A Tutorial on Software Fault Injection

J. Voas (jmvoas@rstcorp.com)
Reliable Software Technologies, Sterling VA USA

1 Introduction

Software failures are receiving increased attention because of their unpredictability and ubiquity. It is becoming commonplace to read news stories mentioning a recent disaster caused by a software problem. Take, for instance, the three recent Tritan rocket launch failures that cost over $3B dollars. In Congressional testimony, the US Air Force admitted that one failure was caused by a decimal point in the source code that should not have been there [5]. This is simply the most recent in a series of rockets failures that began back in 1994 with the Ariane 5 disaster. That disaster was directly traced to a software reuse error [2].

Physical failures occur as a result of manufacturing defects or physical fatigue and decay. In contrast, software failure occurs as a result of: (1) defects (or faults) in the logic of the code, (2) incorrect data being fed into the software from its environment, or (3) some combination of the two. While there are numerous reasons why code can be logically incorrect, the key reason is that either the program was mis-coded or there was an error in the requirements/specification/design of the system. The key reasons why incorrect data gets fed into software are either human operator error or the failure of some external hardware or software system that is responsible for feeding input to the software.

Logical defects in software are not always problematic, however. In an IBM study from 1984, Ed Adams showed that often the faults that are the most prevalent in the code actually cause infrequent failures. If those failures are not severe, then the defects that are responsible may not be worth fixing. A good rule of thumb to remember is that the frequency and severity of a type of failure are key factors as to whether the resources should be spent to fix the incorrect logic.

This is a catch-22, however. To have information on: (1) the severity of a defect, and (2) the frequency of failure caused by the defect, we must also have information on *where* and *what* the defect is. Testing and other types of analysis can help in isolating "easy to find" faults. But testing, unless exhaustive, can never guarantee to have isolated all faults. Why? Because of the possibility that any untried input could cause software failure because of an unknown fault.

This catch-22 is a generally unsolvable problem. We hope that the defects that will lead to serious failures will be debugged prior to code release. But we are likely to not know about their existence. This makes debugging them unlikely.

This problem has led our research group to advocate a change in mind set: instead of spending resources to debug defects, spend resources to reduce the impact of any residual defects by first determining how badly the software can behave if it is defective and then

adding the appropriate level of fault-tolerance [1]. This champions going after the *effects* of bugs instead of the bugs. The question is "how?" After all, we will still need information about where and what the defects are in order to accurately assess their effect.

Also, there is an interesting legal dilemma here relating to knowing where defects exist. By law, there are two kinds of software defects: patent defects and latent defects [7]. *Patent* defects are defects that the software publisher knows about yet chooses not to fix prior to releasing the software. *Latent* defects are those that the publisher does not know about prior to software release. Interestingly, there is no law in the United States that requires a publisher to notify licensees about patent defects. If patent defects turn out to frequently cause unacceptable failures, the publisher is more likely to be sued for negligent behavior. Note that by thwarting the effects of defects as opposed to removing defects, the likelihood of being sued still decreases since the negative impact of both patent and latent defects is reduced. This is simply the process of adding fault tolerance to a system (but, of course, without employing the standard techniques such as redundancy).

Since we cannot directly solve the problem of knowing where all of the defects are, one option for boosting software's fault-tolerance is to: (1) hypothesize the existence of defects, and (2) then determine the impact of the hypothesized errors (anomalies). From there, we can speculate on how actual defects will cause the software to behave. This provides clues as to how robust the software is in its current state. After all, if a program is intolerant to artificial defects, it is likely that the software will be intolerant to real defects. And if that intolerance can result in hazardous or malicious behavior, the pre-deployment phase is the time to take action to prevent the software from doing so.

2 Understanding Software Fault Injection

The approach that we will employ that hypothesizes the existence of defects is termed "software fault injection." Software fault injection's goal is to "trip up" the software as it executes in order to see how the software reacts. Software fault injection is similar to software testing, but different. It provides similar information to formal verification but is not an absolute guarantor of certain behaviors as formal verification is. Instead, fault injection is empirical.

Software fault injection is similar, in principle, to the hardware fault injection techniques that have been used for years. (A nice overview of these techniques occurs in [4].) The key difference is the target of fault injection and the types of anomalies that are injected. For example, hardware fault injection has traditionally flipped the bits in the pins on the processor, changed the power supply to the chip, and bombed the chips with heavy ion radiation. More recent hardware fault injection tools are now being applied to simulations of processors during the chip design stage. These tools can be applied long before a chip is ever produced.

