Testing z/OS: The premier operating system for IBM's zSeries server

by S. Loveland
G. Miller
R. Prewitt
M. Shannon

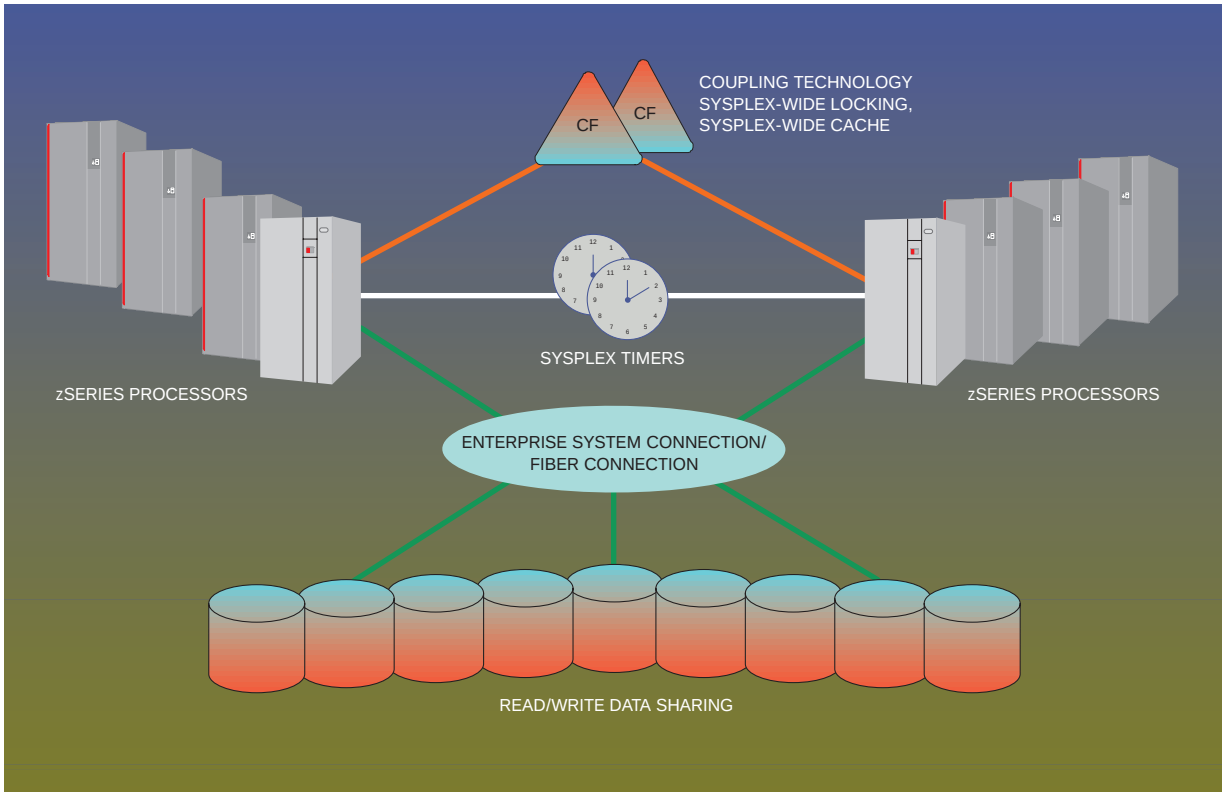
The "z" in zSeries™ stands for zero down time. As businesses have come to rely more and more on the continuous availability of their largest systems, the verification techniques used by IBM in developing those systems have had to evolve. Methodologies, techniques, and tools need continuous enhancements to develop the necessary verification processes that support development for a "zero down time" system. This paper describes the verification methodologies used in z/OS™ development, as well as test technologies and techniques. Special attention is paid to tool and test case reuse, and to techniques for testing for data integrity and system recovery. We also explain how these methodologies can be used for both traditional on-line transaction processing and newer Web-based or distributed applications.

What is z/OS*? It is the premier operating system that powers IBM's zSeries* processors. It is a general-purpose operating system that many businesses rely on. Commonly referred to as "mainframes," the zSeries processors (and their predecessors) have been the backbone of commercial computing for decades, renowned for their reliability, scalability, availability, and other industrial-strength attributes. But why the distinction between mainframes and other types of servers? Historically, some platforms were designed for scientific and technical computing, with massive, extended "number crunching" on relatively small amounts of data and relatively low interactions between the processor and external storage. Other platforms were designed for a commercial environment, with exactly the opposite characteristics—constant movement of data between the processor and external storage, very large amounts of data, and processor resources consumed in short bursts for each transaction or read/write operation.[1] The IBM zSeries server and its z/OS operating system were designed for just this kind of commercial environment.

