A "Crystal Ball" for Software Liability

J. Voas, G. McGraw, L. Kassab, and L. Voas

Abstract

Software fault injection is an emerging technology that can be used to observe how software systems behave under experimentally-controlled, anomalous circumstances. In so doing, software fault injection acts as a "crystal ball," predicting how badly software might behave should things go awry (both internally and externally) during execution. Such predictions provide clues as to how robust a piece of code is, where in the code robustness is deficient, and most importantly, what level of liability is incurred by relying on a particular software system. Once we find places in the code that are intolerant to the occurrence of anomalous events, we have found a source of liability within the software. Fault injection is an efficient, scientific means for predicting how events might unfold in the future.

Keywords

fault injection, anomaly, safety, failure class, fault class, liability, risk

1 The Liability Problem

The current Information Revolution will arguably have as strong an effect on society as the Industrial Revolution did in the early 1900's. But today's revolution carries with it challenging new problems (*e.g.*, how do you ensure that the new currency of information is not lost, incorrectly computed, or maliciously mangled?). If the correctness and authenticity of information cannot be guaranteed, information consumers will be left vulnerable, and information system developers may find themselves liable. For these reasons, the prospect of seemingly unbounded liability hangs over the heads of today's software developers and suggests that we are entering a new age where haphazard development and assurance philosophies will be met with severe penalties. So many people have become dependent on software that it is only a matter of time before lawsuits over information and its quality become commonplace.

For most software practitioners, 1986 was probably not a particularly noteworthy year in the software industry. However 1986 was the year of the first court case ever to result in a software development company being found guilty of software engineering malpractice [4]. It is not unreasonable to foresee the day when even public domain software (provided freely without compensation or warranty) will result in litigation against its developers. Legal precedence has been set according to the Hill-Burton Act that allows patients to sue medical personnel that volunteer their services

in free health clinics.[1] It is unwise to assume that developers of *freeware* will always be immune to this type of unwanted attention, even when explicit disclaimers are included. Of course, most attention will probably be focused on companies with deep pockets that are developing and selling code for a profit.

Software liability issues are characterized by two key questions: (1) who is liable when a piece of software fails, and (2) what is the probability that a liable event will occur (in other words, what is the *risk* associated with using a piece of software)? The obvious answer to the first question is the developing organization. The answer to the second question is more subjective and is the focus of this paper.

Questions of liability are legal and ethical quagmires that in the case of software also happen to introduce technical questions. We argue for employing fault injection — dynamic methods that simulate *the effects* that real faults will most likely have (as opposed to simulating the faults themselves) — as a means of quantifying the "risks" created by the software component of a system. If we can accurately and scientifically measure these risks, we will have a means for probabilistically bounding liability.

If software never produces undesirable outputs, then questions of liability should not arise (however given today's litigious atmosphere even this is not guaranteed). But determining *a priori* that software will never fail is, in general, impossible. In order to quantify the risks of software failure, we need to know the *probability* that the software will fail in a manner that is consistent with a resulting loss. After all, not all failures translate into losses.

In this paper, we consider a *software failure* to have occurred when the software does not perform as specified such that the aberrant behavior causes financial loss or injury. This definition of failure is slightly different than that defined by the IEEE since it adds the clause about loss. One important caveat, there is nothing that makes perfectly correct code immune to legal liability challenge. In fact, software, whether it is correct or not, may be especially susceptible to *phantom risks* wherein "legal liability [is] claimed and occasionally found for poorly supported but popularly endorsed allegations concerning cause and effect" [10, page 9].

Generally speaking, risk can be divided into two types: *speculative* and *pure*. Speculative risk can result either in a loss or a profit (*e.g.*, investing in company stock may pay off or may not). With pure risk, the only potential consequence is loss (*e.g.*, automobile collision insurance only pays off if an accident occurs). Risk is not problematic if it is either too small or if affordable insurance can be obtained to offset the potential loss. Insurance gives companies that produce goods and services an opportunity to order their business affairs so that a certain cost (a *premium*) is substituted for an uncertain potential cost (specifically, a disastrous loss of unacceptable magnitude). The ability to acquire insurance provides some degree of security, and thus continued economic activity in an environment containing risk. Failure to substitute risk through the payment of premiums means retention of the potential adverse consequences. Few prudent businesses are willing to assume such a risk.

