot occur at the location.

EPA has been used extensively in several industrial projects, providing valuable objective data regarding the effectiveness of fault-injection methods for assessing software's "behavioral" quality. In this paper, we have chosen to highlight four case studies that we feel best demonstrate the utility of *EPA*. The first two studies involve small avionic systems exercised using inverted operational profiles. (The differences between *EPA* runs performed with the original distributions and *EPA* runs performed with the inverted distributions were quite interesting, as we shall show.) The second two case studies are real-world applications involving an advanced, automatic train control system and the control software for a high-risk medical device.

3.1 Case Study 1: Yaw

Our first case study involves a software module, `yaw.c`. This application is a small yawdamp controller for a 737 aircraft delivered to NASA for research purposes. The set of inputs provided to us by NASA can be nicely fit to a *Gaussian* distribution with parameters $\mu = 0$ and $\sigma = 2/3$, for the single input variable, `YAWDAMP_STATE`, in the `main()` subroutine. Our experiment was conducted by randomly generating one suite of 100 test cases, where the Gauss(0, 2/3) distribution was used for the `YAWDAMP_STATE` variable values, and another suite of 100 test cases where the inverted Gauss(0, 2/3) distribution was used for the `YAWDAMP_STATE` variable values. Figure 1 shows a histogram representing the two test sets. (The y-axis represents the probability of a particular value occurring for `YAWDAMP_STATE` variable. The `yaw` program was then executed using the original Gauss(0, 2/3) test suite. The results from analysis performed during these runs are shown in the middle column of Table 1. (In the following tables, bold-faced results are the results of more interest, and an * indicates a location in the code that is never exercised.) The total failure tolerance of the `yaw` program was estimated to be 0.435. Further investigation of the results revealed that the `lim()` procedure in the `yaw.c` module was responsible for limiting the failure tolerance of the entire program.

The `yaw` program was then executed using the inverted Gauss(0, 2/3) test suite. The results from analysis performed during these runs are shown in the right column of Table 1. The rare-event failure tolerance estimated for the `yaw` program dropped to 0, which was again limited by the score assessed in the `lim()` procedure. If the results from instrumented locations of source code within the `lim()` procedure are examined in both test cases, we discover that for location 286, the inverted Gauss test suite resulted in a rare-event failure tolerance score of 0.0, while the original Gauss test suite resulted in a score of 0.714.

The differences in the scores resulting from running the two different test suites provide a good example of the benefits that can be realized by using inverted distributions. The score of 0 returned from running the `yaw` program on the inverted test suite does not mean that this test suite has revealed safety faults in the code. However, it does call attention to places in the code that may cause hazard conditions — places that are not obviously revealed with tests using the original operational distribution.

3.2 Case Study 2: Auto-pilot

The `b737` program is a more complex piece of C source code, providing the auto-pilot controls for a B737 airplane. It was provided to us for research purposes only by NASA. The code was generated with a CASE

