

## Guiding fault-injection and broadening its scope using ODC Triggers

**Ram Chillarege**

Center for Software Engineering  
IBM T.J. Watson Research Center  
30 Saw Mill River Road, Hawthorne NY 10532  
(914) 784 7375, ramchill@watson.ibm.com

### Abstract

- *This paper draws on the relationship between rapid reliability growth and the trigger profile during the early part of a product's release in the field.*
- *Trigger distributions for the first eight quarters of a product are illustrated and compared with the software reliability growth experienced in the same period. These data illustrate the specific triggers that contributed towards the high reliability growth as compared to the others that did not.*
- *The techniques of fault-injection can benefit from the knowledge of triggers and also assist testing and reliability growth by emulating an environment closer to that of the customer.*

*Thus, the ideas of fault-injection are made applicable to a much larger domain - namely of products which don't have to be fault-tolerant in design to take advantage and improve overall system reliability.*

### Introduction

The purpose of this paper is to explore the concept of fault-injection to a much broader area of application. The traditional view is that fault-injection is used to evaluate the fault-tolerance capability of a system. In addition, fault-injection is a method of testing used for parts of the system that would not be exercised without a fault. The area has certainly seen a large growth in papers and experiments [1,2,3,4,6]. However, the domain is fairly restricted. Since the ideas are generic and the question is what are the limits that this concept can be extended to provide greater value?

Most software that is produced in the industry is not what would be classified as "fault-tolerant". However, most software is designed to tolerate certain kinds of faults. The capability is designed at different levels in the product or system. For instance, error handling is a common aspect of most software. Error handling routines are written to deal with human errors, system interrupts, peripheral devices, etc. These features are now taken for granted and are not labeled as fault-tolerance, although that is precisely what they provide. A more sophisticated kind of fault-tolerance would permit system state or data to be reset across power cycles. Such enhancement is more visible in products such as data bases where there is a significant effort to maintain the transaction history in a log and recover data blocks which could be obliterated by a power outage or crash of the underlying operating system. Such methods are often categorized as fault-tolerance or sometimes not, largely depending on sophistication or the adherence to fault-tolerant principles. Often a software product gets labeled as "fault-tolerant", depending on how it is positioned. Thus, very few products in the market have the label of fault-tolerance, but do have a degree of fault-tolerance in it.

The issue we explore is how the concepts of fault-injection could provide value to the larger domain of software, not all of which fit into the strict definition of fault-tolerance.

Before we dive into this topic, let us briefly examine the larger goals. All software that is developed goes through some development process which may involve inspections, and certainly function testing, system testing, and so forth. These steps help remove faults inserted during the development process or the design. Eventually, when the product is released to the field it demonstrates a certain level of software reliability in the eyes of the customer. The ultimate goal of fault-tolerance and testing is to improve this level of reliability the customer experiences. As far as the customer is concerned, a failure causes an impact, regardless of the cause, be it a design fault, an implementation fault, or an external event. If fault-injection methods could improve the reliability of software not designed to be "fault-tolerant", it would help towards achieving the larger goal. The question is how?

## Reliability and Trigger Profiles

In this section we bring together a few ideas to make better progress and to improve the overall reliability experienced by the customer.

Every software product goes through some form of reliability growth in the field. This growth is mostly accomplished by fixing of defects as they are discovered. An underlying phenomena which is present enables the detection of these latent software defects. The phenomenon is the set of triggers that works as a catalyst to allow faults to surface. We begin by trying to relate these two phenomenon. This relation gives us a more clear understanding of the underlying processes at play. Understanding the relationship precisely gives us new tools that can be exploited.

This discussion is best conducted with some real data. The two types of data that we would use are: (1) the overall software reliability growth experienced in the field and (2) the profile of triggers that helped surface the failures during the same period.