Software risk assessment is still a "black art." Thus far, insurance companies have purposely opted not to insure software as a stand-alone entity apart from the entire information system. Before insurance companies (assumers of risk) can offer the security that comes with insurance, they must

[1] Two such interesting legal cases are: (1) American Hospital Association v. Schweiker, 721 F.2d 170, and (2) Flagstaff Medical Center, Inc. v. Sullivan, 962 F.2d 879

quantify their risk. Risk quantification evaluates uncertainties and places a dollar amount on the risk to be assumed. *Perils* are the cause of pure risk. Examples of perils include floods, fire, and disease. Underwriters also look beyond specific perils to the causes of such perils (*hazards*) which precipitate the resulting loss. Examples of hazards include bad weather patterns for the peril of flood, poor electrical wiring for the peril of fire, and poor health habits for the peril of disease. Thus the providers of insurance need tools to measure the risk and assess hazards so that they can mathematically calculate premiums. Without such tools, the insurance provider cannot prudently assume risk and accurately set premiums.

The early writers of marine insurance estimated the risks of ship and cargo loss by taking into account weather patterns of specific routes at specific times, patterns of known pirate activity, and the success and failures of key crew members (such as the captain) as well as the particular vessel. Underwriter accuracy in assigning a premium/risk ratio is enhanced if a large data base of previous shipping experiences is available. For example, suppose that an underwriter assigns risk numbers from 1 to 10 (smallest risk to greatest risk) on each of the assumed equal categories of the perils: weather, pirates, previous crew/ship experiences. Next assume that a winter voyage calls for a weather rating of 9, pirate activity calls for a 2, and the crew/ship have an excellent rating of 1. In this case, 12 is the combined risk out of a possible total of 30. A premium of 12/30 of the insured value would be assessed (as well as some value for underwriter profit). The relatively high premium for this proposed trip could be justified if the shipper anticipates inflated profits due to a winter delivery. However, the shipper might decide to sail later when the weather peril decreases. Another option is for the shipper to sail without protection and retain all risk. The concept of seeking data to turn *total* uncertainty into *partial* certainty is the main impetus of both those who seek and those who provide insurance.

Insurance companies can "play the percentages" when they have extensive prior experience that is directly relevant to a certain risk. Prior experience allows for prediction with reasonably high confidence. In the software domain, however, extensive prior experience is often not available because each piece of software is one-of-a-kind, and a reasonable amount of prior experience with a particular piece of software will almost certainly not be built up until the life-span of the software is essentially complete.

We are now positioned to probe deeper into our second question: what is the probability that a liable event will occur? If we can find ways to observe how bad the future *might* get by looking at the types of output the software produces, then we can search for bounds on the degree of liability being assumed from a fixed software system before it is deployed.

2 State-of-the-practice: software risk assessment as alchemy

In terms of the definitions above, software is a *pure* risk. Software is expected to work correctly even though the reality is that failure occurs often in practice. But, as we mentioned above, even correct software can carry risks. This problem is compounded by the fact that correctness is defined with respect to the requirements/specification. If the requirements or specification are not correct (as is too often the case), it is foolish to expect that the software will perform properly in all circumstances. When a problem related to correct code from incorrect requirements/specification occurs, the issue of who is liable becomes even muddier. This is because it is the *customer* who

uses the software, not the *developer*, that is often tasked with writing the requirements.

Software presents a relatively novel form of risk — particularly software that is capable of triggering an event that causes loss-of-life. Each software system is unique, hence generalizing risk assessments from previous projects will not necessarily be beneficial, although they might. This is the case even if factors such as application domain and personnel are very similar. A good example here is the Ariane 5 rocket disaster. The code that caused the disaster on the Ariane 5 had not caused problems for the Ariane 4, and because the code had worked properly for the Ariane 4, it was assumed it would work well for the Ariane 5 [7].

It might appear that to assess risk, all we need to do is accurately quantify *reliability*. But 100% reliability does not necessarily mean 0% risk. Even if this were not the case, there is yet another problem with using reliability as a basis for liability prediction: the overabundance of conflicting software reliability prediction models. Because we can almost never know the true reliability of a piece of software, many models typically employ *error history* information to predict future failures by tracking prior failures. But different error-history–based reliability models often compute different reliability predictions for the same data, making it impossible to determine which model will be the most accurate for a specific system.

Before the current emphasis on software *development processes* came to the forefront, the most popular method of assessing software quality was by *testing*. Testing focused on the *final product* — the software system itself. The current popularity of process-oriented techniques such as formal methods is, in part, a reaction to the intractability of performing adequate software testing. Unfortunately, the relationship between development processes and the attainment of some desired degree of product quality is not well-established.

Testing software can aid in measuring some but not all classes of risk, though testing cannot measure risk to the desired precision. Process measurements are more tractable, but they measure the *wrong* thing: the process. Risk assessment demands accuracy. Today's state-of-the-practice in software risk assessment is typically *ad hoc*. For these and many other reasons, we say that the "science" of software risk assessment is analogous to alchemy.

3 Software Anomalies

3.1 Classifying Anomalies

To assess liability, we need to know how bad things will get in the future and how often these things will occur. Liability exists when the probability of disaster is non-zero. If this probability is zero, liability is voided.

