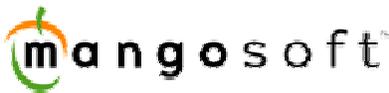


Targeted Software Fault Insertion

A Holiday at Faulty Towers

Paul J. Houlihan
Mangosoft Corporation
1500 West Park Drive, Suite 190
Westborough, MA 01591

paulhoulihan@mangosoft.com



Abstract

Since the completely random software fault insertion techniques suggested in much of the research literature[1] are not practical for most software products, this paper suggests that a modest targeted software fault insertion effort for a few common error conditions can have a dramatic impact on defect detection rates and quality. The paper uses the example of a software fault insertion subsystem, code-named Faulty Towers, which was added to Mangosoft Incorporated's test automation in order to target common failures and errors. Mango software, like so much other software, obeys the maxim that 90% of the code is written to handle error conditions. To help target the bulk of the code, Mango software can simulate common failures and errors, which are referred to as faults. With precise fault insertion rates intentionally selected by a test, the exact stimulus that led to a failure is often well understood. This leads to quicker problem isolation and allows QA to return to the exact failure environment when attempting to reproduce a problem or to validate a fix. This paper presents data on the effectiveness of software fault insertion, discusses the advantages and risks of fault insertion, provides tips on gaining cultural acceptance for fault insertion and suggests high payback areas for fault insertion which have proven themselves over multiple products. In a typical software development cycle, defect detection starts to trail off once the mainline code stabilizes. With software fault insertion, it was found that the defect detection rate does not level off and the hardest task becomes not one of finding defects but one of prioritizing the stream of defects.

Keywords

fault simulation, fault insertion, software fault insertion, test automation, software testing process, peer-to-peer

Why Read This Paper?

In a busy world, the idea of software fault insertion (actually faking errors within software) seems hard to justify. Doesn't software and testing generate enough errors on its own? Why bother with the extra expense and infrastructure? These are some of the questions answered by this paper. The data in the next section documents the compelling payback of an organized software fault insertion approach to testing error handling code paths. Having considered the data, the paper describes the benefits and risks of software fault insertion and shows that the costs of the fault insertion infrastructure can be minimal. It then presents a process to identify and prioritize faults.

Fault Insertion Testing Data

Distributed caching is a key Mangosoft specialty. The data in this section is for a product called Medley™ which is a distributed file system for Microsoft's Windows-95, Windows-98 and Windows NT operating systems that provides performance exceeding the fastest server hardware. While the Medley drive appears like a local PC drive, it is really a distributed, peer-to-peer LAN disk spread over up to 25 PCs. Files on the distributed LAN drive migrate to the local disk of the user who accesses them the most. The product also has data mirroring built in to eliminate single points of failure that are common in most client server solutions. Thus drive data can reside anywhere on 25 PCs and yet Medley must maintain drive reliability in the face of different operating systems, device errors and PCs that randomly crash or shutdown.

The fault insertion data in this section was gathered at Mango where we have combined substantial test automation with fault insertion to produce a prodigious testing capacity [2]. An extensive database stores test results on every automated test ever run in the history of the company. Figure 1 is derived from this database and shows the testing volume differentiating non-fault tests from fault tests which reflects the *classic fault insertion cycle*. In this cycle, fault testing is not needed in the early stages of a product version when defects are spread uniformly throughout the code. However, as the mainline code paths stabilize, fault testing of error code paths becomes a sizable percentage of testing. Near the end of a product version, fault testing is reduced in favor of maximum stabilization of mainline code paths, although fault testing is never entirely stopped. Overt control of the testing mix is a great advantage during the product lifecycle.

The term *fault* is used in this paper to mean an externally controlled mechanism to alter code path execution within a program in order to increase error handling code coverage and robustness. External control allows the tester as well as developers to manipulate the fault. In QA research literature, fault has a negative connotation, synonymous with code defect. But at Mango, a fault is considered a good thing, used more in the hardware testing sense as a powerful tool to exercise challenging areas of the product to assure that the highest levels of quality are consistently attained in the shortest amount of time.

