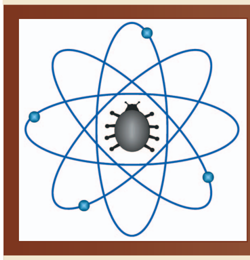
Fighting Bugs: Remove, Retry, Replicate, and Rejuvenate

Michael Grottke and Kishor S. Trivedi
Duke University



Combatting vastly different types of software bugs requires different strategies.

During the Gulf War, 28 US Army reservists were killed and 97 were injured on 25 February 1991 when the Patriot missile-defense system at their barracks in Dhahran, Saudi Arabia, failed to intercept an incoming Scud missile.

This well-known incident occurred due to a software fault, or bug, in the system's weapons-control computer. This event led to a *system failure*—that is, a deviation of the actual system behavior from correct service. It was also a case in which engineers employed multiple techniques to fight the software bug.

BOHRBUGS: REMOVE

Finding and removing software faults is the classic strategy for dealing with them. Fixing bugs in the operational phase is considerably more expensive than doing so in the development or testing phase. Published literature reports cost-escalation factors ranging from 5:1 to 100:1 (B.W. Boehm and V.R. Basili, "Software Defect Reduction Top 10 List," *Computer*, Jan. 2001, pp. 135-137).

Therefore, engineers expend much effort on detecting and removing bugs during software development via both dynamic software tests and static techniques like code reviews and walkthroughs. Systematically conducted unit and system tests play an important role in revealing faults that lead to failures during software execution.

However, diagnosing and isolating the underlying fault responsible for an observed failure becomes difficult if the failure can't be reproduced. Software testing is, therefore, mainly suitable for dealing with faults that consistently manifest under well-defined conditions. Testers sometimes refer to such faults as Bohrbugs, an allusion to Niels Bohr's simple and intelligible atomic model (J. Gray, "Why Do Computers Stop and What Can Be Done About It?" *Proc. 5th Symp. Reliability in Distributed Systems*, 1986, pp. 3-12).

MANDELBUGS: RETRY, REPLICATE

However, testers do encounter failures they can't reproduce. Under seemingly exact conditions, the actions that a test case specifies can sometimes, but not always, lead to a failure.

Fault activation

To explain this phenomenon, it's useful to take a closer look at how the static fault in the software is connected to the dynamic failure occurrence.

Usually, activating a fault by executing the part of the software where it's located doesn't immediately cause a failure. Rather, it produces an internal condition in the system that deviates from the correct internal condition—referred to as an *error*—even though the user might not perceive this discrepancy. An error can develop into further errors before a failure finally occurs. This functional chain between errors and failure is called *error propagation*.

For example, a fault in an algorithm's implementation can lead to an erroneous computation for specific values of a program variable—a case of fault activation causing an error. The software can use this incorrect result internally for further calculations, in which case the error propagation leads to additional errors. A failure occurs only when the system uses one of these incorrect calculations in a way that influences a perceivable system behavior, or when error propagation causes a failure occurrence.

Based on the relationships between faults, errors, and failures, we can offer two explanations as to why software may behave differently under apparently identical conditions. First, if there's a long delay between the fault activation and the final failure occurrence—for example, traversing several different error states in the error propagation—then it's difficult to identify the user actions that actually activated the fault and caused the failure. Simply repeating the steps carried out a short time before the failure occurrence might not lead to its reproduction.

Second, other elements of the software system—such as the operating system, other applications, or the hardware—can influence a fault's behavior in a specific application. We refer to the set of these elements as the application's *system-internal environment*. For example, inadequate synchronization in multithreaded software can give rise to race conditions, in which the program behavior depends on the relative timing of the threads the operating system schedules. Since a failure only occurs if the operating system schedules the threads in a specific order that the programmers didn't foresee, troubleshooters find it difficult to reproduce such failures and isolate the underlying faults.