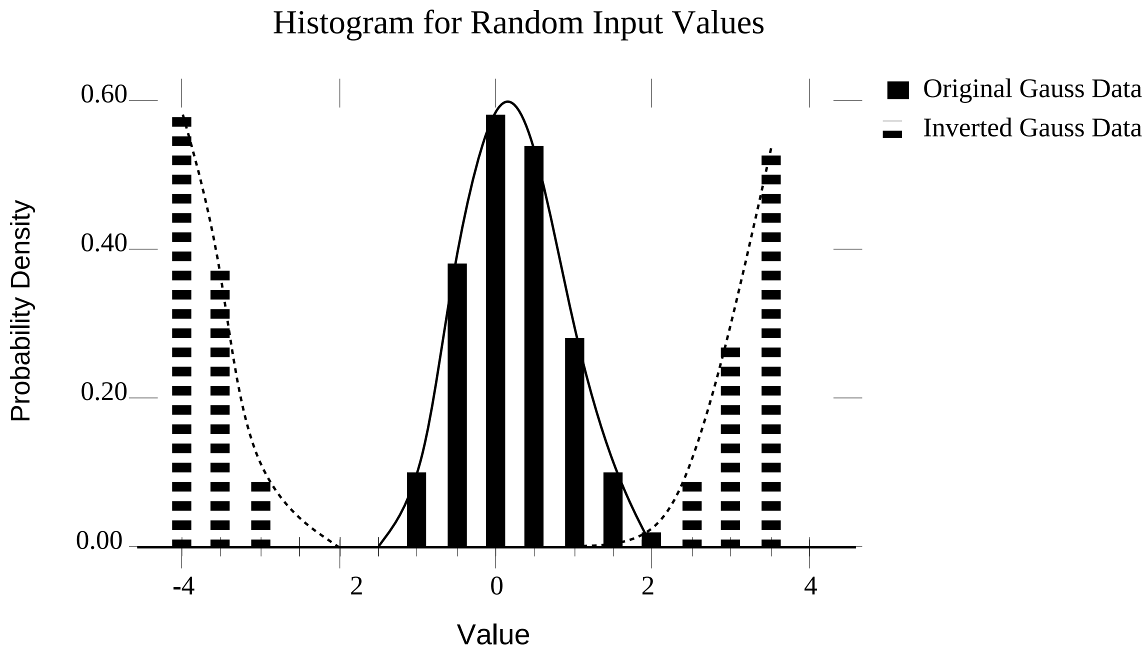

Figure 1: A histogram showing two sets of random input values generated for the `YAWDAMP_STATE` variable in `yaw.c`. The inverted Gaussian distribution represents test cases from the "ultra-rare" region.

Source code	Failure tolerance Gauss(0, 2/3)	Failure tolerance inverted Gauss(0, 2/3)
gauss_set.sfr	0.434783	0
main.c	1	1
GetDouble	1	1
GetBool	1	1
main	1	1
yaw.c	0.434783	0
YAWDAMP	0.72	0.45
WOUT	1	1
lim	**0.434783**	**0**
Line 280:	0.77	0.51
Line 282:	0.434783	0.5306
Line 286:	**0.714286**	**0**
Line 288:	0.454545	0.5294
Line 292:	0.981818	1
FCAS	1	1
LINTERP	1	1

Table 1: **Selected results for the execution of the yaw program.**

tool and has the functionality of a portion of an avionics flight control system. Given data for the inputs into the b737 program, we were able to select distributions to approximate the expected usage for several of these input variables. These selected distributions were then inverted. One test suite of 100 input sets was generated using the selected operational distributions, and another test suite of 100 input sets was generated using the inverted distributions. (There is no significance to 100 test cases being chosen; that number was selected arbitrarily.)

Source code	Failure tolerance original	Failure tolerance inverted
b737.sfr	0.95	0.5
b737ansi.c	0.95	0.5
LIMITER	0.96	0.54
Line 117:	1	1
Line 121:	1	1
Line 125:	**0.96**	**0.54**
Line 130:	1	1
LONGAP	1	0.5
Line 1813:	1	1
Line 1820:	1	1
Line 1830:	1	1
Line 1834:	1	1
Line 1845:	1	1
Line 1851:	**1**	**0.5**
Line 1856:	1	1
Line 1857:	1	1
Line 1859:	1	1
Line 1860:	1	1
CAS_ADJ	0.95	0.6
Line 2247:	0.95	0.6
Line 2248:	**0.98**	**0.64**
mainansi.c	1	1

Table 2: **Selected results for the execution of the b737 program.**

EPA was applied for each test suite, using identical hazard conditions specifications. Selected results from the failure tolerance analysis are displayed in Table 2. The original distribution test suite resulted in a high failure tolerance score of 0.95, contrasted with the low score of 0.5 resulting from the inverted test suite. Comparing the functions and location scores for the two sets of analysis results, we discover that at several locations in the code, the inverted test suite reveals significantly lower failure tolerance scores than the original test suite.