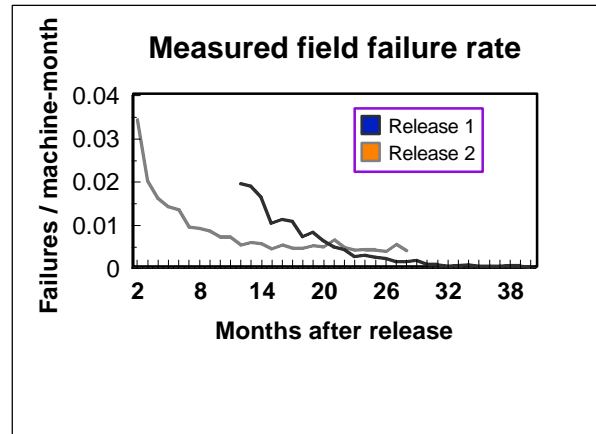


Figure 1. Software Failure rate measurements from a widely distributed software product. Service data is consolidated with development defect data to assign f to a specific product. Compensated for change in lice base in the field.

Figure 1 shows the software failure rate as a function of time for a little over 24 months after the product was released. The improvement that occurs in the failure rate is primarily due to the identification of unique faults and there subsequent removal by a fix. As more faults are fixed and the product stabilizes, the failure rate drops significantly

This is a well-accepted phenomenon with numerous mathematical models. However, these data are the first actual measurements on a widely distributed software product.

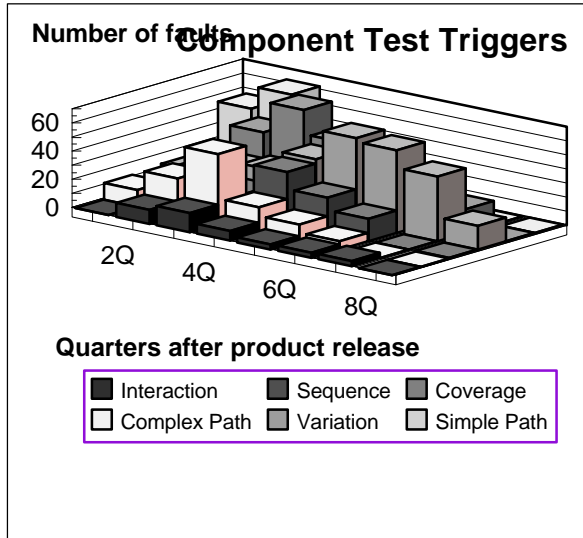


Figure 3. Software triggers that are typically associate with function and component test. Signify the type of cases that should be developed.

Figure 2 shows the trigger profile from defects found into the first eight quarters (2 years) after product release [7]. Since it is a relatively new concept, discussion on triggers together with their individual definitions is provided in the appendix. For now, we need to understand that each unique fault has a corresponding trigger associated with it. The trigger is primarily the mechanism that facilitated the fault into surfacing and caused a failure. It is not the cause of the fault and neither is it the cause of the failure (since, the cause of the failure is the fault). However, it is the reason why a particular fault surfaced resulting in the failure. Thus, for each of the unique faults, which when fixed improved the software reliability there is a corresponding trigger which helped in the improvement.

Figures 3 and 4 show two additional trigger distributions from during the same period. Figure 3 has triggers categorized as function test triggers and Figure 4 as that of review and inspection. Essentially, any failure that occurs in the field is categorized into one of the three trigger categories depending on what the most likely event was which triggered the failure. These triggers again reflect similar trends in that there are certain triggers that are more dominant in the early part of the product

life compared to the others. This gives us an insight into the kinds of triggers that are responsible for the rapid increase in software reliability. Essentially, certain trigger conditions are prevalent in the customer workload and the product is susceptible to these trigger conditions. Had these conditions been applied during system tests it is very likely that these failures would have been caught earlier. The relative proportion of faults under the three different categories of triggers gives us further information as to which one of the three trigger groups need to be emphasized. This is mostly dependent

Figure 4. Software triggers associated with inspector type activity. Volume also signifies the escape from early phases of development.