Figure 1: Mango Monthly Test Counts

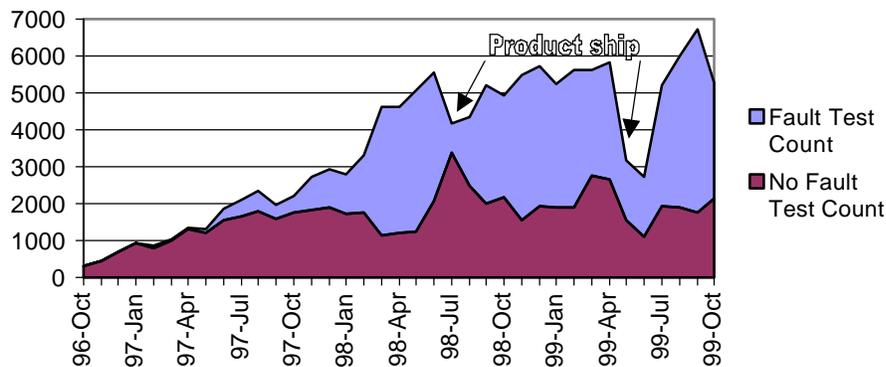


Figure 2 shows the new product problem reports (excluding duplicate reports and test automation problems) that were produced from the testing in Figure 1 starting in July 1998, when a new version of Medley was begun. Data from 98-Jul to 98-Aug shows that initially, faults are not needed, as simple non-fault tests find plenty of problems. However, as the mainline code stabilizes, non-fault tests find fewer problems. The tremendous insight offered by the volume of fault insertion problem reports dispelled any illusion about error handling code path quality, greatly informing the entire development process. Interestingly, at the end of the Medley version in May 1999, even though fault insertion testing was cut way back (see Figure 1), fault insertion problem reports equaled the number of non-fault problem reports. The sustained effectiveness of fault insertion testing is visible in the graph. Over the entire period covered by Figure 2, non-fault testing produced 37 % of all problem reports while fault testing produced 63 % of all problem reports.

Figure 2: Problem Reports by Month

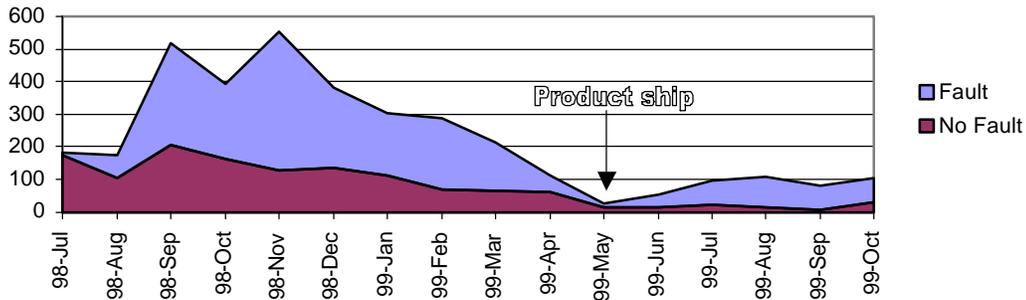
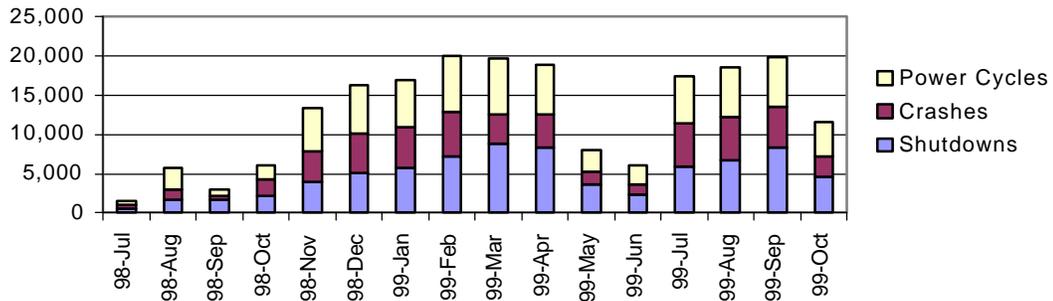


Figure 3 shows data since July 1998 on three automated faults: the number of shutdowns, crashes and power cycles by a node participating in a Medley drive. It was clear how central these common computer faults would be to the success of a shared, peer-to-peer LAN disk product and Mango would doubtless have done quite a few faults by hand if a manual testing approach had been adopted. But by automating these faults and combining them with an automated test infrastructure, Mango QA created an army of test lab machines performing these faults around the clock, day in and day out. [8] From July 1998 until October 1999, using test automation, Mango QA inserted into Medley drive configurations 76,482 shutdowns, 53,898 crashes and 72,725 power cycles. This utterly surpasses by orders of magnitude what Mango QA could have achieved manually, at much greater expenditure of money, time and resources.

Figure3: Insertion Counts for Three Faults



The Problems

Grasping the complexity of modern software products layered on multiple operating systems, databases, run time libraries, third party products and various hardware platforms is akin to the job of astronomers who attempt to grapple with all the known matter in the universe. There are infinities layered upon infinities: countless galaxies containing countless solar systems, containing countless planets. The limitless nature of the universe is comparable to the states entered by complex software products like operating systems that run on millions of PCs. The problems only increase in distributed software where algorithms are spread over multiple computers. At half a million lines of code, Medley is a distributed file system that can be spread over 25 PCs running Microsoft's Windows 95, Windows 98 or Windows NT operating system. A file's data can reside anywhere on the 25 PCs but Medley must provide the highest of data integrity guarantees in the face of different operating systems, device errors and PCs which can shutdown or crash at will. The combinatorial complexity is exponential.

The current state of the art is inadequate to the task of producing software able to encompass all of this complexity or to cost-effectively test this complexity. Typically, development design specifications focus on the mainline structure and flow of information through the subsystem and do not focus on what errors might occur and how they might be handled. The error handling and exception processing are often left as details for implementation, making them subject to error-prone code rewrites and re-design late in the development cycle. This occurs despite the fact that mainline code comprises only about 1-10 % of the code, while the bulk of the code is devoted to error handling. Added to this situation, there is constant pressure to produce software faster and cheaper. Time to market can spell the difference between success and failure for a startup software company.

