

Analyzing Software Sensitivity to Human Error

Jeffrey Voas

Reliable Software Technologies Corporation

`jmvoas@rstcorp.com`

Abstract

Human operator errors in human-supervised computer systems is becoming a greater concern. Software fault injection is an inexpensive way to simulate thousands of human operator error scenarios to determine what will occur if they were to happen. By trying these different scenarios before a system is deployed, greater confidence can be achieved that the software and human will work amicably.

1 Introduction

Humans and machines are continually becoming more interdependent. Machines perform functions that either the human cannot or functions that the machine can perform more inexpensively or reliably.

Determining how well the interactions between humans and machines will go during operational usage at system design time is a speculative science at best. For example, how should the task load be allocated between the machine and human [7]? How can tasks be allocated to both parties such that maximum safety is achieved? How can psychological needs of the human be incorporated into the design? These types of questions cannot be fully answered using static approaches. Static approaches ignore the dynamics that play a key role in determining whether an event will lead to disaster or not (and in real-time). Only during “real life” simulations and actual operational usage can correct answers be found.

Humans make bad decisions when they feel stressed or overloaded with tasks. But even perfect task allocation strategies will only be beneficial towards avoiding certain types of errors caused from human overload or under load. There are other types of scenarios that involve either human or machine error that are not related to stress or work overload that enable disastrous consequences (e.g., the human making a wrong decision during a time where the workload was normal). Erroneous human/machine interactions such as these that are based on poor judgment or incorrect timing are the focus of this paper.

Human operator supervision is simply one form of input to computer software as shown in Figure 1. It is no different than when a software program reads in information from

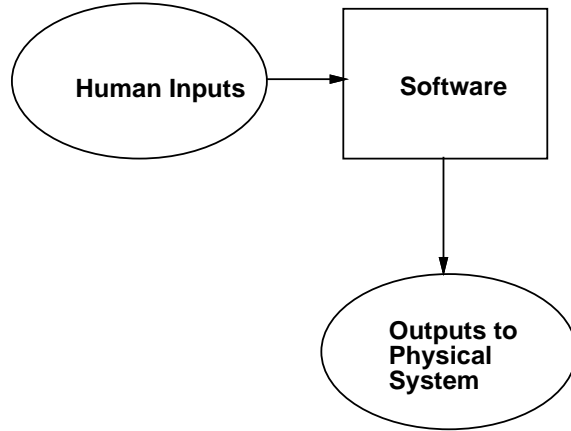


Figure 1: Basic Information Flow

a file or database. The machine accepts the user’s commands and then performs various computations (given the state of the software). The results from these computations control actuators that force state changes in a physical system.

Software is inherently safe by itself. Safety problems occur when the software forces the system that it controls to be put into an undesirable/unsafe state. This is usually a physical system, however software can also be unsafe if it returns defective information back to a human who then in turn makes improper decisions.

For example, suppose that software fails to recognize that pressure has been gradually increasing over time and hence fails to warn the operator. If the operator does relieve the pressure, an explosion could occur.

Ways to ensure safety in human-supervised systems include extensive human training, system designs that account for reasonable workloads and cognitive and psychological limitations (as well as human error), and thorough system and software design analysis. Our paper focuses on software design analysis.

Our approach uses *software fault injection*. Software fault injection is a “late life-cycle” software analysis that can simulate human operator errors and observe their impact on the software as well as the *total* system (including the human(s), all software and hardware, and the controlled mechanical parts). This provides a way to predict how badly things might get in the future before human-supervised systems are deployed. The benefit of doing this is that the results from this analysis can be pumped back into the human training process or system re-design when it is deemed that things could get unsafe if the human operator errs.

In summary, it is vital that computers under human supervision that perform safety-critical tasks have low probabilities that human error will result in unsafe states. For software systems such as games, safety is not a requirement, but for systems such as aircraft, telemedicine, and plant-control systems, safety must be assured.