What are some characteristics of z/OS that allow it to support these commercial requirements? One characteristic is usable capacity. E-business applications have dramatic swings in user activity, with orders of magnitude changes occurring in seconds. In order to survive these peaks, many platforms are typically over-configured to run with a peak CPU utilization of 50–60 percent, with average CPU utilization of 20–30 percent. zSeries systems, on the other hand, can automatically and continuously reallocate system resources (processors, memory, and I/O) in response to changes in demand. This allows z/OS to run at a peak of 100 percent, and an average of 65–75 percent—although it is fully capable of running at 100 percent CPU utilization 24 hours a day.[1] In addition, zSeries systems can move the large amounts of data required in a typically data-intensive e-business application among processors, memory, and I/O at a rate of up to 25 times that of other high-end servers.[2] Its usable capacity allows z/OS to effortlessly handle workload spikes and enables customers to squeeze maximum utility out of their computing investment while keeping their end users satisfied.

©Copyright 2002 by International Business Machines Corporation. Copying in printed form for private use is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the *Journal* reference and IBM copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free without further permission by computer-based and other information-service systems. Permission to *republish* any other portion of this paper must be obtained from the Editor.

Figure 1 Parallel Sysplex elements



Another characteristic is concurrency. A zSeries server is a tightly coupled, multiprocessor (MP) system, where all processors are interrupt-driven and share a common memory store. Furthermore, up to 32 such servers can be clustered together into a loosely coupled Parallel Sysplex*, utilizing a shared storage medium known as a coupling facility (licensed internal code running in a special type of logical partition in certain ES/9000*, S/390*, and zSeries processors[3]), as shown in Figure 1. A fully configured Parallel Sysplex could contain as many as 640 processors operating in concert to achieve multiple, concurrent business objectives. Another aspect of z/OS concurrency is robust support for multiple, disparate workloads running concurrently on the same server image, or within the same sysplex cluster. This approach stands in stark contrast to servers with a scientific computing heritage, in which a given server is typically dedicated to one and only one task. The additional concurrency allows z/OS customers to consolidate multiple applications on a single server, reducing systems management costs while maximizing a given server's exploitation.

Many z/OS customers have business requirements for continuous system availability. System down time or unplanned outages, even of short duration, can cost millions of dollars in lost revenue or other significant negative business impact. Thus z/OS customers do not think about availability in terms of minimizing the time of an outage; they think in terms of minimizing outages—period. Again, this is historical. From the beginning, for z/OS as well as its predecessors, a basic assumption has been that hundreds or even thousands of users would depend on it. That is why its error detection and correction systems are so deeply ingrained.[4] In fact, the anticipated mean time between failures of IBM z900 systems approaches 30 years.[2]

Although z/OS itself has a long history, so do its users' suites of applications. It is not unusual for a z/OS

customer to have millions of dollars invested in business-critical applications that have been operating and evolving on z/OS and its predecessors for 20 years or more. Ensuring that this customer investment is preserved requires each z/OS release to maintain compatibility with prior releases. For some releases this is easy; for others, such as the change from 24-bit to 31-bit addressing, and again from 31-bit to 64-bit addressing, it presents challenges. But regardless of the complexities involved, application compatibility is expected for every z/OS release.

Finally, an operating system must maintain data integrity—a system that is up but quietly corrupting data is worse than one that is down. z/OS address spaces separate applications from each other to minimize the risk of one program corrupting another program's private storage or data area. Storage-protect keys prevent user programs from altering system storage. Extensive system locking and serialization techniques coordinate system events and actions. Data integrity is a core attribute of z/OS.

Testing challenges

The very attributes that distinguish z/OS as a premier commercial operating system create very real challenges with respect to its testing. Wide-ranging application compatibility from release to release must be demonstrated; capacity and concurrency must be validated across large system configurations driven at or near 100 percent utilization for extended periods; failure isolation, recovery services, and overall system availability in the face of harsh conditions must be verified; and the maintenance of data integrity must be confirmed. At the same time, customer demands for these qualities of service are colliding with business requirements demanding speedier "time to market." Such a conflict drives a need to maximize test efficiency without sacrificing test coverage.

Most research on software testing focuses on formal methods for identifying coding errors in a program. While the continuation of this research and the improvement of the tools and practices that result from this work are important, it is equally important for testers to understand other approaches for software verification. In practice, methodologies must be followed that focus different test phases on the defects they are most suited to extracting, direct testing efforts toward the system attributes of most concern, and optimize testing resources by emphasizing the value of reuse. Test cases must be written with automation in mind. Tools must allow high-volume test execution with minimal intervention, maximize a test case developer's efficiency in complex environments, and adapt to allow expansion and reuse. Sometimes this requires introducing special "testability" features into z/OS itself. Finally, special attacks need to be targeted against critical, yet tricky areas such as system recovery and data integrity. Above all, the test team must ensure that within the set of defects they find is the subset of defects most likely to disrupt real production environments; in other words, they must find the defects that matter.

Addressing such realities is a challenge for any software development team. This paper describes some of the actual methodologies, tools, and experiences of the z/OS software test team at the IBM z/OS development laboratory in Poughkeepsie, New York, in meeting the needs of both our business and its customers.