The problems resulting from these development realities are:

- *Defects in error handling code are not discovered until the very expensive latter stages of the development cycle.* The lack of a formal focus on errors means that the bulk of defects in error handling code paths are left to be discovered at the most expensive step in the development cycle – system integration testing [3]. The chaos that results when all the subprojects in a new version are first integrated can be dramatic. Engineers in their unit testing cannot easily insert common failures in their subsystem or failures in other surrounding subsystems. This results in unit testing that is less systematic and less robust. The bulk of the error handling code is left to be exercised when integrated with all the other changes in the system. The resulting chaos produces a system that is not even useable for days or weeks, incurring substantial schedule slips.
- *QA testing focus on error handling code paths is often undirected and without prioritization.* With no error or fault insertion controls available to testers, testing is often relegated to varying user loads and types of user transactions. Testers focus on what they believe to be “typical” user interactions, exercising only mainline code paths. Error handling code path testing is often undirected, relegated to those errors that happen to occur accidentally in the course of testing. Testers are also unable to prioritize testing, targeting the most important error handling code paths first.
- *Manual QA fault testing is typically tedious and very expensive.* The faults that are intentionally applied are often manual, tedious and time consuming. In the case of a distributed product that relies on network communication, there are just so many times one will pull out the network cable between two machines to verify that the resulting communication failure is handled appropriately. To test disk full handling, it takes time to fill a disk completely full. Perhaps the most expensive single area of testing is large scale testing. Medley supports configuration of 25 computers which can be an unwieldy distributed testing environment. Time consuming testing of such large configurations is often expensive to configure and manipulate, therefore it rarely receives adequate coverage. The manual aspects of such fault testing are very expensive and take great discipline on the part of the tester to be rigorously performed the same way for each release of the product. What is tedious and time consuming is less likely to be done thoroughly in every release cycle. Without automation of faults, testing is thereby more expensive to perform and less comprehensive.
- *Tempted to ship product as “illusion of quality” sets in.* Once the mainline code stabilizes, fewer “accidental” errors arise to exercise the error handling code paths. Problem reports tend to trail off.

This leads to the illusion of quality, the fatally attractive notion that since mainline code paths are exhibiting a level of stability and quality, that error handling code paths likewise have this same level of quality and maturity. Unless the tester can make plain the level of quality over the whole product, there can be tremendous time-to-market pressure to ship software once the problem reports trail off. The time it typically takes to stabilize the mainline code paths, coupled with the high cost of traditional approaches to testing error code paths, means that error handling is invariably the least well tested aspect of the software.

The Solution

How do developers test failures from other subsystems with which their code interacts? They might set a breakpoint and fake an error. By changing a success return status from an I/O system service into an error status they can test how their code handles I/O failures. This is a powerful concept but isn't very flexible and doesn't scale. If one wanted to fail all I/O, this technique would result in a lot of breakpoints firing and a lot of success statuses to override. Once the breakpoint is removed, the knowledge of the most effective point for fault insertion is lost, it can't be used by other developers in unit testing or by testers in their system testing.

The solution of software fault insertion builds on this simple unit testing technique. The first step is to add code to simulate the fault in the subsystem. In the I/O error example, the developer creates code to replace the success status with an error. The next step is to build a fault infrastructure that allows external control over the frequency that the fault insertion code executes. The fault insertion code for the I/O Error example would be surrounded by an IF statement containing a call to the fault infrastructure to determine if the fault is to be inserted for this I/O. With the fault and fault infrastructure in place, a tester can externally request that the fault be inserted 25 % of time, that is, 25 % of all I/Os will fail with the selected error.

The fault infrastructure should also provide external reporting on the number of fault insertions that have occurred in order for a tester to tell how many faults have actually been inserted. This reporting is vitally important, as it is very easy for product code and test loads to change in subtle ways such that the targeted error handling code paths are no longer being exercised. Typically all fault and fault infrastructure code is conditionally compiled so that it only appears in debug builds and there is no risk of an accidental fault insertion in a shipping version.

At Mango, the fault infrastructure is called Faulty Towers. It was originally conceived as a software tool designed to facilitate fault insertion into the layers of protocol code towers. Faulty Towers is a database of fault objects that resides in kernel mode so that internal kernel mode driver code can benefit from its services. The fault object largely contains information on how frequently to insert the fault and on the number of insertions that have occurred. Faults are enabled at a specific frequency from a user mode command line interface that allows another developer or a tester to enable faults, disable faults and report on fault insertions. The initial development of the Faulty Towers infrastructure to support the insertion of faults took one developer only two weeks. Development of the product code to actually insert the fault was a modest additional cost but, with practice, it naturally flowed out of the development process.