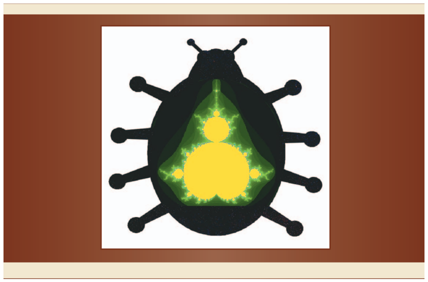
A fault can cause the software to exhibit a chaotic and even nondeterministic behavior with respect to the occurrence and nonoccurrence of failures if its activation or error propagation are complex in at least one of these two ways. Software engineers sometimes refer to faults with this property as Mandelbugs, an allusion to Benoît Mandelbrot, a leading researcher in fractal geometry (E.S. Raymond, *The New Hacker's Dictionary*, MIT Press, 1991).

Sometimes, the literature also calls these software faults Heisenbugs. However, Bruce Lindsay, who invented the term, derived it from Heisenberg's Uncertainty Principle, referring to faults that change their behavior when probed or isolated (M. Winslett, "Bruce Lindsay Speaks Out," *ACM SIGMOD Record*, June 2005, pp. 71-79). Since the system-internal environment induces the change in behavior, Lindsay's Heisenbugs are actually a type of Mandelbug.

Unique problems and possibilities

A problem with Mandelbugs is the high probability that a developer won't detect them during testing. Even if a programmer or tester were to execute the parts of the code containing Mandelbugs, they will only cause failures if they meet the complicated conditions related to the system-internal environment. Mandelbugs can therefore go unnoticed until long after the software's release.

Moreover, even if a Mandelbug should cause test cases to fail, reproducing the failures is difficult, as is isolating the underlying Mandelbug. As a consequence, removing the Mandelbug before the software's release might not be possible. Therefore, it's plausible to assume that the majority of the faults remaining in a well-tested piece of software are Mandelbugs. However, the published data is inconclusive, indicating that Mandelbugs account for between 15 and 80 percent of all software faults detected after release (S. Chandra and P.M. Chen, "Whither Generic Recovery from Application Faults? A Fault Study using Open-Source Software," *Proc. Int'l Conf. Dependable Systems and Networks*, IEEE CS Press, 2000, pp. 97-106; I. Lee and R.K. Iyer, "Software Dependability in the Tandem GUARDIAN System," *IEEE Trans. Software Engineering*, May 1995, pp. 455-467).



On the other hand, Mandelbugs' seemingly nondeterministic behavior makes it possible to deal with them in ways infeasible for dealing with Bohrbugs.

First, when a Mandelbug has caused a failure, a simple retry of the failed action can result in success. This explains the phenomenon that restarting an application or rebooting the system after a crash often solves the problem. We can improve the approach by combining it with checkpointing, a technique that involves regularly saving a snapshot of the application state in stable storage. After a failure, we can restart the application to the latest available snapshot.

Second, adopting replication, that is, using redundant resources—an approach from the hardware reliability field—is possible. Since natural phenomena like physical deterioration cause most hardware faults, failover to an identical component upon failure of the first usually won't lead to a second failure. However, software faults are human-made design errors that lurk in the software code.

Different installations of the same operating system running the same applications should contain the same faults. Scholars, therefore, have questioned whether software replication can offer benefits similar to those for replicating hardware. After all, when executing the commands on a second installation of an identical piece of software, a user would encounter the same fault that led to failure the first time, causing another failure.

Of course, this reasoning implicitly assumes that all faults are Bohrbugs. However, since many software faults are in fact Mandelbugs not manifesting consistently under well-defined conditions, software replication has indeed proven useful. It plays a key role, for example, in the Object Management Group's fault-tolerant CORBA standard.

AGING-RELATED BUGS: REJUVENATE

In recent years, researchers have studied yet another approach to handling software faults. Anecdotal evidence suggests that restarting a program or rebooting a computer *before* the user experiences a failure can be beneficial for avoiding future failure occurrences.

Such proactive measures only make sense if the failure-occurrence rate increases with the runtime; if the rate were constant, then restarting or rebooting wouldn't affect the risk of experiencing a failure. In fact, software systems running continuously for a long time tend to show a degraded performance and an increased failure-occurrence rate, a phenomenon called *software aging*. Consequently, the preventive counter techniques are referred to as *software rejuvenation* (Y. Huang et al., "Software Rejuvenation: Analysis, Module and Applications," *Proc. 25th Int'l Symp. Fault-Tolerant Computing*, IEEE CS Press, 1995, pp. 381-390).