Software liability directly stems from three classes of problems: (1) erroneous input data (from sensors, humans, or stored files), (2) faulty code, or (3) a combination of (1) and (2). To accurately assess liability, we need worst case predictions concerning software outputs if any of these types of problems were to occur in the future. Testing and other analytical techniques (such as static fault-tree analysis) are geared toward detecting the risks of defective code.[2] Fault injection will be used to determine a software phenomenon that we term "sensitivity" (or tolerance). Sensitivity is

[2]Note that there are other applications of fault injection that require defective code to be simulated (e.g., mutation testing [6]).

the reaction of the software to the injections, and can be used to assess the risks associated with (1) and (3). High sensitivity means that the software, after receiving injections, frequently output undesirable information (where undesirable is defined by either the specification, requirements, or definition of the software hazards). High sensitivity implies lesser failure tolerance which suggests greater liability. As we will discuss later, a sensitivity level is a probabilistic estimate (in [0.0, 1.0]) that can be used to quantify risk.

If software does not experience problems during execution, then it cannot behave badly. That is, only when software encounters problems that corrupt its program state can things go awry. An *anomalous event* (or anomaly) is a problematic event that occurs during software execution. During an anomalous event, the state of the software is altered. This alteration may or may not affect the output of the software. The number of different anomalies that software can experience during its lifetime is almost always infinite, and the range of the set is unknown. The anomalies that are of interest as candidates for fault injection are: (1) problems that can arise from code defects (caused by programming errors or design defects), (2) problems related to human factor errors, (3) problems with corrupt data being read in from stored files, and (4) problems caused by other external failures (*e.g.*, hardware or other software upon which the program depends for inputs).

As concerns over safety and potential liability increase, it is becoming common for professional standards to enumerate certain classes of problems that must never occur. The argument *for* doing this is that it is unreasonable to expect developers to protect against problems until they are told what problems to protect against. Standards usually define problems in one of two ways: (1) as a class of output failure that must be avoided, or (2) as fault classes that must be avoided if they have the potential to result in unacceptable output failures. Recent examples include NASA and Underwriter's Laboratory who have both released standards that define broad classes of anomalies software should tolerate and possibly correct. The NASA standard (NSS 174013) mentions the following anomalies:

> input/output timing, multiple events, out-of-sequence events, failure of events, adverse environments, deadlocking, wrong events, inappropriate magnitude, improper polarity, and hardware failure sensitivities, etc.

Underwriter's Laboratory Standard, UL1998, lists:

> b) Coding errors, including syntax, incorrect signs, endless loops, and the like; c) Timing errors that can cause program execution to occur prematurely or late; d) Induced errors caused by hardware failure; e) Latent errors that are not detectable until a given set of conditions occur; ...

The argument *against* enumerating classes of problems is that it is simply not possible to enumerate enough of the potential anomalies to justify the effort. Such an effect can be likened to trying to drain a lake with a teaspoon. To circumvent this problem, standards committees often (and purposefully) write vague standards. Even if the definitions within these standards were tightened, the act of showing that there are no circumstances in which the software could violate them is virtually impossible. On one hand, precisely spelling out all anomalous instances is infeasible. On the other hand, spelling out broad classes of problems leaves it to the developer to

interpret what they must protect against. But from a fault injection standpoint, these standards are ideal starting points from which to develop simulated anomalies or define what output types are undesirable.

3.2 Injecting "Hypothesized" Anomalies

The underlying ideas behind fault injection are not new, and there exists much literature describing how to employ them for hardware system validation, software testing, and hardware design validation [6, 9, 2, 1]. Unfortunately, the migration of those ideas into practical methods for software validation has not occurred, mainly due to concern about the plausibility of the anomalies injected. For example, in an integrated circuit, the failure classes are obvious: stuck-at-one, stuck-at-zero, etc. But for a software system, the internal data states are not as simple as just a '0' or '1', and hence the number of different data mutations that could be injected is intractable.

The key concern surrounding all software fault injection methods is that the information collected may not be relevant for the actual problems that the software will experience in the future, *i.e.*, the results of fault injection will not hold for the real problems that will eventually manifest. Further, the prototyping and commercialization of fault injection tools has been very limited (*e.g.*, MOTHRA [5], SafetyNet from RST, Fault Tolerance and Performance Evaluator from the University of Illinois, etc.). Without automated fault injection tools, the analysis is rarely feasible.