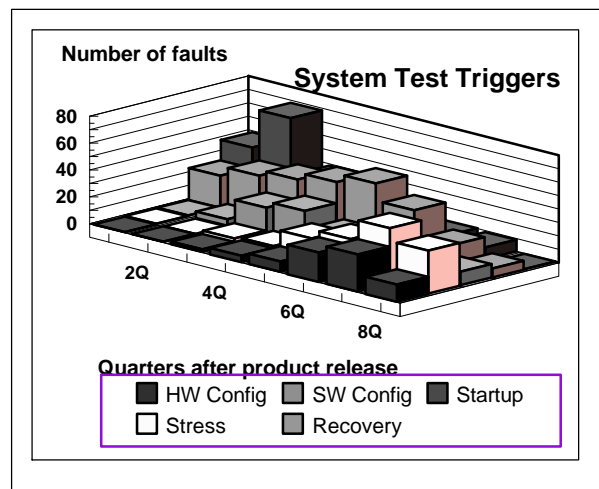


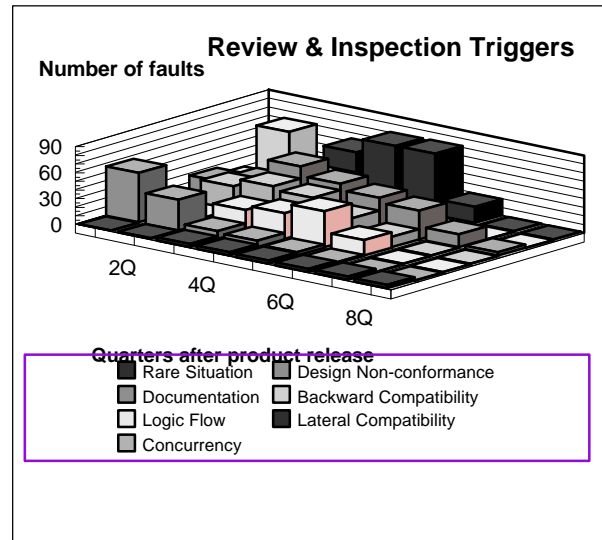
Figure 2. Software triggers volume shown as a function of time. Each field defect is assigned its trigger - these specific triggers are commonly associated with system test.

on product and process maturity. Profiling a particular product by these triggers allows one to develop a strategy for the very next release.

## Guiding Fault-Injection

Fault-injection has the difficulty that it is a fairly expensive process unless it is carefully guided. The set of faults that could possibly be injected is a very large set and one has to judiciously use that subset of faults so they best represent real faults existing in software. The domain into which faults are injected, namely the product, is again a very large space. Thus, the combination of what fault, where and when, becomes an extremely large space or possibilities. To conduct a fault-injection experiment with reasonable input coverage[7] is similar to the testing problem which is combinatorially explosive. However, it is different from the testing problem in that fault injection focuses on the likely ways the product could fail. Whereas, testing focuses on correct behavior which could span a larger set of possibilities. The question of how to reduce the fault-injection space by specifically identifying the critical faults and error conditions resulting in high impact failures is discussed in [9].

The issue that remains open is controls on the environment. A fault-injection experiment involves not only the injection of an error but also the set up of an environment. The environment should reflect the conditions under which the system is expected to operate and potentially provide a few controls to accelerate the failure process. The classical view of acceleration has mostly been the use of stress. Stress is not uncommon in software testing or fault-injection. Over the past several years that has primarily been the means of acceleration. However, from the trigger profiles it is evident that there are several other factors that are involved in allowing faults to surface. From the trigger profiles it is evident that no one particular trigger mechanism overwhelmingly dominates all the time. Different triggers dominate at different periods of time over the first several quarters after product release. These provide better information on what else ought to be used to accelerate the failure process.



A trigger captures the environmental condition prevalent just prior to the occurrence of a failure. This environmental condition could be present for either a short or long duration prior to the failure. However, it is present during the time just leading up to the failure. As to when a failure occurs in the field is unknown; there are no controls. On the other hand, it is far more controllable during fault-injection. Thus, knowing the triggers gives fault-injection a significant new control to emulate field conditions during an experiment. The question is how does one force a trigger condition in software?