Carrieras *et al.* [4] discuss three categories of hardware faults: permanent, transient, and intermittent. Permanent faults are the result of irreversible damage to the hardware component. Transient faults are triggered by environmental conditions, and intermittent faults are caused by unstable hardware. And as Carrieras *et al.* point out, while software faults are permanent, the behaviors are usually transient, although for some software faults, they can cause permanent (meaning never ending) failures. Our goal here will be to discuss

the types of software faults and other external anomalies that can be employed to test the robustness of a piece of software to a variety of permanent and transient anomalies.

Unlike applying fault injection early in the design phase to simulations of those hardware chips, software fault injection is applied later in the life-cycle. Software fault injection should be applied before the software is released but closer to the time when the software is about to be deployed. It is premature to apply software fault injection to a system that is experiencing large amounts of modification. Once the code is stable, the code is ready for it.

Software fault injection produces precise results. For example, when simulated error A is injected into a program executing test case B, and the program's behavior changes from C to D, that is an precise piece of information. Software fault injection is not a perfect technology, however. Imperfection occurs as a result of the problem that practitioners face when interpreting the results.

For example, the practitioner must now consider whether the occurrence of D means that the software can naturally produce D for a different test case than B without being artificially stimulated to do so? Does A represent an anomaly that is at all reasonable with respect to B or any other test case? It is questions like these that ultimately must be addressed before the results from this process are used to modify the software to increase its fault tolerance.

Therefore, to justify the costs of fault injection, the following hypothesis needs to hold.

> Accurate information about how a program will behave under *real*, anomalous circumstances can be obtained from injecting *artificial* anomalies into the program.

If this hypothesis rarely or never holds, fault injection is both a waste of resources and could even be harmful. Fortunately, however, this hypothesis often holds even though there is no clear theoretical understanding among researchers as to why it does.

3 Methods of Performing Fault Injection

The hypothesized errors that software fault injection uses are created by either: (1) adding code to the code under analysis, (2) changing the code that is there, or (3) deleting code from the code under analysis. One key requirement from these processes, however, is that the code that is either added, modified, or deleted must change either the software's output or an internal program state for at least one software test case. (Different applications of software fault injection will guide the decisions as to which of these two alternatives applies.) Without this requirement, the hypothesized errors will have had no semantic impact to the original code base and thus were meaningless (they were not anomalies at all). In mutation testing (a type of fault injection that we will discuss later), this is the dreaded "equivalent mutant" problem. The difficulty stems from the fact that equivalent mutants are often undetectable, forcing the costs to perform mutation testing to be much greater than they should be [3].

Figure 1 shows the software fault injection process. Code that is added to the program for the purpose of either simulating errors or detecting the effects of those errors is called *instrumentation code*. To perform fault injection, some amount of instrumentation is always necessary, and althrough this can be added manually, it is usually performed by a tool.

Instrumentation code can be placed on top of input or output interfaces to the software or directly into the logic of the software.

Instrumentation can be added into a variety of code formats: source code, assembly code, binary object code, etc. In short, any code format that can be compiled, interpreted, or that is ready for execution can be instrumented.

There are two key approaches for simulating errors: (1) directly changing the code that exists (this is referred to as code *mutation*), or (2) modifying the internal state of the program as it executes. We will now walk through an example of each approach beginning with code mutation.

Suppose a program has the following code statement:

```
a = a + 1;
```

This statement could be mutated as follows:

```
a = a + a + 1;
```

(provided that `a` does not have the value of zero). We could also modify the statement to:

```
a = a + 10;
```

And we could delete the statement as well. Note that all of these mutations change the resulting value of `a` from what it would have had not we not mutated the code (and for every test case that allows this statement to be executed).

The concept of forcefully changing the internal state of an executing program is a slight variation on the code mutation examples just shown. Clearly, each of the mutations above will change the state of the program after they are executed. But note that that is not necessarily true for all mutants. There are code mutants that although they are exercised will not modify the software's internal state. That would be the case if the value of `a` before the mutant a = a + a + 1 was executed is zero. (This would be an example of a transient fault using the definitions provided by Carriera *et al.*)

To forcefully modify a program's internal state to a value different than the one it currently has, we will add a function call to the code that overwrites the current internal value of a portion of the program's state. Typically, we overwrite programmer defined variables or the data that is being passed to or from function calls. By modifying this data, we are simulating the internal effects of faulty logic or any other anomalous event that could possibly affect the software's internal state.