The process of injecting anomalies into a piece of software requires *instrumentation*, which is the process of adding code inside the software being analyzed, and then compiling and executing the modified (or *instrumented*) software.[3] Instrumentation can be accomplished in one of two manners: *intrusively* or *non-intrusively*. During intrusive instrumentation, code is added to the software being analyzed and then compiled in. Non-intrusive instrumentation involves executing extra code outside of the program being analyzed, with communication hooks into the program providing access to the state of the program so that its behavior can be observed. Because some communication must occur, which requires insertion of "hooks", the term "non-intrusive instrumentation" is almost always an oxymoron.[4] In any case, instrumentation generally injects its anomalies at one of two different levels of abstraction: *mutation* or *state perturbation*. The mutation approach changes the syntax of existing code statements. The state perturbation approach usually does not change existing code (except in very specific cases), but instead adds new code that changes the program states created by the original code.

We will now describe how the enormous anomaly space whose members can be simulated via fault injection can be partitioned. The goal is to employ fault injection like a fortune-teller would employ a "crystal ball" for the purpose of seeing how damaging a piece of software can be when the software experiences anomalous events. To do this, we must be confident that what is injected is appropriate. If the fault injection process can be successfully fine-tuned such that the results provided are clear and unambiguous, then it will be possible to predict *a priori* whether the future behavior of a program will be acceptable.

Software fault injection methods suffer from a similar intractability problem to the exhaustive testing problem:

[3] MOTHRA is an example of a tool that employs an interpreter to avoid the compilation step, but it is quite primitive [5].

[4] Some control control systems can be instrumented non-intrusively.

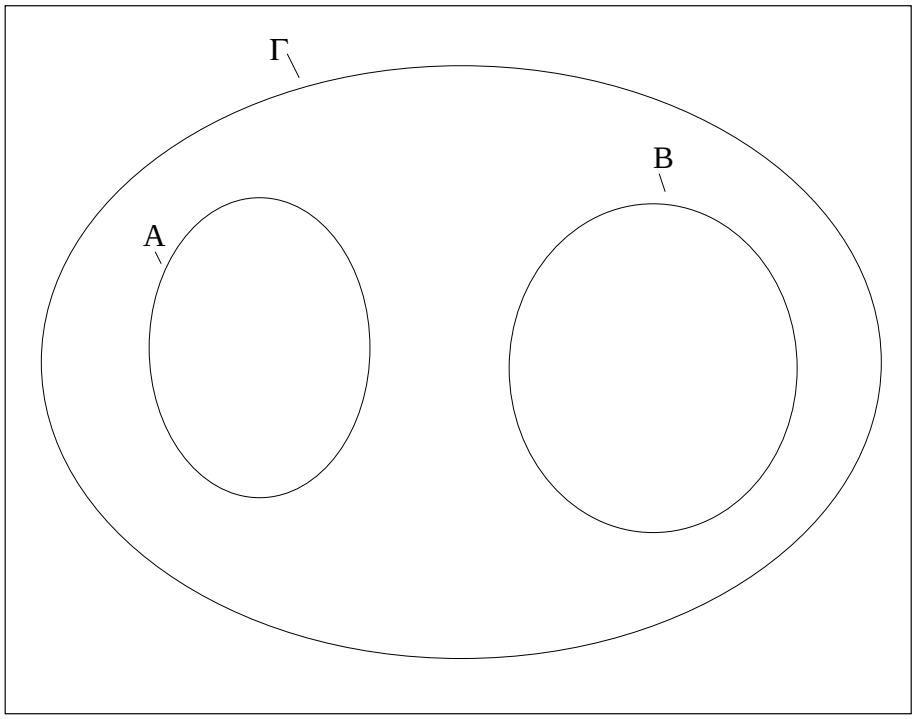

Figure 1: Anomaly classes where **A** is the set of anomalies internal to the code and **B** is the set of anomalies outside the code. In this simple case, **A** and **B** are disjoint.

> Since it is not possible to inject every possible anomalous operational scenario and then observe the software's behavior, it is necessary to somehow ensure that the anomalies injected are representative of the types of anomalies that the software will experience during its life-time.

Even if it could shown that the simulated anomalies *always* behave as if they were real anomalies, i.e., they are completely representative, there remains one serious problem left to resolve: *Since the space of potential problems is infinite, the space of anomalies that could be simulated is also infinite.* What do we choose to inject and what do we choose to ignore?

Addressing this question is easier if we set up a framework within which to approach the problem. We begin by looking at the two key classes of anomalies that software can experience: (1) internal problems and (2) external problems. We aim to demonstrate that even though both of these spaces are huge, there is only a small number of problems that we actually care about — specifically, those that will affect the software after it is deployed. If we can show that we will likely be able to simulate anomaly classes that will exhibit behavior patterns equivalent to these important few, then we have conjured up "the crystal ball" that we so badly need.

To begin, we will assume that no modifications will be made to program **P** during its analysis. This constraint is relaxed in Section 3.3.