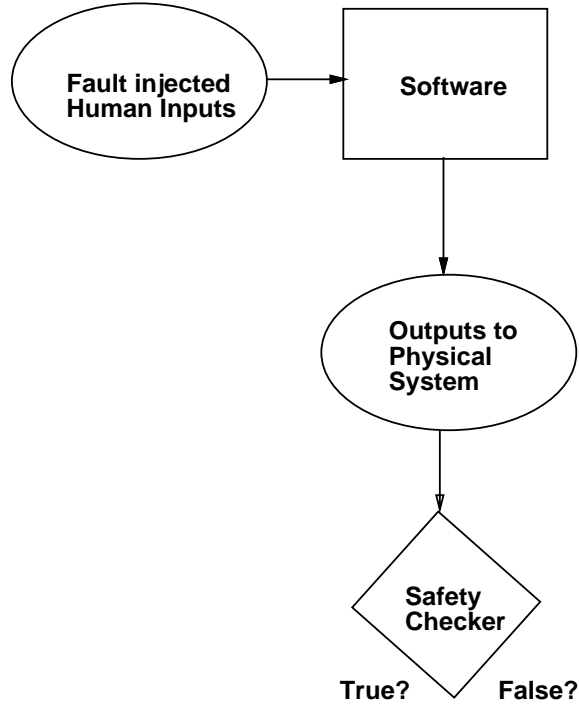


Figure 2: Simulated Human Operator Error Process

2 Fault injection

Our approach will be to exploit software fault injection analyses to simulate human operator errors. Software fault injection is a “what if” analysis technique that can forcefully corrupt the information in a program’s state (as well as mutate the code) [5]. Because inputs to a software system are part of its state as soon as they are read in, inputs too then can be corrupted (either right before they are read in or immediately thereafter). We have already published ways to use fault injection to simulate erroneous inputs going into COTS-based systems [9]. The approach we will use here is a twist on that.

The process we will use is shown in Figure 2. Here, we see human inputs that have been corrupted by a fault injector being fed into the software. Next, an *assertion checker* tests to see if those corrupted inputs violate the safety constraints of the physical system. The assertion checker has definitions for all system-level hazards defined during creation of the system safety requirements. If during this process the inputs do violate those defined hazards, then from a safety standpoint precautions must be taken to ensure that the human not make the same or possibly similar mistakes. This assumes, of course, that the simulated, corrupted human errors injected by the fault injector are considered as plausible.

Software fault injection can be a complicated process. Many different parameters must be decided before the analysis is run. The selected values for these parameters do impact the results of this analysis. We will briefly run through a review of the basics behind the concept of fault injection (some refer to this process as *fault insertion*) [4].

In terms of modifying internal software states, the key question surrounding the fault injection process is: *do forced modifications (anomalies) of a program's state to a state that the program (or its external environment) did not naturally create provide any insights into how badly the program might behave in the future?* This question is usually referred to as the “reality” issue. Or are artificially manufactured data state anomalies totally void of real-world implications? If the latter is true, fault injection is almost certainly useless. If the former is true, then the goal is to derive as much insight from the results as possible.

Before performing fault injection, there are two unique classes of parameters that must be defined by the user. First, there are those parameters that the user wants forcefully injected into the state of the software as it executes. These are termed the *data anomalies*.

The second set of parameters define classes of behavior that the user of fault injection does not want the software to exhibit. These could be events such as hazardous output states (that are defined during hazard analysis) or calls to system-level utilities that the application software should not be making. This second class of parameters are generally referred to as *output anomalies*. Output anomalies must be defined with respect to the state of the physical system that the software and human control. (In Figure 2, the box labeled “Safety Checker” is testing for whether output anomalies have occurred.)

The reality issue is particularly important when fault injection uses a pseudo-random number generator to create data anomalies. For example, modifying the value held in some variable from something like 100 to 1,000 may or may not be a reasonable data anomaly to make. Alterations such as this are common when generation of the altered value is based on the original value. The fault injection methods that we generally use for simulating programmer faults and faulty input data employ pseudo-random number generation. And there will be cases where pseudo-random number generation is a reasonable approach for simulating human operator error, but this will be more rare.