The faults and fault infrastructure provide control and repeatability, which are key ingredients to a disciplined process. The problems listed previously are addressed as follows:

- *Defects in error handling code are discovered earlier in the development cycle.* Developer unit testing benefits from a rich palette of faults. In a timely and efficient manner, developers can subject code changes to all the major faults that their subsystem must handle *before* subjecting the entire organization to the changed code. Their unit testing can also tap faults from subsystems adjacent to their subsystem for an extra degree of quality assurance. This goes a long way towards avoiding chaotic integration and the accompanying schedule slips.
- *QA testing focus on error handling code paths can now be directed and prioritized.* Instead of mere "accidental" testing of error handling code paths, testers can now target different sets of error handling

code paths as they become available for testing. Error handling code path testing can now be scheduled and those of highest priority to the quality goals of the product can be targeted first.

- *QA fault testing becomes easy and cheap.* A fault that simulates a device failure is now easy to insert and at any desired frequency. There is no need for tedious manipulation of the test environment and no time consuming delays in simulating the fault. Using faults to vary internal timings in a smaller, more manageable test configuration can reveal defects normally only seen in large hard-to-test configurations. This can greatly reduce, but not eliminate, the need for expensive large scale testing. Faults are now easy and cheap to insert, providing cost-effective, repeatable testing of error handling code paths. The cost of fault testing can further be reduced by incorporating faults into test automation. Mango has combined faults with test cases in a database of over 13,000 automated tests, which greatly increases error handling code coverage.
- *Product ship decisions are better informed about the quality of error handling code paths.* The power of fault insertion to exhaustively test the most common errors under a broad range of states helps identify serious defects before products ship. Indeed, the degree of testing resources devoted to mainline code paths and error handling code paths can now be a conscious project decision. The mix varies throughout a Mango project's lifecycle as shown in the Fault Insertion Testing Data section. The insights provided by being able to control testing code coverage throughout the project lifecycle greatly inform and validate each project milestone.

This solution also provides the following additional benefits:

- *Problem reproduction is quicker.* Many problems cannot be diagnosed completely with one system crash or assert. More extensive logging may need to be added in order to see the sequence of events that lead up to the failure. Multiple crashes may also be needed by troubleshooters to isolate the common patterns underlying the failure. Relying solely on "accidental" errors, it may be impossible to reproduce a problem. With faults, a tester can immediately return to the precise level and mix of fault insertion. The ability to replay at exact fault levels usually leads to quick reproduction of failures.
- *Problem troubleshooting is easier.* With the set of faults that were active at the time of a system crash or assert clearly reported, problem isolation is normally quicker. The troubleshooter knows the overt target of the test, which errors were inserted, and what code paths to be looking at carefully. Without faults, any random "accidental" error could have occurred and lead to the crash, resulting in many more leads to run down.
- *Fix validation is more reliable.* Validation of fixes can now use faults to target the changed error handling code paths. Once a fix is in hand for the defect, the fix can be validated with the exact pattern of fault insertion that caused the initial system crash.
- *Controlling the fault's scope means less time wasted on "red herrings".* By implementing the fault, the developer exerts precise control over the fault's *scope*. If fault insertion is done in a shotgun fashion, problems may be triggered in code that is not under examination. For example, if a tester tries to restrict physical memory available to the OS in order to test "out of memory" error handling in their code, they are just as likely to encounter failures outside their code in the operating system or in other third party products. By constructing the "out of memory" fault so that it only targets allocations in the code under test, time wasted on defects in someone else's software can be minimized.

Fault insertion is especially effective if developers create a rich mix of asserts that are constantly validating the software activity. This can drastically reduce problem isolation costs. Developers should be encouraged to add debug build asserts when adding new code and especially when troubleshooting problems[4]. Since the asserts are only in the debug build, performance and reliability of the release build will not be impacted.

Risks with This Solution

Software fault insertion can also create problems especially if a fault is combined with automated tests. A test becomes a unique combination of user loads and fault frequencies. For Medley, Mango has created 209 faults and has automated much of the testing. Mango has created a huge battery of tests/fault combinations totaling over 13,000. With all of this flexibility at the tester's command, there are risks:

- *Avoiding meaningless fault chaos is essential to establishing the credibility of fault insertion.* The existence of new faults that stretch the product in various dimensions does not mean that all dimensions need to be stretched, and certainly not to the absolute limits of the dimension. Testers must deeply immerse themselves in the operation of the product and in all the nuances of customer expectation for the product. For example, a fault might exist to fake I/O errors but at what rate should the fault be inserted? What does the user expect of the product if an I/O fails once a week? Fails once a day? Fails once a minute? Fails once a second? Fails all the time? Such questions can help prioritize the important testing dimensions and set reasonable testing limits for each dimension. Another technique for setting reasonable limits on test dimensions is to have the developers who have designed the subsystem with specific tolerances set the limits on the fault as it is created. This helps make plain the limits of the product and, by providing the supported limits, encourages developers to buy into the concept of fault insertion. Good limits help the tester assure that faults are operating within valid operating limits that the software is reasonably expected to handle.
- *Test combinations become unmanageable.* With the creation of a large number of faults and test program variables, managing the nearly infinite number of possible combinations given finite test time becomes a major challenge. A single Medley test tool has 85 test load variables. If 2 settings exist for each of the 209 faults and 85 testing control variables, the number of combinations is 2^{294} . This is clearly an unmanageable number, even before acknowledging that faults and test variables seldom have just two settings. Once fault limits are agreed to, faults can usually be inserted at thousands of frequencies. For the Medley file system, test variables like the size of data files, the number of reads, the number of writes, the number of updates, the degree of sharing, and the number of sharers, each have a large number of settings. Assuring uniform coverage of a complex domain is probably the toughest aspect of a tester's job [5]. One solution is to profile the most common user operations and create tests to cover these profiles. To these key tests, the tester would then add the important faults.

