

# Can Chaotic Methods Actually Improve Software Quality Predictions?

J. Voas

## 1 Introduction

Predictability is simply a measure of how accurately a future event can be forecast. 100% predictability is synonymous with omnipotence. 0% predictability is synonymous with no clue whatsoever.

We have many different ways to define what it means for software to be of “good” quality. When we say that a particular package is of high quality, what we mean is that the software *will* behave in a manner that is defined as acceptable.

Note the key verb here: will. Such an assertion generally translates into expectations that in the future, the software will be available, fail rarely, and operate with acceptable performance. And the key word here is generally, and unfortunately that is not a strong enough assertion for systems for which failure cannot be tolerated. General statements about quality do not provide precise confidence about how the software will behave in the future.

Before we begin discussing our ability to predict software quality, consider analog devices. They generally provide good levels of predictability about future behavior. For example, lightbulbs are sold with MTTF scores. A car traveling at 40 mph will still be traveling at a speed +/- some small amount from 40 a second later assuming there are no obstacles. Even analog systems such as the weather are quite predictable if we only look as far as the immediate future. We can predict the weather one minute from now with much greater accuracy than we can predict the weather 5 days from now.

Software system predictability is quite different from that of analog systems and devices, regardless of how immediate into the future we make the prediction for. Flipping one bit one nanosecond later, and all bets are off as to what will the consequences will be.

Our inability to *precisely* predict how a software program will behave in the future (as opposed to *generally* predict) is caused by two interacting culprits: (1) hiding defects, and (2) enormously large input spaces. If the size of input space were more tractable and if all existing faults could be identified, the predictability problem would decrease substantially.

To have 100% software quality predictability, we must know the *consequences* of each fault. That requires the impossible task of knowing where each fault is and how each fault interacts with each software input. Nonetheless, since 100% predictability is impossible, the questions are “how close to it can we come and how?”

## 2 Faults and Input Spaces

We begin by looking at the problem caused by incorrectly coded logic. *Faults* (or *defects* as they are often called) are manifestations of a mental mistake made by programmers and system designers. (Mental mistakes are typically referred to as *errors*).

Clearly, if we knew where every fault was in a program, we could reduce unpredictability by: (1) fixing them, or (2) determining their behavioral consequences. Option (1) would be preferable if all faults could be successfully fixed without new ones being introduced. But let’s look at the magnitude of the problem of doing so.

For simplicity, let’s assume there is only one defect in a 1M SLOC program. And assume we also know that the defect is isolated to an unknown single line.

Suppose that we randomly select a line, inspect it, and find that it is correct. We continue doing so until 10K lines have been inspected yet the fault does not get detected. Inspecting 10K lines offered a 1% probability of finding the fault yet we failed to find it (assuming we inspected correctly).

Let’s next look at the difficulty of option (2) for the same 1M SLOC program: determining the behavioral consequence of all faults (which in this case is just one). Assume the program reads in 4 32-bit integers and assume that there are 1M unique inputs on which the program will fail. Randomly selecting one input during testing provides a probability of finding one consequence of the fault of  $\frac{1,000,000}{2^{128}}$ . Thus it is unlikely that testing will ever determine the consequence of this fault for one input let alone for all 1M inputs.

This brings us to an interesting point. While most people consider software complexity to be caused by large amounts of syntax, there is another factor that makes software programs complicated: “seemingly infinite” input spaces. As an example, consider again the legal input space containing input vectors with 4 32-bit, legal integers. The potential number of input combinations is  $2^{128}$ . Executing the program on each one is infeasible.

Consider next that there are two types of input spaces: legal and anomalous. Legal simply refers to the set of input vectors that the software is expected to execute on according to the input domain defined in the specification. Anomalous refers to the set of input vectors that corrupt the software’s state regardless of whether a fault was executed. Anomalous inputs can come from a variety of sources: human operator error, hardware failures, and other software systems (*e.g.*, databases, standard libraries, and operating system utilities). As an

example, consider a software program whose legal input space is five, 2-character buffers, where each buffer has two members that are the same alphabetic character. Here, ((aa), (bb), (cc), (dd), (ee)) would be a legal input while ((aa), (bb), (cc), (dd), (ec)) would be anomalous.

To achieve 100% predictability for a software package, not only must testing consider all legal inputs, but it must also consider all anomalous inputs since their manifestation during operation could flush out software behaviors never observed during testing using legal inputs. Although the anomalous space may be smaller than the legal input space, it is likely to be very poorly defined, thus nullifying access to it during testing.

Thus we really have interacting culprits: (1) unknown defects, (2) unknown anomalous inputs, and (3) intractable numbers of legal inputs. But all hope is not lost. Operational reliability testing based on the operational profile, while unable to guarantee 100% predictability due to the vast number of legal inputs, can provide predictability that is proportional to the number of legal inputs tried.