If there is no foreseeable way in which a particular data anomaly could occur during the execution of the software in its natural environment (other than by force using fault injection), then the benefit of performing fault injection may seem negligible. But interestingly enough, even though a data anomaly may not be representative to naturally occurring data anomalies, it could still forewarn us of output anomalies that will result from the “more realistic” data anomalies. Thus even poor choices of artificial data anomalies can potentially reveal undesirable output behaviors that were not known to be possible prior to fault injection being applied. In a sense then, this shows how fault injection can act in a similar capacity to an Failure Modes and Effect Analysis (FMEA).

Software faults, design errors, faulty input data, and human operator error can all force internal data anomalies to creep into the state of the application software at run-time. To inject realistic artificial data anomalies, consideration must be given as to which *root cause* was assumed to have originated the data anomaly. Because some root causes are easier to instantiate (and then simulate) than others, it is easier to argue that their artificial data anomalies are indeed realistic.

Fortunately, we are in luck here, because the two easiest root cause classes to simulate

correctly (meaning simulate in a way that does reflect the types of real errors that would occur naturally) are: (1) *faulty input data*, and (2) human operator error (which is just a type of faulty input data). Software faults and design errors are tougher to justify, because the space of possible software faults and design errors is typically infinite. Thus knowing which faults or design errors are representative of all other possible faults or design errors (in order that their data anomaly effects can be injected) is not generally possible. (This problem is one of the key reasons why *mutation testing* [8] never caught on.) The space of different ways that external data sources or human operators can err is generally more tractable (however it still can be a very large space).

For example, if the software reads digital data from an A/D converter (that gets its data from an analog sensor), we can exploit information concerning the expected sensor failure modes and apply those failure modes to the data going into the A/D convertor before it comes into the software. Or if we fear that the A/D convertor might fail, we can simulate that as well. By doing so, we can see how the software reacts to sensor and A/D convertor failures.

Likewise, for systems that employ human operator control, there is likely to be a fixed number of choices that a human can decide between given a specific time-critical situation. Some of these choices will result in acceptable system-level outputs. Others will result in unsafe output anomalies.

Human operator choices can be modeled using customized data anomalies. These anomalies can be injected while the software repeatedly executes in a fixed operating scenario. From there, observations can be taken as to which human choices resulted in acceptable software outputs (with respect to the complete system—physical and software subsystems) and which ones did not.

What really makes the space of human input events explode is when time is factored in. That is, when did the human give the command? Here, time is defined according to the clock of the physical system. For real-time systems, time will always be a consideration when simulating human errors. Time adds this new dimension, because now, not only must we consider the human making the wrong selection independent of time, we must now account for the additional errors of: (1) the human giving the right command but at the wrong time, and (2) the human giving the wrong command at the wrong time. This increases our space of options greatly during fault injection.

As an example, suppose that a human is trained to press switch A within 5 seconds of hearing a warning alarm. It might be of interest to see what happens if A is pressed at times outside of the 5 second period. It might also be of interest to see what occurs if an incorrect switch is pressed during times before and after the 5 second threshold.

These and other classes of human operator error can easily have their outcomes tested using data anomalies that are created by fault injection methods. Further, fault injection methods can be used to simulate a different class of human operator error: *out of sequence inputs*. For example, suppose that an operator is trained to give the software a fixed sequence of commands when a system reaches a particular state, but instead, the operator inadver-

tently modifies this sequence. Whether this modification has hazardous consequences may be unknown. If it can be shown that this modification does not have hazardous consequences, then we have preliminary evidence that this operator mistake can be tolerated by the system, and in fact, calls into question whether this mistake is really a human error at all.

3 Delayed Human Inputs and Input Sequence Errors

Fault injection is a process that either instruments source code or instruments the interfaces to executable objects. Instrumentation is simply code that is added to the existing code to either: (1) gather data about what is occurring internally at that point during an execution, or (2) change the software's state at the point where the instrumentation is added. Our human error simulation uses instrumentation that changes the software's state. In the following examples, we will show how instrumentation is added using pseudo-code.

We are now ready to discuss ways that human operator errors can be simulated. Here, for brevity, we will only consider two classes: (1) inputs that were given at the wrong time, and (2) inputs that were provided to the software in the wrong sequence. Timing and sequencing problems are often the most curious human operator errors and therefore we will concentrate on those, however others of interest include: (1) invalid abort or cancel sequences, (2) incorrect input format, and (3) incorrect keyboard input.