Once limits on individual faults and test variables are agreed upon, the tester now faces the problem of which faults to combine with what loads. A simple strategy of only employing one fault at a time and varying both fault frequency and user loads as widely as possible is a responsible approach. The chaos is minimized and the defects in the code are likely to be easily identified. Combining multiple faults into the same test should be approached with caution. Each fault adds complexity that can push testing over the edge into extreme chaos so divorced from reality that it is seen as wasting testing and troubleshooting resources. This can be costly and non-productive.

- *Prioritizing the flood of problem reports is difficult.* Mango's experience with combining fault insertion and test automation is that prioritizing the flood of problem reports is not a simple task. Mango is at the stage where QA can control the stream of problem reports and manage it to match the organization's capacity to process them. The controls also allow us to target areas based on change history. With the ability to generate any number of problem reports, it becomes necessary to direct the testing to the most appropriate areas and to prioritize problems that are in a "must fix" category. Once again it comes down to identifying the essential user expectations for the product and establishing a categorization methodology to assure essential areas/problems are addressed first. Otherwise the QA and development response to problem reports is random and chaotic. Losing focus in a substantial stream of problem reports can cause serious schedule slips.

The Process of Fault Creation

Once the fault infrastructure is in place, the highest payback faults need to be identified and implemented. This section describes sources for faults, suggests fault areas to investigate, and discusses fault prioritization.

Fault Sources

When beginning the task of identifying promising faults, the mainline structure contained in development design specifications can be a great aid. The structure, operations and assumptions of the mainline flow in a

subsystem should be clear in a good design specification. Each of these mainline code path areas provides good opportunities for faults, as described in the next section.

Code inspections are a good vehicle for identifying faults as the whole structure of a subsystem flows before the inspection team. It is very productive to take time to identify the key interfaces and the central error handling focus which could profit from fault insertion. This also helps build team confidence in fault insertion and starts changing developer mind set about faults and error handling. Developers should be evaluating all substantial blocks of error handling code for testability, asking themselves: "How can I package this code so that a fault would allow myself and QA to test this block efficiently and rigorously?"

Another good source of faults for the tester is the collection of unit testing tools created by developers to fake important failures. These tools may be created if other pieces of the system were not ready in time or other subsystems don't inter-operate in a developer's private unit test environment. With a modest amount of effort these tools can often be made into outstanding faults. By enhancing the developer's tools and ideas, the tester continues to build credibility for fault insertion. The developer now has a stake in making fault insertion work.

The final source of faults results from a careful examination of each product defect for the root cause (referred to as causal analysis). Its goal is to learn where the system that produced the software has broken down. This technique is covered in more detail by other authors [6]. However past defects are often repeated and instituting faults to catch past and possibly future defects can be very cost-effective. The only tip offered here is that causal analysis is very hard to do thoroughly after the fact. Only at the time of the fix are all the factors known, including the history of changes, the design implications, and the changed code. It is only at this point that effective causal analysis can occur at low cost. To wait even a day is to forget context relevant to causal analysis. A good example of this practice is the NT Driver Verifier in the Windows NT operating system [7]. Through careful analysis of the common failures in kernel mode drivers, Microsoft has created a powerful tool to insert faults that isolate/eliminate these failures cost-effectively during testing. The utility allows one to insert memory exhaustion faults and a fault that invalidates page-able code and data sections in order to identify memory accessed at invalid IRQLs. Such problems are normally only seen in systems extremely tight on memory. NT Driver Verifier also demonstrates the effective use of asserts to validate operating system interfaces commonly used by drivers, doing a superb job on the memory allocation and de-allocation interfaces.

Fault Focus Areas

The following questions can help identify promising areas for fault insertion. At Mango, these areas have had the biggest payback in both user mode and kernel mode software. Some of the questions move beyond an exclusive focus on error handling code paths.

- *What are the common faults that customers expect the product to tolerate?* Any user of a personal computer experiences many failures. What are their expectations? Do they expect to never lose any data? If the machine crashes in the middle of an extended edit, do they expect to be able to retrieve most of their edit? If a media failure occurs on a disk do they expect to be able to retrieve all data except for the failed disk sectors? These types of reliability expectations need to be discussed, written down and agreed to while implementing a product, something that is rarely done.