Intuitively, the software-aging phenomenon appears impossible as we are executing software without introducing any changes into its code. Why would the failure-occurrence rate change over time if we don't modify the software code? There are two possible solutions to this puzzle.

First, the aging-related bugs can cause errors to accumulate over time. These error conditions can accrue either within the running application, such as round-off errors in program variables, or in the system-internal environment, such as unreleased physical memory due to memory leaks in the application. In either case, the error conditions don't lead to failures immediately. Otherwise, there would be no aging. The failures occur with a delay.

Second, the total time that the system runs continuously can influence an aging-related bug's activation rate. We can consider this runtime an aspect of the system's internal environment.

Obviously, both types of aging-related bugs are Mandelbugs.

Example: Patriot system

The software fault in the Patriot missile-defense system responsible for the Scud incident in Dhahran was the second type of aging-related bug. To project a target's trajectory, the weapons-control computer required its velocity and the time as real values. However, the system kept time internally as an integer, counting tenths of seconds and storing them in a 24-bit register. The necessary conversion into a real value caused imprecisions in the calculated range where a detected target was expected next.

For a given velocity of the target, these inaccuracies were proportional to the length of time that the system had been continuously running. As a consequence, the risk of failing to track, classify, and intercept an incoming Scud missile increased with the length of time that the Patriot missile-defense system operated without a reboot.

On 21 February 1991, the Patriot Project Office warned Patriot users that "very long runtimes" could negatively affect the system's targeting, implying it should be rebooted regularly. Unfortunately, the Army officials assumed that the users would not continuously operate the Patriot systems long enough for a failure to become imminent; therefore, they did not specify the required rejuvenation frequency.

Costs

While rejuvenation can clean internal error conditions from a system and avoid failure occurrence, it does incur costs.

For example, during a Web server's reboot, the hosted Web site might be unavailable, or if multiple servers provide service at the same site, the running servers must share the load.

In transactions-based software systems, initiating rejuvenation can lose jobs that the system currently serves. The Patriot missile-defense system reboot, which also reset the internal clock to zero, took about 60 to 90 seconds; during this time, the system could not react to incoming missiles.

Rejuvenation, therefore, requires optimal timing. The two main approaches are based on models and measurements.

Model-based approaches use analytic models to capture system degradation and rejuvenation. Under a given rejuvenation policy—such as "Rejuvenate an idle server if at least x hours have passed since the last rejuvenation"—operators can then use the model for determining the optimal time interval x and the dependability measures following from this policy.

The main idea behind measurement-based approaches is to periodically monitor system attributes that might show signs of software aging. For example, a continuous increase in the amount of used physical memory might suggest the existence of memory leaks that would ultimately lead to a system crash. Online-monitoring systems can use the collected data to assess the system's current health and predict aging-related failures (M. Grottke et al., "Analysis of Software Aging in a Web Server," *IEEE Trans. Reliability*, Sept. 2006, pp. 411-420).

Even if software developers don't fully understand the faults or know their location in the code, software rejuvenation can help avoid failures in the presence of aging-related bugs. This is good news because reproducing and isolating an aging-related bug can be quite involved, similar to other Mandelbugs.

Moreover, monitoring for signs of software aging can even help detect software faults that were missed during the development and testing phases. If, on the other hand, a developer can detect a specific aging-related bug in the code, fixing it and distributing a software update might be worthwhile. In the case of the Patriot missile-defense system, a modified version of the software was indeed prepared and deployed to users. It arrived at Dhahran on 26 February 1991—a day after the fatal incident.

Acknowledgments

This work was supported by a fellowship within the Postdoc Program of the German Academic Exchange Service.

Michael Grottke *is an assistant research professor in the Department of Electrical and Computer Engineering at Duke University. Contact him at Michael.Grottke@duke.edu.*

Kishor S. Trivedi *holds the Hudson Chair in the Department of Electrical and Computer Engineering at Duke University. Contact him at kst@ee.duke.edu.*

Editor: Michael G. Hinchey, NASA Software Engineering Laboratory at NASA Goddard Space Flight Center and Loyola College in Maryland; michael.g.hinchey@nasa.gov