Fault tolerance scores of 1.0 for Line 1851 of function LONGAP and 0.95 for the entire CAS_ADJ function, as returned from the analysis performed using the original distributions, would not cause much concern regarding those parts of the program. However, the rare-event failure tolerance scores resulting from the inverted distribution test run, 0.5 for Line 1851 and 0.6 for CAS_ADJ, indicate that some additional failure-tolerant mechanisms may be warranted. As in the yaw case study, the b737 results demonstrate that running programs with inputs drawn from the rare-event space can demonstrate lower failure-tolerant scores than inputs only drawn from the normal operational profile.

The results from Experiments 1 and 2 are not conclusive, but they are suggestive. They show that when distributions radically change, the tolerance of a particular program to corruptions in its program states can radically change. If the failure tolerance improves for the inverted distribution (when compared to the original), and the failure tolerance for the original distribution was acceptable, this is a nice bonus. If the failure tolerance decreases to an unacceptable level, however, we must be somewhat concerned about the robustness of our system when it is operating in unlikely modes. Clearly the most interesting result from this case study is that divergent operational profiles can produce quite divergent results, and hence the benefit of having an accurate operational distribution is again demonstrated [10].

3.3 Case Study 3: Magnetic Stereotaxis System

Our third case study involves a real-world control problem in a medical domain — the Magnetic Stereotaxis System (MSS). MSS is an experimental device for performing human neurosurgery being developed in a joint effort between the Department of Physics at the University of Virginia and the Department of Neurosurgery at the University of Iowa. The system operates by manipulating a small permanent metal seed that is moved throughout the brain using an externally applied magnetic field. By varying the magnitude and gradient of the magnetic field, the seed can be moved, positioned at a site requiring therapy (*e.g.*, a tumor location), and used for ablation and/or cauterization of the diseased tissue. The magnetic field required to move the seed is extraordinarily powerful. The field is produced by six superconducting magnets that are located in a chamber that surrounds the patient's head. Dr. John Knight of UVA provided us with access to the source code for an early version of the control software for the MSS; what relationship if any our code has with the code to the real system that is slated to undergo FDA approval is unknown [16]. Further information concerning the operation of the MSS is provided in [3].

EPA was applied to a portion of the MSS responsible for controlling the current level in the coils. In the MSS, the coil current is adjusted by the controller subsystem in order to create a magnetic field that will move the seed to a specified location in the brain. The coil controller code contains functions that communicate with the hardware, reading and setting coil parameters. It is critical that when a new current level is set by the coil controller, it falls within a specified range. If the current-level limits are exceeded, the consequences could be fatal for the patient. If any one of the six electromagnets or one of the six power supplies fail, a distorted magnetic field around the brain could occur. In this case, the software has been designed to quickly shut down the other electromagnets in order to halt the seed and prevent patient injury.

The coil control subsystem is written in C++. A test program for the coil controller was built, and these source files were instrumented for failure tolerance analysis. A hazard condition was specified in the `set_current` subroutine, by defining an upper bound on the 12 bit integer representation of the current value passed on to the coil hardware. If this upper limit is exceeded, patient safety is compromised. Test cases were generated and then supplied to the coil control driver program for the failure tolerance analysis. The numbers in Table 3 show the results of performing failure tolerance analysis on the coil control driver program with the test cases generated.

The failure tolerance scores indicate that for 52% of all program executions tried, the specified hazard resulted. These results shown in Table 3 were then used to examine the coil controller code at the places where the EPA failure tolerance scores are lowest. A closer inspection of location 90 of the `set_current` function reveals a conditional statement that appears to filter out current values (stored as reals) that are outside a defined range. As long as the minimum and maximum current values are the same as the 12-

Source code	Failure tolerance Gauss
test_servo1.sfr	*** 0.48008386**
coil.cc	* 1
basic_functions.c	* 1
ramp_mdl.cc	* 1
servoamp.cc	* 0.48008386
servoamp::servoamp	* 1
servoamp:: servoamp	1
servoamp::set_current	**0.48008386**
Line 89	1
Line 90	**0.48008386**
Line 94	**0.83857442**
Line 95	1
Line 96	**0.95387841**
servoamp::get_actual_current	* 1
servoamp::get_current_settings	* 1
servoamp::read_fault	* 1
servoamp::inhibit	* 1
servoamp::uninhibit	* 1
servoamp::get_amp_status	* 1
servoamp::read_config_file	1
servoamp::adc_to_amps	* 1
servoamp::amps_to_adc	* 1
servoamp::amps_to_dac	0.83857442
Line 350	0.83857442
servoamp::dac_to_amps	* 1