The challenge for Medley, which is a distributed file system, is that the data for a file can exist on any computer that is part of the shared Medley drive. Thus a computer participating in the Medley drive that crashes or shuts down can impact the data access of a user on another computer. As in many distributed systems, the loss and gain of participating computers is a reality that users expect to be handled seamlessly. At Mango, the most fruitful Medley faults have been three faults that shut down, crash, or power cycle a single PC participating in the shared Medley drive¹.

¹ Interestingly, we found that crashing a PC and power cycling a PC were not quite the same fault. We speculate that the power cycle was more abrupt and less synchronized with other PCs.

Other common Medley failures involved persistent or transitory device errors. Since the product is distributed, network device failures were highly relevant. Disks can also have media failures that cause a read I/O or write I/O to fail. And since this is a file system, it was important to handle the ever-present disk full condition gracefully.

- *What are the key widely-used interfaces?* The more widely-used the interface, the more error handling code is associated with it. By exercising the error returns of such interfaces, the fault will achieve the widest possible coverage. Of course, each widely-used interface must be evaluated for the likelihood that errors will occur. Since Medley is a distributed product, most functions have to perform network I/O to synchronize and communicate with other PCs involved in the Medley Drive. Thus the Network I/O interface is a good candidate as the interface is subject to frequent failures and is widely-used throughout the product. Other widely-used interfaces are those used for locking and for memory allocation and de-allocation.

As errors are inserted into widely-used interfaces, the likelihood increases that exception handlers will be exercised. Exception handling in a language like C++ has all the risks associated with an infrequently executed GOTO. Exception handlers must be able to sense, rollback and cleanup all the possible state in code that throws an error. In a large routine with many throws, it can be hard to keep the context from all throws straight. Additionally, it is easy to overlook code in the disconnected exception handler when making bug fixes to the body of the routine. Since exception handlers are rarely executed and involve an interrupted flow of execution, they are fraught with potential defects. The ideal of being able to exercise every exception handler is normally hard to realize. However the pervasiveness of memory allocation makes memory allocation failures among the most universal of all faults, ideal for getting at such error handling code paths.

- *What assumptions are made in the design?* If a design uses a strict lock acquisition ordering to avoid deadlock then be sure that faults check that the order is never violated. Waiting for a deadlock between two locks to occur is an expensive way to remove deadlocks from the product. Deadlocks are better detected by checking on the acquisition of a new lock that no others locks are currently held which would violate the locking order. Detecting the ordering violations before the deadlock occurs is a cheaper way to eliminate the threat of deadlocks.

Another example of a fault testable design assumption would be a queue which is unordered. If the design states that objects can be placed on a queue in any random order, then create a fault to verify this assumption. Maximize randomness with a fault that always reorders objects on the queue before removing an object from the queue. Stating the idea more abstractly, the randomness of reality is the enemy; seek to make it your friend. Acknowledge that tests artificially order the product's code into pseudo-random sequences. When the code ships, the richer mix of real customer loads and events may expose problems as code executes in different sequences. Whenever some entity is known to be random, always ask: Is the randomness of the test environments sufficient? What is the payback of adding a fault to minimize or maximize the randomness to better simulate the random order of the real world?

Other assumptions may be unstated characteristics of certain hardware or interfaces. For example, any network protocol has to deal with three important contingencies: a given message can be re-sent, reordered or discarded. How does the product cope with this reality? Creating faults to randomly reorder messages, re-send messages, or drop messages is an ideal way to assess how a distributed product handles these fundamental network failures. Discussion with developers who are experts on a device or interface may be the best way for a tester to isolate these assumptions.

- *What are the error retry policies?* Faking a single error from an interface in a fault can be defeated if all callers simply retry the call five times before giving up. The error will not percolate up through all the layers, perhaps all the way to the user, as was intended by the tester. Understanding a product's retry policies is very important to effective fault insertion. In general, all retry mechanisms should be

under the control of a fault. Being able to disable all retries allows single fault insertions to have their most dramatic impact on error handling code paths.

- *What timeouts or delays exist?* Every complex product will have some timeouts or delays built into them. Medley, for example, places a timeout on requests that are sent to a remote computer in the Medley Drive before declaring the remote computer to have failed. Since different network requests can take different times to complete, this timeout is really an arbitrary boundary with regard to the processing of the request. The request's response might arrive just before the timeout. Will the sending computer handle this correctly? What if the response arrives just after the timeout? What happens on the sending computer? Is everything cleaned up correctly? Will the sender try to re-send the message? Will this confuse the remote computer? By creating faults to manipulate these arbitrary fixed timeouts and delays that are placed in code, these values can be varied, achieving a higher level of robustness in the algorithms.
- *What code is under-executed?* Code that is rarely executed will contain defects. Such code receives only the most cursory of testing by nature and yet the defects may be so catastrophic as to force a company to immediately generate a new release once the code is in the field. The astute tester will constantly be on the lookout for large blocks of code that are under-executed. A tester might be able to spot such code merely from design specifications. Rare and esoteric states in a design could hide defects. Faults that force these states to occur at higher frequencies allow the tester to vary the execution frequency to assure higher reliability. Code coverage tools can be of some assistance, but it is hard to obtain code coverage for kernel mode code. A close look at problem reports filed on a product can often reveal code that is under-executed and that might be an excellent target for fault insertion.