In Figure 1, we let Γ represent the space of *all* undesirable anomalous events that *could* affect **P**'s output during **P**'s life-time (if they were to occur).[5] It is not necessarily the case that all members of Γ *will* affect **P**'s output, just that they are capable of doing so. Hence it is also not

[5]Each anomalous event in Γ is in theory detectable by examining the appropriate software state, but they may not be detectable in **P**'s output.

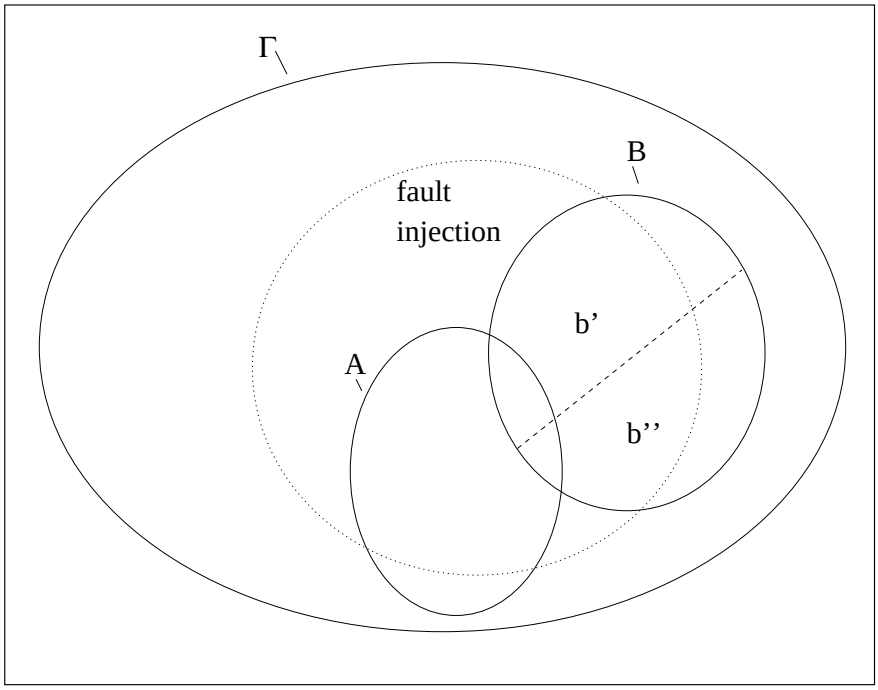

Figure 2: Anomaly classes where **A** and **B** overlap. Once again **A** is the set of problems internal to the code and **B** is the set of problems outside the code.

necessarily the case that all members of Γ are liability concerns. In Figure 1, **A** denotes that portion of Γ representing potential *internal* anomalous events (i.e., program state corruptions that are with respect to the input space of **P**) caused by existing program logic errors *in* **P**, and **B** denotes potential *external* anomalous events caused by human factor errors or external hardware and software failures that could be foisted on **P** during execution.

Admittedly, most of Γ's members are unknown, and hence Γ can be likened to a black-box. **A** represents either: (1) an "effectively" infinite and unknown space of anomalies, or (2) an empty and unknown space of anomalies. **B** represents an "effectively" infinite and partially unknown space of anomalies. $\mathbf{A} \cup \mathbf{B} \approx \Gamma$. Those anomalies in Γ that are not in **A** and not in **B** represent state corruptions that are not currently possible in **P**'s state as it executes with its current input space, but those anomalies could be forced into **P**'s state as it executes (*e.g.*, by fault injection methods). Obviously, those members in Γ that are in neither **A** nor **B** are not anomalies that we care about for purposes of liability.

In some situations, an anomalous event may not occur until both an external problem occurs and an internal logic error is executed. This situation is shown in Figure 2 (which we believe is a more realistic portrait of how most instantiations of Γ will truly be partitioned). In Figure 2, anomalies in $\mathbf{A} \cap \mathbf{B}$ represent those anomalous events caused by bad incoming information and exercised faults. For example, it might be the case that bad incoming data is required to exercise a fault in a manner that results in an anomalous internal event.

For the purposes of software liability, the impact of the anomalies that are exclusively in **A** will be determined, to some degree, by testing the program **P** in the usual way. Traditional testing,

then, gives us a handle on the liability consequences of the anomalies in **A**.[6] It will not, however, be capable of predicting the impact of those anomalous events outside of **A** (including the members of **B**). In the more complicated case of Figure 2, any anomaly in $\mathbf{A} \cap \mathbf{B}$ will require more than traditional testing to simulate. That is, traditional testing will be able to assess the impact of the **A** anomalies, but the **B** anomalies require "something else."

Fault injection can play a role in providing this other information. Stated simply, simulating anomalies in **B** is possible using fault injection. This is because fault injection can be used to observe what effect external errors (such as sensor failures, corrupted file information, and human error factors) will have on **P**. Once again, the more complicated case in Figure 2 maps in a straightforward way into this discussion. It is easy to see that in order to simulate anomalies in $\mathbf{A} \cap \mathbf{B}$, both fault injection and testing must be applied.

