Predictability is a measure of how accurately we can forecast a future event. 100% predictability is synonymous with omnipotence, 0% with having no clue whatsoever. Analog devices generally provide good levels of predictability. For example, manufacturers sell light bulbs with MTTF (mean time to failure) scores. A car traveling at 40 mph will still be traveling at a speed plus or minus some small amount from 40 mph a second later, assuming no obstacles exist. Even analog systems such as the weather are quite predictable if we only look at the immediate future.

Software system predictability, however, is quite different from that of analog systems and devices. When we say a software package is high-quality, we mean that it will behave in a manner we’ve defined as acceptable. That is, we generally expect it to be available, operate as it’s supposed to, and rarely fail. Unfortunately, the key word here—“generally”—is not a strong enough assertion for systems that can’t tolerate failure. General statements about quality don’t provide confidence about how software will behave, no matter how immediately into the future we predict.

Software system predictability, however, is quite different from that of analog systems and devices. When we say a software package is high-quality, we mean that it will behave in a manner we’ve defined as acceptable. That is, we generally expect it to be available, operate as it’s supposed to, and rarely fail. Unfortunately, the key word here—“generally”—is not a strong enough assertion for systems that can’t tolerate failure. General statements about quality don’t provide confidence about how software will behave, no matter how immediately into the future we predict. Flip one bit one nanosecond later, and all bets are off on what the consequences will be.

Two interacting culprits cause our inability to precisely (as opposed to generally) predict a software program’s behavior: hidden defects (faults) and enormous input spaces. If the input-space size were more tractable and we could identify all existing faults, the predictability problem would decrease substantially.

To have 100% software quality predictability, we must know each fault’s consequences. This requires the impossible task of knowing each fault’s location and how each fault interacts with each software input. Because 100% predictability is impossible, the question becomes, “How close to it can we come and how?”

Faults

First, look at the problem caused by incorrectly coded logic. Faults are manifestations of mental mistakes (usually called errors) by programmers and system designers.

If we knew where all the faults were in a program, we could reduce unpredictability by fixing them or determining their behavioral consequences. Fixing them would be preferable if we could fix all faults without new ones arising. But let’s look at the magnitude of the problem of doing so.

For simplicity, let’s assume only one defect exists in a program comprising one million source lines of code. And assume we know the defect is isolated to an unknown single line. Suppose we randomly select a line, inspect it, and find that it is correct. We continue doing so until we have inspected 10,000 lines, yet we don’t detect the fault. Inspecting 10,000 lines offers 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 the second option for the same program: determining all faults’ (in this case, just one) behav-
ioral consequences. Assume that the same program reads in four 32-bit integers and that 1,000 unique inputs on which the program will fail exist. Randomly selecting one input during testing provides a probability of 1,000,000/2^{128} for finding one of the fault’s consequences. So, it is unlikely that testing will ever determine this fault’s consequence for one input, let alone for all 1,000 inputs.

Input Spaces

Although most people consider large amounts of syntax to cause software complexity, another factor complicates software programs: seemingly infinite input spaces. For example, consider again the legal input space containing input vectors with four 32-bit integers. (Legal refers to the set of input vectors on which the software is expected to execute according to the input domain the specification defines.) The potential input-combination number is 2^{128}. Executing the program on each one is infeasible.

Next, consider anomalous input spaces. Anomalous refers to the set of input vectors that corrupt the software’s state, regardless of whether it executed a fault. Anomalous inputs can come from various sources: human-operator error, hardware failures, and other software systems (for example, databases, standard libraries, and operating system utilities). Consider a software program whose legal input space is five two-character buffers, where each buffer has two members that are the same alphabetic character. Here, ((aa), (bb), (cc), (dd), (ee)) would be legal and ((aa), (bb), (cc), (dd), (ec)) would be anomalous.

To achieve 100% predictability, testing must consider not only all legal inputs but also all anomalous ones, for their manifestation during operation could flush out software behaviors never observed during testing using legal inputs. Although the anomalous space might be smaller than the legal input space, the system designers will likely have defined it poorly, thus nullifying access to it during testing.