The startup and shutdown paths in a product often fall into this category. Typically they are executed only once at system startup and shutdown. Faults to introduce errors in the startup and shutdown sequences can be fruitful. Tests that loop repeatedly starting up and shutting down the product are also a big assist. In an operating system, process creation and process deletion are good examples of under-executed functions that are not relatively common and thereby typically not as robust as more mainline code paths. For a binary-tree implementation the very complex operations of splitting and merging data buckets are normally among the most complex operations and yet they may occur infrequently. These operations involve simultaneous locks on multiple buckets and shuffling data between two or three buckets. The splits and merges can propagate all the way up to the root of the tree. Creating faults to force these bucket splits and merges to occur at a higher frequency, even though a bucket doesn't really need this maintenance operation, allows targeting of these complex under-executed operations.

- *What controls the frequency of background operations?* Many complex software products boost mainline performance by off-loading large non-essential functions onto background threads. However, these background threads or daemons can hide a lot of defects. Because background daemons don't run at a high frequency, conflicts with the mainline operations are not readily observable. If the background daemon is driven by a work queue, what happens if there are huge delays in processing transactions? What if a single big delay stops all processing and then the huge backlog of work items on the work queue is processed in a frenzy? Creating faults that control how frequently background daemons are requested to run, or that place stalls around the processing of the work queue, are effective tools to assure that background daemons are robust with regard to foreground mainline operation. Any timer-based routine that runs on a regular interval can fall into this category.
- *What optimizations are frequently used?* An optimization in the code that is used at a high frequency can hide the fact that the non-optimized code path is not robust under all circumstances. Medley has an in-memory cache of data to reduce the number of I/Os required from local disk or from a remote system for metadata or data. While this is a vital ingredient to superb user performance, it means that requests to the disk subsystem are rare. Does the complex disk subsystem perform flawlessly in the face of PCs leaving and joining the Medley drive? Creating a fault that drops data from the in-memory cache when the user is finished accessing it allows Mango to effectively deactivate the cache. This puts extra stress on the disk subsystem and allows it to achieve a new level of reliability. Implementing a

fault for every optimization in the system, no matter how trivial, may not be practical. But a developer should at least consider the payback for creating a fault to deactivate every optimization that is added.

- *What are the synchronization primitives?* The locking primitives tend to order the execution of code into set patterns. Only when the patterns are stretched to their limits do problems become visible. If locks are held unusually long, this can perturb the execution order of others threads. In addition, there can be tiny windows where context is not properly synchronized via locks. In order to detect these windows and reorder external execution, create faults to insert stalls within the locking primitives. Stalling for a few milliseconds before taking the lock gives other threads ample opportunity to modify the stalled thread's data structure context before the lock is taken. Stalling to open wide similar windows after taking the lock but before returning to the caller, and on the release of the lock, are powerful fault insertion techniques.

One of the most vulnerable areas in any system is thread stall and resumption. Unless very disciplined, it becomes easy to stall holding locks that should have been released or to restore locks and context incorrectly when the thread is resumed. A big advantage of faults that stall within locking primitives is that this forces other threads to stall and resume more frequently than normal, thus improving the coverage of these sensitive thread stall and resume operations. Trying to identify and exercise all thread stalls and resumes is a worthwhile goal.

- *What resource exhaustion and reclamation conditions exist?* Resource exhaustion is another important failure. What happens if there is no free memory or no free disk space? At Mango, faults exist to fake resource exhaustion conditions such as out of memory errors and disk full errors. However the notion of a persistent fault is useful here. Normally, when a fault is inserted, a single error is returned. But in reality, memory exhaustion states or a disk full states tend to persist for a sustained period of time. A feature of Faulty Towers is the ability to create *persistent faults* that will fire multiple times in a row. When memory is to appear exhausted, the fault fires for the next N memory allocations in a row. When a disk full error is returned, the next N disk space allocations will fail. Persistence can also exist on a resource by resource basis. If a disk sector goes bad due to a media failure, then in order to simulate disk device reality, the sector should remain bad. An error should be returned on all reads until the sector is written again. In this case, fault specific storage allows tracking of the addresses of the disk sectors that have been declared bad so that media failures can be simulated consistently.

Associated with Medley memory pools is a memory pool reclamation mechanism. When memory is running low on the system, Medley triggers callbacks that force subsystems to return unused memory in their pools. Forcing maximum reclamation of idle data structures is an excellent means of stressing memory resource handling and assuring that access to data structures is fully synchronized in a subsystem. Note that while only two resources have been mentioned, these techniques could apply to any resource.