Let us put aside for the moment the complications resulting from the existence of anomalies in $\mathbf{A} \cap \mathbf{B}$. We can further divide **B** into two partitions shown in Figure 2, b' and b'', where b' represents those problems that *will* occur in **P**'s future, and b'' represents those problems that could possibly occur in the future but in actuality *will not*. b' is of finite size, and b'' is probably of infinite size for most **P**s. b' and b'' are unknown both before and after fault injection analysis is performed. We cannot know which partition any particular member of **B** belongs to.

To get a handle on the liability created by members of **B**, the best that can be done is to simulate as many members of **B** as are known (without ever knowing whether such anomalies are members of b' or b''), and then sample as many other members from Γ as resources allow. In practice, however, we are likely to not know exactly what space a particular simulated anomaly belongs to, as illustrated in Figure 2. The dotted circle in Figure 2 shows the anomalies simulated by random fault injection, i.e., corrupting internal data states in a *partially* random manner. We say "partially", because the part of the data state that gets corrupted is fixed. For example, if the most recent source code statement that was executed affected the value of some variable `x`, then we would only corrupt `x` in the data state. Doing so simulates the failure of the computation on `x`. The manner by which data in `x` is corrupted is where randomness comes into play. If the most recent source code statement affected control flow, then we would modify the result of that decision by forcing an alternative (selected at random) to be taken as the next statement executed. Doing so simulates the failure of the conditional expression.

The dotted circle in Figure 2 encompasses anomalies from **A**, b', b'', and those not in **A** and not in **B**. It would be Utopian if the fault injection methods employed on **P** only happened to simulate members from b', but to do this requires either incredible luck or omniscience. In any case, since b' is unknown, criteria for success would be impossible to develop as well. Our approach for simulating the anomalies in the dotted circle in Figure 2 is to inject anomalies intrusively using state corruptions, not mutants. The code that we add to modify internal states is termed a *perturbation function*. A procedure for coding perturbation functions is further described in [11].

Some of **B**'s members may be known because certain failure modes for human users, external software components, and external hardware systems (*e.g.*, sensors) that feed information into **P**

[6] This is not entirely true, since faults of very small size are likely to have no impact on **P** during testing, and hence simply running **P** is not likely to make their impacts observable. Thus it is often interesting to inject faults into such systems (even though we know they are incorrect) to ascertain how likely it is that faults could be hiding from testing. This is the basis for Voas's testability model [11].

may be known. However, empirical evidence for the benefit of selecting which anomalies to inject in a *random* fashion has been recorded in several real-world case studies [12]. In these examples, random sampling from Γ without any knowledge of any of $\mathbf{B}$'s members was applied, yet weaknesses in these systems were discovered. More specifically, in the real-world control problem, the Advanced Automated Train Control (AATC) system, Hughes was able to take corrective actions at 12 places in the code after randomly sampling from Γ. These 12 potential problems had not been discovered by any other software safety methodology. Successes using the same fault injection approach on a medical application are also provided in [12].

In these studies, no members of $\mathbf{B}$ were known, yet random sampling from Γ discovered serious potential risks within the software that were quickly repaired, thus modifying $\mathbf{P}$. After $\mathbf{P}$ was modified to thwart the possible problems caused by these risks, it was clear that $\mathbf{B}$ had been shrunk (quite possibly, b' had shrunk as well). To what degree this shrinking occurred is unknown (since we can never know how large $\mathbf{B}$ is initially). This shows that fault injection is not only useful for assessing liability, but when liability is demonstrated, it can help developers pinpoint where problems were able to propagate from, thus playing a proactive role in decreasing liability.

The next question that arises is whether or not the members of $\mathbf{B}$ that are simulated via fault injection methods will display *representative* propagation characteristics of the remaining space. By "representative", we mean that they will share both the likelihood of propagation and a similar output class. Recent research suggests that distinct, incorrect data states (for the same test case) often tend to exhibit similar propagation behaviors, regardless of exactly what the corruptions are [8]. As an example, suppose that at some point in the code some variable should have a value of 100, but has a value of 101. If this incorrect value causes a problem in the output, then it is likely that had the incorrect value not been 101 but 102, this too would cause a problem. One clue as to why this phenomenon occurs is the fact that specific partitions of $\mathbf{B}$ are likely to follow specific execution traces, since members of those partitions can only penetrate $\mathbf{P}$ at fixed points, and hence will probably follow similar subpaths of $\mathbf{P}$. If this claim is generally true, *the absolute worst* case induced by any member of b' might be discernible from observing what happens when simulating a subset of the members of $\mathbf{B}$. Results substantiating this conjecture are not yet in hand, but given the results from [8, 3, 12] (in combination with the fact that there are no other liability measurement solutions in our immediate future), this hypothesis cannot be dismissed outright without some justification. In fact, even if future research refutes the hypothesis advocating fault injection for absolute worst case prediction, fault injection will always be capable of building a list containing the leading candidates for that dubious title.