Table 3: **Selected results from the MSS.**

bit unsigned integer representations used in the hazard condition limits, this conditional statement does not appear to negatively impact the program safety. However, the 84% failure tolerance estimate for location 350 in the `amps_to_dac` function (responsible for the 84% estimate for location 94 in `set_function`) indicates a possible problem in the code. The 95% score for location 96 in `set_current` further emphasizes the potential for a problem. Although the conditional statement in location 90 is used to assert that the current level is within a safe range, *the current value is modified in the subsequent statements*, including a function call to `amps_to_dac`. The converted 12-bit integer representation of the current level is sent to the coil hardware without an assertion! This suggests that if the data should somehow become corrupted or incorrectly converted in the statements after the initial assertion, an unsafe value could potentially result and be sent on to the coils for current adjustment.

After discovering this mistake in the code, a fix was easily implemented by moving the current-value assertion down in the code so that it occurs *after* all value transformations have been made. Tests on a version of the MSS coil controller code that had been "patched" in this way resulted in fault tolerance scores of 100%. The magnitude of the difference in score values is not important and can be ignored. What is important is the *relative* improvement from one iteration to the next of the actual safety of the code under analysis. The safety measures show objectively that improvement has been made.

An additional benefit that stems from performing our failure tolerance estimation method on the coil control subsystem is that the failure tolerance of the software in the event that corrupt data is received from the hardware can be assessed. Perturbations applied to values returned from the hardware can be used to determine the ability of the subsystem to tolerate the occurrences of invalid data from hardware.

In summary, the results of Case Study 3 isolated a potential weak spot in the coil controller code. Specifically, we isolated a region of the code that allows the current level to be sent to the coil hardware without a check on the hazardous current levels. If the current level is corrupted and is not checked prior to being sent to the coils, then a hazardous situation appears eminent. After discovering this error with *EPA*, a “patch” that objectively improved the safety of the software was easily applied. In addition to this impressive result, *EPA* identified several locations in the code that were not exercised by the approximately 400 (boundary value) test cases that we employed, hence suggesting that simple test coverage analysis would have provided useful results. This information could be of use in the development of a more thorough set of tests.

3.4 Case Study 4: Bay Area Rapid Transit

Our fourth example involves another real-world control problem, the Advanced Automated Train Control (AATC) system [1]. Hughes Aircraft Company, El Segundo, CA, in partnership with the San Francisco Bay Area Rapid Transit (BART) district and Morrison Knudsen Corporation, is producing this new train control technology with partial funding from the US government’s Advanced Research Projects Agency (ARPA). In this innovative control system, the head and tail cars of each BART train are equipped with spread-spectrum radios. Additional radios are situated along the trackside and at stations. The trains in a station’s “control zone” form a wireless local area network with time-division multiplexing. This wireless network provides not only data communications between the trains and the station but also provides, through radio ranging techniques, information about the location and speed of the trains. A computer at the station uses this information to calculate safe speed commands, which it sends to the trains via the wireless network. The system provides BART with the ability to increase the throughput of trains through BART’s Transbay Tube from the current 16 trains/hr to a minimum of 30 trains/hr. The new system will enable BART to locate the position of trains with an accuracy of +/- 15 ft and support control algorithms that will allow the train service to operate with greater “throughput” and reduced braking, which will result in more efficient use of energy and a higher quality of service to patrons.

While almost all previous control systems have been primarily hardware based, the AATC is primarily software based, which presents both opportunities and vulnerabilities. The software is written in a restricted subset of the C++ programming language. Hughes has incorporated the *Pisces Safety Net* into its methodology for developing safety-critical software and is using the tool to analyze the station computer software. Basically, hazards were identified and a fault tree was developed that depicted all credible ways that a hazard could occur from the causal and logical interrelationships of system events and conditions. Certain leaves of the fault tree related to particular software output being incorrect. These hazardous output conditions were expressed in terms of output program variables. *EPA* assisted Hughes in identifying locations where additional safety preconditions, postconditions, and assertions were inserted to mitigate the hazard. The terms “preconditions” and “postconditions” are used in an informal sense here; they are safety assertions that are evaluated upon entry to and exit from a “safety port function.” *Safety port functions* are a small subset of the modules function which provide the interface to other modules.