So, we really have three interacting culprits: unknown defects, unknown anomalous inputs, and intractable numbers of legal inputs. All hope is not lost, however. Operational-reliability testing based on the operational profile, while unable to guarantee 100% predictability because of the vast number of legal inputs, can provide predictability proportional to the number of legal inputs tried. But such testing can’t help predict how the software will behave with anomalous inputs, unless the specification defines them along with the appropriate output behaviors. However, defining what we want the software to do for the legal inputs is hard enough, without having to do so for anomalous inputs.

Consider a deck of cards. The precise outcome of shuffling a deck is somewhat unpredictable, but we can still write the probability of the occurrence of any specific card as a uniform probability density function, because we know what exists in a standard deck: four suits with 13 cards each. But if we didn’t know this (for example, maybe the deck has 51 of the same card and one unique card), we couldn’t ascribe a probability density function. This latter case represents the problem the anomalous input space causes.

Software Fault Injection

Possibly the greatest problem plaguing software quality predictions is high-consequence failures caused by anomalous inputs or extremely infrequent legal inputs that de-
signers overlooked. (For some recent examples, see the “Consequences of Software Quality Unpredictability” sidebar.) Even though reliability testing provides predictability for high-probability legal inputs, it does nothing for low-probability ones or undefined anomalous inputs of unknown probability.

Techniques exist for dealing with low-probability legal inputs. For example, if operational profiles exist, we can invert and sample from them. This again provides predictability proportional to the amount of testing done. But we can’t deal with the anomalous space as easily—because we can’t define it, we can’t directly sample from it using standard, systematic techniques.

One approach to this problem involves hypothesizing how the anomalous space might look. By using software fault injection to force the software to experience artificial and arbitrary anomalous inputs, we can observe new software output behaviors that the developers and requirements engineers didn’t know to be possible. Learning about unknown output behaviors provides additional predictability concerning future potential software behaviors.

Although the process of throwing arbitrarily created corrupt input data at a program might appear ad hoc (and hence the results from doing so useless), this process’s nonsystematic nature works because we are dealing with nonsystemic and arbitrary anomalies that will affect the software in the future. (Here, nonsystematic suggests no relationship exists between any two of these anomalous inputs and therefore we cannot ascribe a probability distribution to their likelihood of occurrence.)

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. Although plausible arguments exist as to why such an approach should only result in nonsense, we find that, instead, this process often flushes out software behaviors that baffle software developers and designers, by revealing potential future problems.

In the natural world, the highly optimized tolerance theory explains why ecosystems and particular plant and animal species tolerate rare and anomalous events. This theory states that although HOT systems can withstand all the anomalies they were designed to withstand, disaster will result if other anomalies occur. For example, a 10% rainfall increase in a season is unlikely to cause a duck population to become extinct, because nature designs ducks to withstand such anomalies. But if a microorganism or new virus strain were to attack the population, it could wipe the ducks out. The same theory applies to software.

Because software suffers from unpredictability—a result of our inability to exercise even small percentages of the legal input space—over the years researchers have proposed different input-space partitioning schemes, such as equivalence class partitioning, to let testers try at least one input from each partition.

Even so, such testing fails to account for undefined anomalous inputs. We cannot map these inputs to a probability distribution. They occur infrequently, and how and why they occur is unknown a priori. Nonetheless, they might cause the most serious consequences, for the software won’t have been designed to tolerate them.

By matching an arbitrary-randomness technique to the arbitrary-randomness problem, we could increase predictability about future software output behaviors. While not perfect, this approach lets us increase predictability when faced with unknown future events that we can’t model probabilistically because of our inability to define them.

Consequences of Software Quality Unpredictability

The following are recent examples of anomalies that designers either overlooked or incorrectly predicted to not be likely. They demonstrate the difficulty in predicting how systems will behave when designers either don’t know all circumstances or incorrectly assign them low probabilities of occurrence.

- **Titan IV Launch Failure:** “Their focus was on controlling areas where previous problems had occurred. Because the roll rate filter constant error had not occurred before, the process was deemed low risk.” (Aviation Week and Space Technology, 2 Aug. 1999, p. 31.)

- **eBay Outage:** “We’re 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.” (Mike Wilson of eBay, Information Week, 6 Sept. 1999, 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, 29 Mar. 2000.)