Simulating input delays in software is a fairly straightforward process. Suppose that the software is expected to sit a wait loop until the user inputs a command. The way to handle this using software fault injection is to trap the user's command and hold onto it for a fixed period of time before then allowing the system to process the input.

For example, if the code waits for a user input,

```
read(user_input_string);
```

the fault injection instrumentor can simply add in a delay beyond that caused by the operator with a command such as:

```
read(user_input_string);
loop 10000 times
    do nothing
endloop;
```

or

```
read(user_input_string);
delay(1000);
```

Depending on the language of the source code, these are the types of schemes employed by our fault injection tool.

This instrumentation holds onto the user’s input for a fixed period of time before allowing the software to perform computations with the input data. Either approach has the same effect as that of delaying the input as if the user had caused the delay into the software. Simulating human operator speed-ups (where the person inputs the information before they are supposed to) is harder than delays but possible none-the-less [1].

Incorrect input sequence errors are slightly harder to simulate via software fault injection but are possible nonetheless. One way to implement them is to order each input signal from the user into an array, shuffle the elements in the array reordering them (or a subset of the elements), and then read the elements back out from the first to last element. This provides the effect of the user supplying the input information in the wrong order. This can also be combined with the time delays just discussed, where not only are the inputs then fed into the software in a shuffled order, but as elements are read in, some members are read in in a delayed manner.

4 Failure Combinations

As already mentioned, fault injection has long been used to simulate errors classes associated with programmer and software design errors in isolation [6]. Fault injection has also been successfully applied to simulating the failure of hardware systems in isolation [2]. The problem becomes much more challenging to prevent coincidental failures.

Multiple coincidental failures of subsystems are difficult to predict. Herb Hecht [3] reported that multiple failures and exception conditions are a prominent cause of program failure in well-tested programs. This is intuitive, because of the rarity of coincidental failures and the low likelihood that one will occur during system-level testing.

Multiple coincidental failures are the types that designers fear most in any safety-critical system. Worse yet, if you cannot imagine what multiple coincidental failure modes are plausible, then predicting the system-wide impact of them is impossible.

Generally speaking, the most traditional approach to limiting damage from parallel subsystem failures is to partition subsystem failures such that the failure of one subsystem cannot “team up” with the failure of another subsystem to result in a greater system-wide loss. Multiple recovery approaches that perform this are described in [4].

Consider further the added concern about these classes of subsystem failures occurring at the same time as when human operator errors are occurring. Clearly, when a system is running smoothly, there are fewer chances for the operator to start making additional errors than if the system is in an erroneous state and the human panics. The possibility of this additional class of error occurring at the same time that non-human errors have occurred results in an intractable number of possible combinations to consider and protect against during design and testing. In fact, even for fault injection methods, this space of possible multi-dimensional events is too large to adequately protect against.

While it is true that all combinations cannot be tested, samples from the space of pos-

sibilities should be employed. Although it might only be possible to execute the system while a handful of these events are firing, it is still prudent to test a system using events that include simultaneous hardware failure, software failure, and human operator error. The reason for this is clear: this is the type of a system scenario for which human judgment *must* be correct. There is no room for error. Therefore it is reasonable to test the effect of human error at those times when other non-human subsystems are experiencing failures.

5 What To Do Next

The question that arises after this analysis is performed is “now what?” When classes of human operator inputs are observed that cause problems to the physical systems, then either: (1) the human operator needs to be trained to ensure that the operators do not perform in a manner consistent with the fault injection tool, or (2) the software needs to be hardened to refuse to accept those classes of inputs that were created by the fault injection tool. Either way, an opportunity is afforded designers to improve the safety of the physical system based on incompatibilities between the human and software.

The first alternative here is more reasonable after it has been determined that the errors simulated by the fault injector were plausible errors that the operator must not make. It makes no sense to train personnel to not do things that they would have never done in the first place. And it does not make sense to start modifying code to protect against human errors that are not plausible. The second alternative is also plausible, but making changes to the software will almost certainly cost more than changes in personnel training.