In running one test scenario (on a subset of the code responsible for initializing the system’s databases, controlling the local area network, and processing the data from the trains), Hughes analyzed about 2000 locations. The analyzed locations that resulted in $1 - \psi_{alPQ}$ lower than 1.0 (about 26) were subjected to a larger number of test cases. Fourteen of the 26 location caused core dumps as their failure mode. This is considered a safe state in the context of the AATC system (although undesirable for performance) since trains are designed to revert to a backup system or stop safely when they cease receiving speed commands. Hughes was able to take corrective actions at the remaining 12 code locations with the result of bringing the failure tolerance score to 1.0. Many of these locations had not been found by any other software safety methodology that was being employed. The tool was able to uncover these potential problem-areas early, in the Integration Test phase, where corrective action could take place relatively inexpensively. *Safety Net* also provided Hughes with a double-check on which software functions were safety critical.

This case study contains a review of results from 4 different analysis points which resulted in failure-tolerance scores of 0.0. At Analysis Point 1, the perturbation function caused a routine to be called with unexpected input parameters. This resulted in the train position being off by one track segment. The mitigation was to perform a precondition check to further validate the input parameters and report errors and/or stop processing of the data if necessary.

At Analysis Point 2, the perturbation caused the software to process a range report as though the trains were located before gross position location had been achieved. The result was that the train position was off by many track segments. The mitigation was to perform a precondition check that gross positioning has been performed prior to processing the report as located.

At Analysis Point 3, the perturbation function caused the lead radio set ID to change. Since validation had already been performed earlier in the code, the program assumed that if the train radio set ID was not equal to the first radio set ID, it would be equal to the second radio set ID. This caused the second radio set ID to be the front of the train, resulting in the train position off by the length of the train. The mitigation was to add an assertion to the conditional location to verify that the train radio set ID equals the second radio set ID before making the second radio set ID the lead of the train.

At Analysis Point 4, in the database initialization code, *EPA* pointed out five areas in the code where data read in during database initialization should be protected with an error-detecting code. A cyclic redundancy code (CRC) check was added to protect this data.

Note again that these cases do not represent actual faults in the code. *EPA* only detects locations in the code where, if anomalies hypothetically existed and were triggered during execution, undesirable outputs could occur.

3.5 Other Applications

EPA, together with its derivative techniques of Sensitivity Analysis [8] and Adaptive Vulnerability Analysis [6], has demonstrated the potential for addressing a wealth of different software quality concerns. In this article, we have demonstrated *EPA*'s application to four projects, two of which are real-world control applications. We now highlight other applications that are amenable to the *EPA* methodology.

Recently, NASA published an Interim Report (NSS1740.13) containing software safety requirements for procured software and in-house critical software. The report states that the Code Safety Analysis phase of development must:

> Identify potentially unsafe states caused by input/output timing, multiple events, out-of-sequence events, failure of events, adverse environments, deadlocking, wrong events, inappropriate magni-

tude, improper polarity, and hardware failure sensitivities, etc. [11]

Underwriter's Laboratory also has a software safety standard (UL1998) in place. It states:

> These requirements [meaning the standard] address risks that may occur as a result of faults caused by software errors, such as the following: ...b) Coding errors, including syntax, incorrect signs, endless loops, and the like; c) Timing errors that can cause program execution to occur prematurely or late; d) Induced errors caused by hardware failure; e) Latent errors that are not detectable until a given set of conditions occur;...[14]

These standards explicitly spell out hazardous output events and internal events that are undesirable. Along with an appropriate set of simulated infections, data state assertions, and output events, *EPA* is able to provide an important relative measure of code robustness. The frequency with which undesirable output events crop up can be used to assess whether problems such as coding errors and infinite loops trigger actual safety hazards.