There are a few methods to do so. Firstly, some triggers are related to specific operations of the product such as start-up and shut-down. These are the kinds of operations under which a fault-injection would be conducted to bring in the influence of the trigger. There are other triggers which are more environmental such as a software configuration and hardware configurations. These can be emulated by setting or upsetting certain environment parameters in the software. Finally, a trigger condition such as recovery might require the injection of an additional error.

Let us reflect on the purposes for fault-injection. We stated that there are primarily two major objectives. One is to evaluate the fault-tolerance capability of a system and the other is to improve the testing processes. In both these cases, knowing

the trigger conditions that actually result in a failure is vital to represent or design a fault-injection experiment.

## Triggers and Operational Profile

It is useful to reflect on how triggers relate to operational profiles. It has been argued for some time that software testing should follow the use of an operational profile[5]. An operational profile is represented by several functional, customer and usage parameters. Some of the parameters may be application or environment specific and some more general. The idea being that the workload used in testing closely corresponds to the field.

When the defects result from the use of an operational profile, a corresponding trigger profile can be computed. The trigger profile is the resultant of the use of an operational profile on the product. Capturing very concisely a fairly large amount of information on cause and effect.

From a fault-injection perspective focusing on a specific set of trigger conditions reduces the number of fault injections that need to be conducted so that one gets a reasonable coverage of the input space. This is different from the discussion on what faults to inject and where to inject [9]. However, it is an equally important step is to create the right experimental conditions under which these faults ought to be injected.

What is hard to establish is that a particular operational profile does indeed a customer situation. This is a difficult task because the number of parameters used to capture the essence of an operational profile is not well defined and is complicated. However, since the triggers of the defects found is the resultant of the use of an operational profile, the triggers in and of themselves will provide us a measurement of the operational profile that is in use. So, if the trigger distribution of faults found through the process of testing is similar to the trigger distribution of faults found in the field, an argument can be made that the operational profile in use in testing is similar to that in the field. This is

a very important observation since it gives us a quantitative method of assessing the completeness of a test suite.

Since triggers are environmental conditions which can be set up either through workload, controls within software, or fault-injection, they provide a controlling mechanism. From the trigger profiles presented in the figures it is clear that there are certain triggers that are dominant in the early part of the products life. This is when there is a significant growth in the overall reliability. Emulating these triggers through the testing process would ensure that we are on an aggressive path to improving the systems reliability.

## Conclusions

In this paper we explore an avenue to enlarge the scope of the fault-injection class of techniques to provide greater value in improving software reliability. Central to this theme is exploiting the knowledge we have from software triggers and bringing that understanding to assist fault-injection. There are really three components to bringing this idea together:

1. We recognize that there is a steep increase in software reliability in the early parts of the field life. When this is contrasted against the profile of software triggers that assist in the detection of defects, we understand the environment that has allowed such a reliability growth to occur. It is this understanding through the trigger profiles that we can exploit for better test planning.
2. On an independent note, triggers are a key component to fault-injection. They provide the mechanisms of accelerating experiments beyond the simple minded stress that has been used so far. Thus, fault-injection that uses the triggering conditions to conduct experiments reflects more of the field environment in the experiment.

3. Ideally, if system test could conduct tests that reflect better the field conditions it is more likely to enhance the product's software reliability. Since several trigger conditions are actually exercisable in the system through careful fault-injection, it provides a mechanism by which system test can be enhanced to better emulate the field condition.

Thus, coupling ideas of the operational profile, triggers, and fault-injection to direct a test environment which mimics characteristics of the early field experience can provide a significant advantage over traditional methods to improve product reliability.

## Appendix

### **Software Triggers and ODC**

Software faults are dormant by nature. This particularly true when considering faults that cause failures once a software product is released in the field. Faults which surface as failures for the first time after a product is released often have been dormant throughout the period of development, which can range from a few months to a few years.