The function calls we add to overwrite internal program values are termed *perturbation functions*. Perturbation functions are code instrumentation. When perturbation functions are applied to programmer defined variables, they typically either: (1) change the value of the variable to a value based on the current value, or (2) they pick a new value at random (independent of the original value). Also, they can simply return a constant replacement value if it is suspected that any fault placed at that point in the code would likely result in one particular value regardless of what the current value was. When non-constant replacement values are used, the perturbation function will produce random values based on the current value and a *perturbing distribution*. Non-constant perturbing distributions include all of the continuous and discrete random distributions. The perturbation function

`newvalue(x)= equilikely( floor(oldvalue(x)*0.6), floor(oldvalue(x)*1.40))` is an example of a discrete distribution that perturbs a value by substituting an equilikely random value on the interval of 40% more and 40% less than the expected value. This function however leaves the possibility of returning `newvalue(x) = oldvalue(x)`. Conditions are placed in the code that executes this function to avoid this.

For example, if we wanted to change `a`'s value to something close to what it has after this computation,

```
a = a + 1;
```

we would replace the original statement with the following code chunk:

```
a = a + 1;
a = newvalue(a).
```

The code for `newvalue()` would also be added somewhere into the program and would look like the following pseudo-code:

```
int newvalue(int a)
{
  counter = 1;
  oldvalue = a;

  do
  {
     a = equilikely( floor(oldvalue * 0.6), floor(oldvalue * 1.4) );
     counter++;
  }
  while ( (a == oldvalue) && (counter < 100) );

  if ( (counter == 100) && (a == oldvalue) )
  {
    a = oldvalue - 1;
  }

  return (a);
}
```

(Note that 0.6 and 1.4 can be modified to however tight or loose of an interval as is desired. For example, 0.0001 and 10000 could be used to widen the interval of choices.)

Because this function could result in an infinite loop while trying to find a different value, a counter is added to ensure that after 100 attempts, the loop terminates and simply decreases the value of `a` by one. (We could have just as easily decided to program it to increase the value by one or even flip a coin as to which it does.)

Note that we can also use fault injection to modify the time at which code is executed by adding function calls that slow down the software. For example, in Ada, we can add a `delay(5)` statement to stop a process from executing for 5 milliseconds. And we can even

simulate events such as the software's state, stored in memory, having its bits toggled due to radiation or other electromagnetic corruption. The `flipBit` function which will now be described provides this capability.

flipBit

The perturbation function `flipBit` toggles specific bits. The first argument to `flipBit` is the original integer value and the second argument is the bit to be toggled (we assume little-endian notation). The function `flipBit` is then written in C as follows and linked with the executable. Note that the `^` represents the `XOR` operation in C and the `<<` operator represents a `SHIFT-LEFT` of `y` positions.

```
        void flipBit(int *var, int y)
        {
           *var = *var ^ (1 << y));
        }
```

`flipBit` can serve as the underlying engine from which other perturbation function can be created. For example, to toggle two or more randomly selected bits in the integer, we can employ `flipNbits`:

```
void flipNbits(int *var, int n)
{
        int bits = 0;
        int bitPos = 1;
        int i,j,k;
        int xbit;
        for (i = 0; i < n; i++)
        {
           bits |= bitPos;
           bitPos <<= 1;
        }
        for (j = 0; j < sizeof(int) * 8; j++)
        {
           xbit = lrand48()
           if ((!!(bits & (1 << xbit))) !=
               (!!(bits & (1 << j))))
           {
              flipBit(&bits, xbit);
              flipBit(&bits, j);
           }
        }
        for (k = 0; k < sizeof(int) * 8; k++)
           if (bits & (1 << k))
               flipBit(var, k);
}
```

4 Applications and Conclusions

Fault injection is an indirect means for trying to predict how events might unfold in the future. How well-behaved a software system might be or how badly behaved it might be are serious concerns to anyone that relies on software for safety-critical or business-critical tasks.