Fault-tree analysis applied to a system of which software is a component may also benefit from *EPA*. Given a fault-tree for a mechanical system that includes software, how do you assess the probability of failure for the software node(s) of the fault-tree? Current practice employs two alternatives: (1) assume that the software always works, or (2) assume that the software always fails. Assuming (1) is clearly dangerous. Assuming (2) means that the resulting fault-tree cannot be mitigated. Our methodology provides a plausible but imperfect alternative to these extremes.

As a final example, we briefly examine a system software application (in contrast to the control software applications discussed above). Operating system testing has a set of inherent problems that stem from the myriad combinations of different peripheral devices and drivers that users can connect. The problem arises from the seemingly infinite (and almost all entirely unique) failure modes that such peripherals can supply to the OS. Almost everyone has experienced operating system failure on a personal computer. Such "crashes" and "hangs" often leave few options for correction: reboot or cycle power. The OS is almost always at fault in such cases, but it is often likely that some third-party or public-domain driver has contributed by sending unexpected signals to the OS that cause it to abort or hang.

One (impossible) solution to this problem would be for the OS manufacturer to test the OS on *all potential combinations of drivers and peripherals*. A more plausible solution would be to abstract away the huge number of drivers and peripheral combinations, and employ something like *EPA* to inject large classes of corruptions simulating failure-modes. This would provide a reasonable test of the OS regardless of the specific drivers and peripherals that are (or might someday) be connected. Such a solution provides immediate guidance as to how well the OS will perform when confronted with "randomized" failure modes, without having to purchase a boatload of peripherals. This is no guarantee that someday a device won't come along that fails in a way that *EPA* never considered. But this approach is far more comprehensive than merely acquiring some small subset of the available devices and then testing with only a few of the possible combinations; or, worse yet, simply assuming that peripherals will always fail in the same ways.

4 Conclusions

We have described an automated software environment that measures software failure tolerance using extended propagation analysis. *EPA* employs fault-injection to detect portions of the code that have demonstrated a capacity to output undesirable events. Results of *EPA* provide a developer with evidence concerning the robustness and tolerance of the code and give some idea of what will happen during what might be potentially harmful situations. *EPA* is also useful for optimum test resource allocation that can allow different input profiles as determined by an operational profile and its inverse.

Extended Propagation Analysis offers insights that set it apart from traditional methods of formal analysis or empirical software testing. The results of *EPA* are not proofs, nor are we testing for correct behavior. Instead, *EPA* identifies code locations that reduce robustness, and that probably lead to unsafe software behavior. On the flip side, if *EPA* measures acceptable levels of robustness in a piece of software, this suggests that the application has been sufficiently hardened against future anomalous events. This is by no means a *guarantee* of future robust behavior, but is instead a clear and objective *indication* of this behavior.

Failure-tolerance measurement with ultra-rare inputs is another tool that has the potential to yield significant benefits in future years. Because testing to the degree necessary to assess the highest levels of reliability is infeasible, a partial alternative is to demonstrate that the software will likely produce acceptable results when the software is operating with rare inputs. We have shown the utility of this method in two experiments on real systems.

More and more mechanical systems are being controlled by software. Since controller software will compute decisions based on the information it receives from imperfect parts, it is vital that those parts responsible for providing data be of the highest quality that is cost-effective. Building high-quality mechanical parts may be prohibitively expensive. One way to lessen this cost might be to employ redundant sets of cheaper parts and make the software more tolerant of such parts' potentially-flawed behavior. If the impact of inaccurate incoming information to the software can be shown to be benign, then the system designer has some evidence that the tolerances and imperfections of a particular component are acceptable. Thus, although this article on *EPA* focuses on software, its benefits could extend to encompass many aspects of system design and cost/benefit analysis.