What is it that works as a facilitator, activating dormant software faults to result in failures? That catalyst is what we call the trigger. Triggers do not identify the specific sensitization that is necessary for each unique fault to be exercised. Instead it identifies the broad environmental conditions or activities which work as catalysts assisting faults to surface as failures. In an abstract sense, these are operators on the set of faults to map them into failures.

There are specific requirements for a set of triggers to be considered part of orthogonal defect classification, and a process to establish them[10]. We do not attempt to completely explain the

concepts here and the details of the necessary and sufficient conditions are best found in the Handbook of Software Reliability Engineering. However, to briefly summarize the ideas, it requires that a set of triggers form a spanning set over the process space for completeness. This is when the trigger value set is elevated from a mere classification system into a measurement and qualifies to be called ODC. Changes in the distribution as a function of activity then become the instrument, yielding signatures. The value set has to be experimentally verified to satisfy the stated N+S conditions. Unfortunately, there is no short cut to figure out the right value set. It takes several years of systematic data collection, experimentation and experience with test pilots to establish them. However, once established and calibrated, they are easy to roll out and productionize. We have the benefit of having executed ODC in around 50 projects across IBM providing the base to understand and establish these patterns.

### **Trigger Definitions**

There are three classes of triggers associated with the three most common activities of verification, system test, function test and review/inspection. Since each of the activities tend to be very different from one another, they call for different sets of triggers. The distinction arises from how they attempt to emulate customer usage. However, the level of observability of the code and function vary and the issues addressed are driven by different motivations.

### **System Test Triggers**

System test usually implies the testing that is done when all the code in a release is mostly available and workload similar to what a user might generate is used to test the product. These triggers characterize that which the regular use of the product in the field would generate. They, therefore, apply to system test in the field.

**Recovery/Exception Handling.** Exception handling or recovery of the code was initiated

due to conditions in the workload. The defect would not have surfaced had the exception handling process or the recovery process not being called.

**System Start-up and Restart.** This has to do with a product being initiated or being shutdown from regular operation. These procedures can get significantly involved in applications such as database. Although this would be considered normal use of the product, it reflects the operations that are more akin to maintenance rather than prime time.

**Workload Volume/Stress:** Indicates that the product has been stressed by reaching some of the resource limits or capability limits. The types of stresses will change depending on the product but this is meant to capture the actions of pushing the product beyond its natural limits.

#### **Hardware Configuration and Software**

**Configuration:** These triggers are those that are caused due to changes in the environment of either hardware or software. It also includes the kinds of problems that occur due to various interactions between different levels of software precipitating problems that otherwise would not be found.

**Normal Mode:** This category is meant to capture those triggers when nothing unusual has necessarily occurred. The product fails when it was supposed to work normally. This implies that it is well within resource limits or standard environmental conditions. It is worthwhile noting that whenever normal mode triggers occur in the field it is very likely that there is an additional trigger attributable to either review or function test that became active. This additional level of classification by a function test or review trigger is recommended for field defects.

#### **Review and Inspection Triggers:**

When a design document or code is being reviewed, the triggers that help find defects are mostly human triggers. These triggers are easily mapped to the skills that an individual has, providing

an additional level of insight. These triggers are not execution related. Therefore they do not play a role in the fault-injection or testing directly. They are briefly included here for completeness.

**Backward Compatibility:** Has to do with understanding how the current version of the product would work with earlier versions or other products that it interacts with. Usually, requires skill beyond just the existing release of the product.

**Lateral Compatibility:** As the name suggests, this trigger has to do with how this product would work with the other products within the same software configuration. The experience required by the individual should span the sub-systems of the product as also other sub-systems that the product interacts with.

**Design Conformance:** These faults are largely related to the completeness of the product being designed with respect to the requirements and overall goals set forth for the product. The skills required for finding these kinds of triggers has more to do with an understanding of the overall design as opposed to the kinds of skills required to ensure compatibility with other products.