6 Summary

This paper has discussed how traditional fault injection techniques can be modified with reasonable effort to simulate the following two types of human operator errors for real-time control systems: (1) input timing mistakes, and (2) incorrect inputs or input sequences (with respect to the state of the complete system). Clearly, fault injection cannot be expected to test using every possible human operator error, but it can go a long way towards isolating those classes of human error that must be avoided.

It is fair to think of this process as a “sensitivity” analysis for how sensitive the software and physical system are to the operator. The more sensitive the software and physical system are, the less room for error there is by the human.

Fault injection can simulate many more combinations of human errors than can be tried manually. Fault injection is highly automatable (although not completely), and therefore, fault injection’s costs are mostly computer time once the user of this approach customizes their fault injection tool to the classes of human operator errors that make the most sense for their particular physical system.

There are many other forms of human operator errors that can be simulated that were not discussed. For example, a pilot simply putting in the wrong coordinates for where a plane is to fly, and the plane then going in the wrong direction. That clearly is human operator error, but that error does not need to be simulated in order to reveal to us that it will cause the plane will go in the wrong direction. So therefore it is prudent to only simulate human errors for which the outcome is unknown. Simulating human errors for which the outcome is already known is foolish except in those cases where validation is desired.

Human errors that are related to *feedback* have not yet been addressed in this approach. For example, suppose that an initial operator error causes no safety problems, but that error causes the operator to make a series of subsequent errors that cascade into an unsafe situation. In this situation, the fact that a successor error was a function of its predecessor error is what leads to an unsafe state. To date, our human error simulation cannot account for this form of correlated corrupt state buildup. Our fault injection tool injects errors at single points in time, and not errors that are dependent on previous errors. We plan to someday add the necessary intelligence to our tool to do so. But this is very difficult to do via existing fault injection technologies.

The results from this analysis can reveal whether different human operator scenarios put a software-controlled system into a hazardous mode. This analysis can also be combined with the more traditional fault injection engines that simulate classes of data anomalies (such as incorrect sensor information or incorrect data being read from a database) in order to test the tolerance of the software to operator errors in an already hostile environment. This is prudent since the interactions of coincidental failures are hard to predict.

Acknowledgments

Jeffrey Voas has been partially supported by DARPA Contracts F30602-95-C-0282 and F30602-97-C-0322, National Institute of Standards and Technology Advanced Technology Program Cooperative Agreement Number 70NANB5H1160, US Army Contract DAAL01-98-C-0014, and NASA-Ames Contract NAS2-98052. THE OPINIONS AND VIEWPOINTS PRESENTED ARE THE AUTHOR'S PERSONAL ONES AND THEY DO NOT NECESSARILY REFLECT THOSE OF THESE AGENCIES.

References

- [1] RELIABLE SOFTWARE TECHNOLOGIES CORPORATION. Quantifying Confidence in the Correctness of Parallel/Distributed Software. Technical report, Sterling, Virginia, USA, July 1993. Final Report for Contract NAS1-19896.
- [2] J. VOAS, F. CHARRON, G. MCGRAW, K. MILLER, AND M. FRIEDMAN. Predicting How Badly 'Good' Software can Behave. *IEEE Software*, 14(4):73–83, July 1997.

- [3] H. HECHT. Rare Conditions: An Important Cause of Failures. In *Proc. of the Eighth Annual Conference on Computer Assurance*, pages 81–85, National Institute of Standards, June 1993.
- [4] M. LYU, editor. *Software Fault Tolerance*. Wiley, 1995.
- [5] J. VOAS AND G. MCGRAW. *Software Fault Injection: Inoculating Programs Against Errors*. John Wiley and Sons, New York, 1998.
- [6] J. VOAS AND K. MILLER. Software Testability: The New Verification. *IEEE Software*, 12(3):17–28, May 1995.
- [7] M. SUJAN AND A. PASQUINI. Allocating tasks between humans and machines in complex systems. In *Proceedings of the 4th Int'l. Conf. on Achieving Quality in Software*, pages 173–184, April 1998.
- [8] 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.
- [9] J. VOAS. Certifying Off-The-Shelf Software Components. *IEEE Computer*, June 1998.