Dijsktra's famous adage that *testing cannot reveal the absence of faults* is applicable to our *EPA* techniques as well. If while using our technique, $PRED$ is not satisfied, that does not mean that $PRED$ can never be satisfied by the software running in the future. If, on the other hand, $PRED$ *is* satisfied, it is up to the user of the systems to mitigate the potential of that sort of event occurring in the future. In other words, our technique can only warn of future problems whose causes can be currently revealed by the software — problems that may even include hardware failure. These problems, once discovered, cry out for investigation. To further complicate matters, it is possible that a particular simulated infection that does ultimately result in a $PRED$-satisfying output cannot in principle really occur during operation. In this case, mitigation is automatic. However, if a $PRED$-satisfying output can be observed while using a *plausible* simulated infection, it is prudent to invoke appropriate failure-tolerance improvement procedures that can thwart the propagation of such data-state corruptions. Such steps include: (1) building higher quality external hardware, (2) reducing the possibility of human factor errors, (3) making use of redundant hardware, and (4) inserting into the code software assertions that are capable of warning about and potentially correcting state corruptions. *EPA* has proven its worth as a tool that can reveal weak points (even in "good" software) and guide such corrective actions.

Acknowledgements

This work was partially funded by the Advanced Research Projects Agency under Contract F30602-95-C-0282, the United States Air Force under Contract F30602-95-C-0158, and NASA under Contract NAS1-

20207. The authors would like to thank Dr. John Knight of the University of Virginia for providing the code for the Magnetic Stereotaxis System and Carrie Walker of Nasa-Langley for providing the avionics code. We appreciate the useful comments provided by Jeffery Payne of RST and Larry James of Hughes Information Technology Systems on earlier versions.

References

[1] M. A. Friedman. BART Advanced Automated Train Control System Safety Concepts. In *Proceedings of American Public Transit Association Convention*, New York, May 1995.

[2] J. J. Marciniak. *Encyclopedia of Software Engineering.* Wiley, 1994.

[3] M. Elder and J. Knight. Specifying user interfaces for safety-critical medical systems. In *Proc. of the 2nd Annual International Symposium on Medical Robotics and Computer Assisted Surgery*, pages 148–155, November 1995.

[4] D. J. Lawson. Failure Mode, Effect, and Criticality Analysis. In J. K. Skwirzynski, editor, *Electronic Systems Effectiveness and Life Cycle Costing*, pages 55–74, NATO ASI Series, F3, Springer-Verlag, Heidelberg, 1983.

[5] J. Voas, C. C. Michael and K. Miller. Using Fault-Injection to Assess Software Engineering Standards. In *Proc. of Pacific Northwest Software Quality Conference*, pages 318–336, Portland, September 1995.

[6] J. Voas, G. McGraw, A. Ghosh, F. Charron, and K. Miller. Defining an adaptive software security metric from a dynamic software failure tolerance measure. In *Proc. of Ninth Annual Conference on Computer Assurance*, National Institute of Standards and Technology, Gaithersburg, MD, June 1996.

[7] J. Voas and K. Miller. Dynamic testability analysis for assessing fault tolerance. *High Integrity Systems Journal*, 1(2):171–178, 1994.

[8] J. Voas and K. Miller. Software Testability: The New Verification. *IEEE Software*, 12(3):17–28, May 1995.

[9] H. D. Mills. On the Statistical Validation of Computer Programs. *IBM Federal Systems Division, Report FSC-72-6015, Gaithersburg, MD.*

[10] J. D. Musa. Operational Profiles in Software Reliability Engineering. *IEEE Software*, 10(2), March 1993.

[11] Nasa. NASA Software Safety Standard. Office of Safety and Mission Assurance, June 1994. Interim Report 1740.13.

[12] K. N. King and A. J. Offutt. A Fortran Language System for Mutation-based Software Testing. *Software–Practice and Experience*, 27(7), July 1991.

[13] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.

[14] Underwriters Laboratory Inc. Safety Related Software, January 1994. Standard for Safety UL1998, First Edition.

[15] M. Friedman and J. Voas. *Software Assessment: Reliability, Safety, Testability.* John Wiley and Sons, New York, 1995. ISBN 0471-01009-X.

[16] P. Yam. Magnet on the Brain. *Scientific American*, August 1996, Vol. 275, No. 2, pg. 32.