**Concurrency:** Has to do with understanding the serialization and timing issues related to the implementation of the product. Specific examples are locking mechanisms, shared regions, c-sects etc.

**Operational Semantics:** Has to do largely with understanding the logic flow within the implementation of a design. It is a trigger that can be found by people who are reasonably new but well trained in software development and the language being used.

**Document Consistency/Completeness:** Has to do with the overall completeness of a design and ensures that there is consistency between the different parts of the proposed design or implementation. The skill is clearly one that mostly requires good training and implementation skills and may not require significant in-depth understanding of the products, dependencies, etc.

**Rare Situation:** These triggers require extensive experience of product knowledge on the part of the inspector or reviewer. It also recognizes the fact that there are conditions that are peculiar to a product which the casual observer would not immediately recognize. These may have to do with unusual implementations, idiosyncrasies or domain specific information which are not commonplace.

### **Function Test Triggers**

The triggers that are described here really reflect on the different dynamics that go into test case generation. Therefore, it is very reasonable to actually do the classification of triggers when the test cases are written. It does not have to be, that the triggers are identified only after a fault is found. These test cases should also be mapped into the white box and black box of testing.

**Test Coverage:** refers to exercising a function through the various inputs to maximize the coverage that is possible of the parameter space. This would be classified as a black box test trigger.

**Test Sequencing:** Are test cases that attempt to sequence multiple bodies of code with different sequences. It is a fairly common method of examining dependencies which exist that should not exist. This is also a black box test.

**Test Interaction:** These are tests that explore more complicated interactions between multiple bodies of code usually which are not covered by simple sequences.

**Test Variation:** is a straight forward attempt to exercise a single function using multiple inputs.

**Simple Path Coverage:** A white box test that attempts to execute the different code paths, increasing statement coverage.

**Combination Path Coverage:** Another white box test that pursues a more complete signal of code paths, exercising brands then different sequences.

### **References**

- [1] J. Arlat, et al., "Fault Injection and Dependability Evaluation of Fault-Tolerant Systems", IEEE Trans. Comp., Vol. 42, No. 8, pp. 913-923, August 1993.
- [2] R. Chillarege, N.S. Bowen, "Understanding Large System Failures A Fault Injection Experiment", Proc. 19th Int. Symp. Fault Tolerant Computing, pp. 356-363, June 1989.
- [3] G.A. Kanawati, et al., "FERRARI: A Flexible Software Based Fault and Error Injection System", IEEE Trans. Computers, Vol. 44, No. 2, pp. 248-260, February 1995.
- [4] I. Lee, R.K. Iyer, "Faults, Symptoms, and Software Fault Tolerance in the Tandem GUARDIAN90 Operating System", Proc. 23rd Int. Symp. Fault Tolerant Computing, pp. 20-29, June 1993.
- [5] J.D. Musa, "Operational Profiles in Software Reliability Engineering", IEEE Software, Vol 10, No. 2, pp 14-32, March 1993.
- [6] Z. Segall, et al., "FLAT - Fault Injection Based Automated Testing Environment", Proc. 18th Int. Symp. Fault Tolerant Computing, pp. 102-107, June 1988.
- [7] R. Chillarege and K. Bassin, "Software Triggers as a Function of Time - ODC on Field Faults", DCCA-5: Fifth IFIP Working Conference on Dependable Computing for Critical Applications, Sept. 1995.
- [8] D. Powell, et al., "Estimators for Fault Tolerance Coverage Evaluation", IEEE Trans. Comp., Vol. 44, No. 2, pp. 261-274, Feb. 1995.
- [9] J. Christmansson and R. Chillarege, "Generation of an Error Set that Emulates Software Faults - Based on Field Data", Proc. 26th Int. Symp. Fault Tolerant Computing, June 1996.
- [10] Handbook of Software Reliability Engineering, Michael R. Lyu, Editor, Computing McGraw-Hill, 1996.