3.3 Anomaly spaces are rarely constants

Requirements, code, and operating modes are extremely fluid entities. Unlike other engineering products (*e.g.*, bridges), software systems are almost always in a constant state of flux throughout their life-times. As software evolves, it presents a moving target to risk assessment, making the risk prediction much more complex. Whenever the software changes, fault injection analysis and testing must unfortunately be reapplied.

Up until now, we have assumed that $\mathbf{P}$ was constant. If $\mathbf{P}$ is constant, $\mathbf{A}$ will remain fixed as well (assuming that the code requirements and input requirements do not change). But in the information age, little is constant — software development is a moving target, and this presents a

conundrum for risk assessment. The upshot of all this is that the assessment of risk must be liberal enough to account for minor yet significant changes to **P**, the requirements for **P**, or the inputs to **P**. Therefore we need a clearer understanding of how Γ might change if:

1. **P** and its requirements do not change, but the input requirements to **P** do change,
2. The code requirements for **P** change and **P** is therefore modified, or
3. The code requirements for **P** do not change but **P** is modified anyway.

We need to consider these situations because if we employ fault injection to predict *the* worst case, and we have simulated anomalies from out-dated **A** and **B** spaces, then our worst-case prediction can no longer be trusted. Once the code has changed, Γ, **A**, **B**, b', and b'' will all likely change as well. Changes to **B** imply that fault injection must be performed again. It would be naïve to assume that the worst-case predictions found from the anomalies originally simulated with the old code are still valid.

Even if **P** is only changed to fix bugs, then **A** will have changed. New code can introduce new bugs. Thus when **P** changes, the space of anomalies contained in **A** is likely to change as well. Those anomalies that require both internal and external events to occur (*i.e.*, those anomalies in $\mathbf{A} \cap \mathbf{B}$), are subject to change as well. Here, *regression testing*, which retests code with test cases used on earlier versions, can be relied upon to account for the impact on risk when **P** changes. To address the possibility that the important interactions between **B** and a modified **P** may have not been disclosed when the original **P** was analyzed, a modified fault injection (using different simulated anomalies) may need to be performed.

Further, software operates in an input environment, typically called its *operational profile* (**O**). **O** describes the frequency with which certain operational inputs are fed into the software. The manifestation of anomalies contained in the space Γ is dependent on the input domain of **P**. The frequency with which members of **A** can affect the state of **P** is also dependent, in part, on **O**. When fault injection is employed for worst-case predictions, simulated anomalies must be injected into **P** while **P** is executing according to **O**. Throughout a program's life, certain operating modes will be frequent and others will be infrequent. To honestly assess worst-case scenarios, less frequent operating modes must be considered. If **O** represents realistic operating circumstances, **O'**represents unrealistic (infrequent but not impossible) operating circumstances. When performing fault injection for liability analysis, it is very prudent to consider **O'** during analysis. This boils down to sampling some of the anomalous events from Γ that would occur when the code is operating in rare input modes.

3.4 Relating fault injection to liability

As already mentioned, because the anomalies injected are *hypothesized* and their likelihood of future manifestation is unknown, the results from fault injection cannot be considered as an *absolute* measure of risk (as can a measure such as *mean-time-to-failure*). Instead, the results are a *relative* measure of risk. As an example, suppose that for each run of a program, an input sensor signal is read. Suppose that on each of 1,000 executions of the software, we corrupt this signal, and 2 of those corruptions resulted in undesirable outputs. Roughly speaking, the software has a 0.2% sensitivity to unwanted output from that sensor, but that does not mean that during operation, 0.2% of

the executions are going to cause harm. If we additionally know the likelihood of failure during operation of the sensor and how close the injected anomalies are to operational sensor failures, then we can produce an absolute measure of the liability risks generated by that sensor, and get a more precise measure of expected loss. But even just the simple observation of undesirable output once during fault injection gives an indication that the code may lack the needed robustness to reduce liability risks.

The mathematics for risk and liability prediction using the results from fault injection require very simple probability estimates. We calculate percentages like the "0.2%" above for each class of anomalous events from Γ that we are interested in, e.g., one class could be for one external device. For a liberal assessment of liability, i.e., the worst-case, take the largest single probability estimate and multiply it by the expected frequency that this class would occur during operational. This provides a software liability prediction.