In this article, we only discussed simulating single, isolated errors. Note that fault injection can easily simulate multiple, distributed errors throughout the code. For example, we can corrupt the value of variable `a` at line X in the code while also corrupting the value of variable `b` at line Y in the software and so on and so forth. In short, we can make as complex of distributed faults as we desire, but recogize that this problem becomes in intractable quickly. We can rarely exhaustively simulate all possible single faults, let alone all combinations of distributed faults.

This overview has focused on: (1) the rationale behind software fault injection and the hypothesis it is based on, (2) basic methods for performing software fault injection, and the application of fault injection to fault-tolerance assessment and improvement. Note that there are other applications of software fault injection. For example, when you go back to some of the earliest discussions about software fault injection in 1978 [3], you see mention of the type of fault injection termed *mutation testing*; mutation testing's goal is to generate a good set of test cases.

Later in the early 1990s, I proposed using a combination of code mutation and state mutation to assess the likelihood that a program would hide errors during testing [8]. This was simply a twist on using code mutation to develop better suites of test cases. Later, I proposed using fault injection as a means for assessing both the security and safety of systems [9]. And in 1999, we showed how fault injection could be used to test the completeness of the system-level hazard analysis done very early on in the software life-cycle [11] as well as information warfare and survivability [6] and human operator error simulation [10].

In short, there are many ways to implement software fault injection algorithms. How it is implemented will be based on the information that the user needs. Software fault injection should be viewed as an assessment technique that does not replace any other technique, cannot be replaced by any other technique, must be applied carefully, and whose results must be interpreted correctly in order for its benefit to manifest.

5 For Further Information

For further education on software fault injection, I recommend:

1. J. Voas and G. McGraw. Software Fault Injection: Innoculating Prgrams Against Errors. John Wiley and Sons, 1998.

2. A. Ghosh and J. Voas. Innoculating Programs for Survivability, Communications of the ACM. 42(7):38-44, July, 1999.

3. Carreira *et al.* Fault-injection spot checks computer system reliability. IEEE Spectrum, August, 1999.

4. R. DeMillo, R. Lipton, F. G. Sayward, Hints on test data selection: Help for the practicing programmer 4(11):34-41, IEEE Computer, April, 1978.

5. Dozens of papers on the subject are available at: `http://www.rstcorp.com/papers`

6. "Developing Software for Safety Critical Systems" June, 1998, IEEE Reliability Society and IEEE Educational Activities, NTSC ISBN 0-7803-4573-8. (To get a copy contact IEEE at 1-800-678-IEEE)

7. "Software Testing: Building Infrastructure, Due diligence, and OO Software" May, 1999, IEEE Reliability Society and IEEE Educational Activities, NTSC ISBN 0-7803-5312-9. (To get a copy contact IEEE at 1-800-678-IEEE)

References

[1] 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.

[2] Prof. J. L. Lions. Ariane 5 flight 501 failure: Report of the inquiry board. Paris, July 19, 1996, available at `http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html`.

[3] 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.

[4] J. Carreira, D. Costa, and J. Silva. Fault-injection spot checks computer system reliability. *IEEE Spectrum*, August 1999.

[5] S. Sobek. "Human error called culprit in 3 rocket launch failures", Florida Today Space Online, June 16, 1999 (www.flatoday.com/space/today/061699e.htm).

[6] A. Ghosh and J. Voas. Innoculating Software for Survivability. *Communications of the ACM*, 42(7):38–44, July 1999.

[7] C. Vossler and J. Voas. *Advances in Computers*, chapter Defective Software: An Overview of Legal Remedies and Technical Measures Available to Consumers. Academic Press, to appear in 2000.

[8] J. Voas. *A Dynamic Failure Model for Performing Propagation and Infection Analysis on Computer Programs.* PhD thesis, College of William and Mary in Virginia, March 1990.

[9] J. Voas. Testing Software for Characteristics Other than Correctness: Safety, Failure tolerance, and Security. In *Proc. of 13th Int'l. Conf. on Testing Computer Software*, Washington, D.C., June 1996.

[10] J. Voas. Analyzing Software Sensitivity to Human Error. *Journal of Failure and Lessons Learned in Information Technology Management*, 2:201–206, 1998.

[11] J. Voas. Hazard Mining. In *Proc. of the 2nd IEEE Workshop on Application-Specific Software Engineering and Technology (ASSET'99)*, pages 180–184, Dallas, TX, March 1999.