- *What "accelerated life" testing opportunities exist?* It is not widely appreciated that, like manufactured machines, data structures age. As persistent data structures stored on disk are used continuously over a number of years they are stretched in different directions which are not always robustly handled by software. For example, Medley has 128-bit addressing objects. While new systems operate fine within the first few gigabytes of the 128-bit addressing scheme, what happens when operating in the middle of the range? What happens when operating in the last gigabyte of this range? It can take months, years, or decades of testing to achieve this aging via the normal file creation/deletion paths. However, with a fault, one can arbitrarily override the next free address in the object's range. The next addressing object created will use this address. Thus one can easily test operation of this object over its entire range. Another opportunity lies in Medley continuation data structures that only exist if many, many files are added to the same directory on a drive. To simulate this condition, a fault exists to control the number of files that triggers the creation of these continuation data structures. Thus these continuation data structures which were once quite rare are now common and easy to test.

Fault Prioritization

Of course not every problem can or should be targeted by a fault. If, to simulate an esoteric failure, one would have to double the size of the product's code, then this is clearly unfeasible and unwarranted. The fault code itself would be so complex as to introduce errors of its own. A tough cost/benefit analysis has to be done on each fault idea. What is the impact of the failure on the user? Does the failure result in data corruption or is it undetectable to the user? What is the likelihood of the failure occurring? If very infrequent, it is harder to justify enormous effort in creating and maintaining the fault. How maintainable is the proposed fault? If the fault is spread all over the system instead of being isolated to one place then long-term maintenance becomes more costly. If the fault requires a substantial amount of code be implemented then the payback of problems it finds had better be worth the investment.

In general it is good for QA to write down the error recovery priorities for a product and those QA intends to target in testing so that all can see these priorities and validate that they are in line with the quality goals of the product. Obtaining buy-in from developers, managers and the salesforce on these priorities can smooth the introduction of fault insertion and short circuit the contention that inevitably arises when problem reports from fault insertion start to pour in. Unless the path is carefully prepared, QA may face the charge that the problem reports were produced by some chaotic, invalid methodology that bears no relation to actual product usage.

Summary

This paper has presented compelling data on the benefits of software fault insertion. A modest investment has reaped huge benefits. While data was presented for only one product, software fault insertion has been used effectively on several software products. This paper outlined a process where one person can identify and target a small number of their most significant faults and build from there. Tips were provided at each step to build the credibility of fault insertion, starting down the road towards full cultural acceptance of fault insertion.

Software fault insertion is most useful if adopted by the culture of the entire organization. If requirement documents and design documents clearly spell out the faults the product is expected to survive, and if developers are aggressively creating and testing with faults, then the tester's job of merely incorporating the faults into the test structure is easy. When an organization's culture has achieved this level of acceptance of fault insertion, then the level of product quality begins to rise dramatically.

Acknowledgements

This work would not have been possible without the superb team of engineers in QA and development at Mango. Special thanks to Bill Goleman, Ed Smith, Jody Zolli, Dan Way and Don Gaubatz for their careful feedback on this paper.

References

- [1] Jeffrey Voas and Gary McGraw, Software Fault Injection, Wiley Computer Publishing
- [2] Ed Smith, Continuous Testing, Proceedings of TCS2000 Conference, June 2000, <http://www.mangosoft.com/technology/>
- [3] W. Humphrey, Personal Software Process, Addison Wesley, page 275
- [4] Jeffrey Voas and Lora Kassab, Using Assertions to Make Untestable Software More Testable, Software Quality Professional, Vol 1 Issue 4, page 31
- [5] William L. Goleman, Robert G. Thomson, Paul J. Houlihan, Improving Process to Increase Productivity While Assuring Quality: A Case Study of the Volume Shadowing Port to OpenVMS AXP, Digital Technical Journal, Vol 6 No 1, page 36
- [6] Gerald Weinberg, Quality Software Management, Dorset House Publishing, page 188
- [7] Jim Finnegan, Nerditorium -- Microsoft Systems Journal, February 2000, <http://www.microsoft.com/msj/0200/nerd/nerd0200.asp>
- [8] Ed Smith, Automated Results Processing, Proceeding of STAREAST 2001, May 2001, <http://www.mangosoft.com/technology/>

Paul Houlihan

Paul Houlihan has been studying for several years how to remove defects cost-effectively from code, especially for distributed algorithms. For the past 5 years, Paul has worked at Mangosoft Incorporated as a principal engineer in the QA group. Mangosoft specializes in peer-to-peer software running in kernel as well as user mode.

Mangosoft's has two premier products. The first is CacheLink™, which is a LAN based web accelerator that caches web pages and makes them available to other users on the LAN. The other is Mangomind™, a shared Internet drive that appears as a local drive on your Windows desktop. With extensive caching for good performance, Mangomind makes collaboration over the Internet as simple as accessing a local drive. While it has been a challenge to automate the testing of these diverse and complicated products, very high quality is essential for distributed products, indeed, it is a key differentiator. Paul has contributed to the extensive test automation infrastructure that in one year alone ran 56,000 tests in Mangosoft's automated unattended 24-by-7 lab, filing 3686 problem reports.

Prior to that Paul was a principal engineer in the OpenVMS cluster group. Layered on the OpenVMS operating system, VMSClusters are a distributed work environment spread over up to 96 computers, spanning multiple interconnects and computer and disk architectures.