Fault Injection for the Masses

Jeffrey Voas
Reliable Software Technologies

ABSTRACT *Many people believe that software fault injection is a complex process that requires investing thousands of dollars in tools to perform. Here, we look at the most basic requirements for performing fault injection and how, for limited software analysis, the instrumentation needed to collect the results can be inserted manually, thus avoiding the need for advanced tools.*

Getting theoretical ideas into practice is a key reason I avoided academia. The key technology that I would like to see adopted by "the masses" is a family of software fault injection algorithms that can predict: (1) where to concentrate testing (as well as how much testing to do), (2) whether systems can tolerate anomalies and remain in safe states, and (3) whether systems can be broken into via malicious events. (There are other lesser applications that I will ignore here for brevity.) From a novelty standpoint, these algorithms were (and still are) unique amongst other methods of performing fault injection. I concede that the algorithms are computational, but the results can provide unequaled information about how "bad things" propagate through systems. Because of that, I've considered the methods valuable to anyone responsible for software quality. This spans the spectrum from one-person Independent Software Vendors (ISVs) to the largest corporations.

Moving this technology from its early adopters (which have been large corporations and US government organizations) to the masses (which is mainly comprised of the smaller ISVs) is no small challenge [1]. Even with testimonials such as:

> *"Based on the successful use of fault injection on prior and on-going programs, we are moving toward standardizing on this technique for all future software-intensive safety critical systems."*
>
> Larry James, Senior Scientist, Engineering & Technology Center, Hughes Information Systems
>
> *[Fault injection is] "the next wave in the rapidly advancing field of software quality assurance"*
>
> Barry Preppernau, Test Training Manager, Microsoft
>
> *"This is a fundamental technology that has no equal. The principles and specific methodologies have been greatly beneficial in increasing the efficiency and effectiveness of our software testing, especially in the area of high-reliability, high-availability systems."*

getting the masses to try out fault injection requires a user to take resources from one V&V pot and place them in another. As a general rule-of-thumb, people become uncomfortable when asked to move from something familiar to something unfamiliar. To most people, fault injection is still a great unknown.

During the remainder of this piece, I'd like to share my thoughts on how the small, resource tight organizations can begin making fault injection a "software reality" in their processes. In fact, you can start seeing results within hours by following the simple steps I will now walk through.

Many people think that to perform fault injection that they must have an expensive automated tool. Rubbish! Early on, I did fault injection by hand. (It was not until several years later that I had access to a tool that automated portions of the analysis.) Then, I coded in the results collecting and injecting instrumentation manually. This clearly limited how much fault injection could occur, but it also had a side benefit. It forced me to think about where I wanted the "injectors" placed in the code. By thinking more about placement, I needed less instrumentation, making manual analysis more feasible. And it also decreased computational costs. Now I'm not suggesting that this is a long-term solution that voids the need for automation, but it does allow you to try out fault injection without acquiring tools.

You may be wondering how I decided where to place the fault injection probes (as well as how many places in the code I could realistically perform this manual process). The most that I recall doing was around 25-30, mainly because the process is so repetitive that I would start making dumb mistakes (like forgetting to delete previous fault injection instrumentation before embedding the new instrumentation). What I would do would be to search through the code for those places where specific variables (that I feared might have wrong information in them) were assigned values. By applying fault injection to those places, my fears were either confirmed or dispelled.

As an example, I almost always opted to place fault injectors right after input information was read in. I was naturally curious about how my program would behave after bad information was fed into it, and fault injection provided a reasonable way to satisfy this curiosity. Such curiosity is justified and is felt by most software developers. They know that assuming that information originating from external sources will be correct is dangerous. The question then becomes "how dangerous?"

To make this manual process a little easier for the reader to visualize, let's walk through a couple of examples. Let's assume you have a sequence of statements in your code like this:

```
x = (r1 - 2) + (s2 - s1)
y = z - 1
  :
  :
  :
T = x/y
```

You might wonder what affect an incorrect value for `x` has on `T`. For example, suppose it is catastrophic when `T` > 100. (Here, I'll assume a very basic test for what it means for

an output to be catastrophic.) Since `x` affects `T`, this can be explored further by manually replacing the above code with:

```
x = (r1 - 2) + (s2 - s1)
x = perturb(x)
y = z - 1
  :
  :
  :
T = x/y
if T > 100 then print ('WARNING')
```

To deal with `perturb(x)`, you will also need to embed the following extra code into your software:

```
x = uniform(a*x, b*x)
return (x)
```

Here, `a` and `b` are user supplied, and `uniform` is a call to a random number generator that will generate random numbers in the interval $[ax, bx]$. (Numerical algorithms for generating uniform random values can be found in most simulation texts. Also, you may have access to software to perform this random number generation via standard math library functions.) Once you have made these simple modifications to your code, all you need to do is run your modified code repeatedly and see how often the warning is printed. If never, you have evidence that `T` is insensitive to `x`.

As another example, suppose you are interested in what happens when `y` is corrupted. To test this out, change the original block of code to:

```
x = (r1 - 2) + (s2 - s1)
y = z - 1
y = perturb(y)
  :
  :
  :
o = x/y
if o > 100 then print ('WARNING')
```

as well as adding in the `perturb(x)` code. When this is compiled and executed, this will tell you how sensitive `T` is to a corrupted value in `y`. The point here is that with these minor modifications, you're doing software fault injection! You're not using any tool, just a little elbow grease.

If from this analysis you discover that a bad `x` or `y` cannot be tolerated (because 'WARNING' was printed), then steps need to be taken that ensure that `x` and `y` cannot be incorrect. To begin, ensure that the equations assigning values to `x` and `y` are correct. Next, ensure that `r1`, `r2`, `s1`, `s2`, and `z` are correctly computed (as well as the parameters that they depend on). Once this is done, *confidence* is provided that `T` > 100 is impossible. Unfortunately, however, you will not have a *guarantee* of this.

In this example, I assumed access to the source-code under scrutiny. If you are the developer of the code, this assumption is realistic. But if you are the consumer of someone else's code, it may not be. Fortunately, fault injection can be performed without source-code if you are willing to treat blocks of code as black-boxes. In [3], my co-authors and I present a method for performing fault injection on COTS and third party software. This type of fault injection (called "Interface Propagation Analysis") assumes that the software is in a black-box. By calling `perturb()` on returned information from a black-box, interface propagation analysis forces corrupt information into a system as it executes and observes the effect of doing so. This technique allows the user to study how a failure of one software component can affect its neighbors and its neighbors' neighbors.

For brevity, this article ignored many important issues, including: (1) how to determine `a` and `b`, (2) how to corrupt non-numeric data, (3) how many times to perform fault injection, and (4) how to decide which outputs are really bad. For help with these questions, you can read [2] or contact me directly.

In summary, fault injection is a basic technology that does not require specialized hardware or software. My hope is that this simple exercise has convinced you to start playing with fault injection. Even without a tool, meaningful results can be found with simple data corruptions like the one I have shown.

Admittedly, to do this analylsis for thousands of lines of code, an automated solution will be required. But until you reach that point, simply insert your checks and fault injectors manually. Then sit back and watch fault injection unlock the inner workings of your code and show you what is really going on in there!

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] J. Voas and G. McGraw. *Software Fault Injection: Inoculating Programs Against Errors.* John Wiley and Sons, New York, 1997.

[3] J. Voas, G. McGraw, A. Ghosh, and K. Miller. Glueing together Software Components: How good is your glue? In *Proc. of Pacific Northwest Software Quality Conference*, pages 338–349, Portland, October 1996.