But operational reliability testing can do nothing for predicting how the software will behave with anomalous inputs unless the specification defines them along with the appropriate output behaviors. Recognize that it is hard enough to define what we want the software to do for the legal inputs let alone having to also do so for anomalous inputs. Thus this is unlikely.

As an analogy, consider a deck of cards. The precise outcome from shuffling a deck of cards is somewhat unpredictable, but the probability of any specific deck occurring can still be written down as a uniform probability density function. Why? Because we know what exists in a standard card deck: 4 suites with 13 cards each. But assume that you did not know this (*e.g.*, maybe the deck has 51 copies of the same card with one card that is unique). Then a probability density function cannot be ascribed. This latter case represents the problem caused by the anomalous input space.

And recognize how often anomalous problems that were not considered during design wound up contributing the high consequence failures.

**Titan IV launch failure** “...their focus was on controlling areas where previous problems had occurred. Since the roll rate filter constant error had not occurred before, the process was deemed low risk.” (*Aviation Week and Space Technology*, 2 Aug 99, p. 31)

**eBay outage** “We’ve designed for no single point of failure, but this particular device [a network switch thought to be in parallel and independent of the other switches] found a weakness.” “Our outages never tend to be [caused by] the same thing, which is incredibly disconcerting.” – (Maynard Webb of eBay, *Information Week*, 6 Sept 99, p. 48)

**Mars Polar Lander** “The sensors were designed to tell the vehicle when it had landed so it could shut down its engines. Instead it appears that because of faulty software, the sensors told the spacecraft it had landed when it was still 130 feet from the Martian surface. That information automatically shut down the engines before they had slowed the vehicle’s speed; the lander probably crash-landed on Mars at 50 miles an hour.” – (Shoddy testing led to Mars probe failure, *The Chicago Tribune*, March 29, 2000)

Thus in my opinion, possibly the greatest problem plaguing software quality predictions is high consequence failures caused by anomalous inputs or extremely infrequent legal inputs that were overlooked during design. Even though reliability testing provides predictability for high probability legal inputs, it does nothing for low probability legal inputs or undefined anomalous inputs of unknown probability.

Note that there are techniques to deal with low probability legal inputs. For example, if operational profiles exist, they can be inverted and then sampled from. This again provides predictability that is proportional to the amount of testing done.

But the anomalous space cannot be as easily dealt with. Since it cannot be defined, it cannot be directly sampled from using standard, systematic, techniques. So what can be done?

One approach involves hypothesizing what the anomalous space might look like. By forcing the software to experience artificial and arbitrary anomalous inputs using a technique termed *software fault injection* that does so, it is possible to observe *new* software output behaviors that were not known to be possible. Learning about unknown output behaviors provides additional predictability concerning future potential software behaviors.

While the process of throwing arbitrarily created corrupt input data at a program might appear *ad hoc* (and hence the results from doing so useless), the nonsystematic nature of this process works because of what we are dealing with: nonsystematic and arbitrary anomalies that will affect the software in the future. (Here, “nonsystematic” suggests that there is no relationship between any two of these anomalous inputs and therefore there is no probability distribution that can be ascribed to the likelihood of them occurring.)

Software fault injection’s *ad hoc* nature for arbitrarily flipping bits to corrupt data is well suited for this problem. In essence, it matches a chaotic problem with a chaotic solution. And while there are plausible arguments as to why such an approach should only result in nonsense, it turns out that instead, this process often flushes out software behaviors that baffle software developers and designers by revealing potential future problems that require protection against. The designers and developers simply would have never predicted the software could do “that.”

### 3 Summary

In the natural world, there is a theory termed “highly optimized tolerance (HOT)” that explains why ecosystems and particular species of plants and animals tolerate rare and anomalous events. The theory simply points out that HOT systems are capable of withstanding all of the anomalies that they were designed to withstand, but if other anomalies occur, disaster will be the result. For example, a rainfall increase of 10% in a season is unlikely to cause the duck population to become extinct. Why? Because ducks are designed to withstand such. But if a micro-organism or new virus strain were to attack the population, it could be wiped out. The same theory applies to software.

Clearly, designing software to handle all possible anomalous events is infeasible. Even designing software to behave appropriately for all legal inputs will generally be infeasible. Therefore we have discussed ways to increase predictability for how a software system will behave under a variety of input scenarios.

Software suffers from unpredictability as a result of our inability to exercise even small percentages of the legal input space. For this reason, different input space partitioning schemes such as equivalence class partitioning have been proposed over the years to allow at least one input from each partition to be tried.

Even so, such testing fails to account for undefined anomalous inputs. These inputs cannot be mapped to a probability distribution. They occur infrequently and how and why they occur is unknown *a priori*. Nonetheless, they may cause the most serious consequences because the software will not have been designed to tolerate them.

By matching a technique based on arbitrary randomness to the problem of arbitrary randomness, predictability about future software output behaviors has the potential to be increased. While not a perfect solution, it offers an approach for increasing predictability when faced with unknown future events that cannot be modeled probabilistically because the events cannot be defined.