4 Summary

We have described one benefit of injecting anomalous, simulated states in a program: software liability analysis. Fault injection can be used to predict what events might occur in the future if the future contains the same anomalous circumstances that fault injection employed. Fault injection is not a perfect approach and is only as good as the anomalies simulated, but it does allow us to observe how bad the future could get. In other words, it allows us to build up a body of knowledge about probable future behavior. Such a capability is arguably the next best thing to having a fortune-teller's "crystal ball", and it certainly beats facing the future with no predictions at all.

Fault injection is a process that does not explicitly state how good software is *per se*. Rather, fault injection provides forward-looking, worst-case predictions for how badly a piece of code might *behave* and how frequently it might behave that way. This frequency is a *relative* metric, and not *absolute*. Nevertheless it still provides a means for observing how often outputs occur that we never desire to see after we have injected corrupt information into the software as it executes. By contrast, software testing explicitly states how good software is. But even when testing can be successfully applied, correct code that has been thoroughly tested can have "bad days" and not work as desired because of external influences. For example, a workstation might be in good working order, but if the network server is not functioning, the workstation's user may find it so unusable that it might as well be broken too. Simulating these sorts of events falls under the rubric of fault injection, and understanding the impact of these events is our justification for why fault injection should be a part of risk assessment.

Today, software developers are living in a liability "grace period" ironically afforded them by the legal profession. This is most likely because the legal system does not yet understand how software works and the insurance sector does not understand how software's potential risks can be assessed. Furthermore, the courts have not demanded of software practitioners the absolute level of professionalism that is required of other engineering disciplines. As the aphorism says, "all good things must come to an end." It is prudent to recognize this unusual period of non-liability in our profession's history for what it is — temporary. Furthermore, it is advisable for our profession to develop honest and objective risk metrics so we can all share the risk of software ventures through insurance. Not having access to such protection is like jumping out of an airplane

without a parachute — maybe you will land in a big bail of hay, and maybe you will not.

It is our contention that software fault injection is a promising technology in software quality assessment. Even if only a small fraction of all possible anomalies are explored during analysis, the results still provide useful information about how well-behaved a software system will be. The key to a successful experience with software fault injection methods is the proper interpretation of the results. When satisfactory outputs are observed after injection occurs, it is very tempting to rush to the premature decision that the software is incapable of ever producing undesirable outputs. Nothing could be further from the truth. The correct interpretation is that the software is only known to be resilient to those anomalies tried (nevertheless it *is* likely that the software is also resilient to many other anomaly instances). Somewhat counterintuitively, the greatest benefit from fault injection occurs when a piece of software does not tolerate injected anomalies. Any interpretation here is less likely to be oversold because the only honest claim is: *the software presents risks.*

Acknowledgements

This work has been partially supported by DARPA Contract F30602-95-C-0282 and NIST Advanced Technology Program Cooperative Agreement No. 70NANB5H1160. The authors wish to thank Jeffery Payne, Dr. Chris Michael, Dr. Anup Ghosh, Roger Turner, Prof. Keith Miller (U. of Illinois), Frederique Vallee (Mathix), and the anonymous reviewers for their suggestions.

References

[1] J. Arlat et al. Fault Injection for Dependability Validation: A Methodology and Some Applications. *IEEE Trans. on Software Engineering*, 16(2):166–182, February 1990.

[2] J. A. Clark and D. K. Pradhan. Fault Injection: A Method for Validating Computer-System Dependability. *IEEE Computer*, pages 47–56, June 1995.

[3] M. Daran and P. Thevenod-Fosse. Software error analysis: A real case study involving real faults and mutations. *Proc. of the ACM SIGSOFT ISSTA'95*, pages 158–171, 1995.

[4] Data Processing Services, Inc. v. L. H. Smith Oil Corp., 492 N. E. 2d 314 (Ind.App. 1986).

[5] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[6] 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.

[7] Prof. J. L. Lions. Ariane 5 flight 501 failure: Report of the inquiry board. Paris, July 19, 1996.

[8] C. C. Michael. On the regularity of error propagation in software. Technical report, Reliable Software Technologies Corporation, Sterling, Virginia, 1996. Research Division Technical Report RSTR-96-003-04.

[9] J. A. Solheim and J. H. Rowland. An Empirical Study of Testing and Integration Strategies Using Artificial Software Systems. *IEEE Trans. on Software Engineering*, 19(10):941–949, October 1993.

[10] F. Smith. Law enforcement information sharing systems as part of the solution to the effective analysis and dissemination of computer network threat intelligence. *Proceedings of the Invitational Workshop on Computer Vulnerability Data Sharing*, June 1996.

[11] J. Voas. *PIE*: A Dynamic Failure-Based Technique. *IEEE Trans. on Software Eng.*, 18(8):717–727, August 1992.

[12] J. Voas, F. Charron, G. McGraw, K. Miller, and M. Friedman. Predicting How Badly 'Good' Software can Behave. *IEEE Software*, To appear in early 1997.