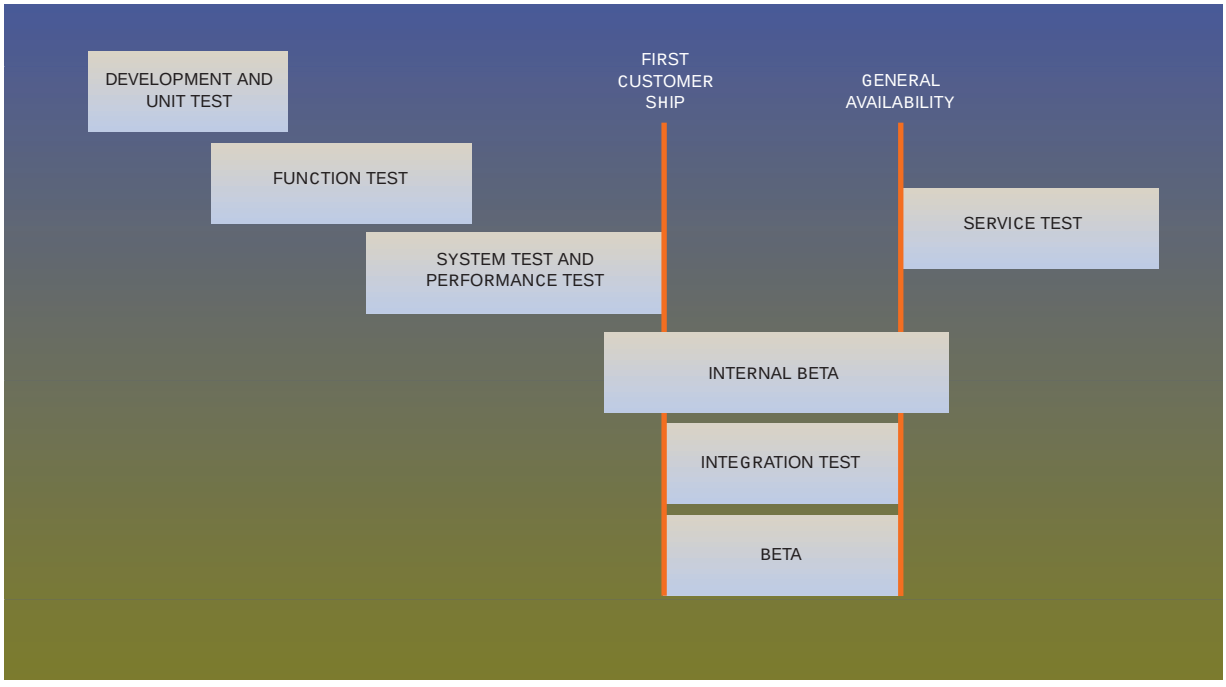
Test methodology and process

This section gives an overview of the z/OS development laboratory's testing methodology, with a focus on system and integration testing (see Figure 2).

Process definition. At a high level, the methodology used for testing z/OS is similar to that of most complex software projects, with testing broken down into phases following the classical waterfall model:

1. *Unit verification test* (UVT). The developer of an individual module verifies its basic operation, including all possible branches, loop terminations, etc.
2. *Function verification test* (FVT). A separate test team validates the new features of an entire function or component (such as the real storage manager, I/O subsystem, etc.), including its complete operation, internal and external interfaces, limit conditions, messages, and so on.
3. *System verification test* (SVT). Another team tests the entire operating system kernel, with all components working together to support a large number of clients.
4. *Performance verification test* (PVT). The performance characteristics of the operating system are measured.
5. *Integration test* (IT). All elements of the operating system are tested in conjunction with resources managers, networking products, and applications as part of an end-to-end solution.

Figure 2 z/OS development cycle



Experiences are published externally for customer reference.

The UVT and FVT phases are executed in a single-user, emulation environment where break points and trace paths can be used to facilitate testing. SVT, PVT, and IT phases are typically executed in large, native hardware environments where the system can be pushed to its limits. Different classes of bugs are targeted at different phases, based upon where it is most efficient to catch them. For example, attacks are devised during FVT to find problems with user interfaces and mainline operation, whereas timing and serialization issues are a prime target during SVT. In IT, the focus is not necessarily on code bugs, but on validating the workings of the entire platform. Are the systems manageable? What tasks and setup are needed to maintain continuous operations? Is the product documentation accurate and complete? How best can new function be implemented without requiring a system outage? One of the outputs of the IT team is a document of the team's hints and tips, suggestions for customization settings, and other relevant experiences.[5]

The role of a tester. Good testers enjoy breaking things, especially other developers' software. They delight in identifying likely vulnerabilities and devising attacks to expose them, including generalized categories of attacks that can be applied to a wide range of software products.[6] This is unlike most other software professionals, who view defects as annoyances to be avoided, "bumps in the road" toward their goal of seeing things work. Having this "breaker mentality" is a crucial trait for those testing software developed to support continuous operations.

But the role of a tester goes beyond simply trying to break things. At a fundamental level, a z/OS tester acts as an advocate, looking out for the well-being of customers. This means more than proving that the software under test does what the specifications state. It also means ensuring that the specifications and design of the software will not cause problems in customer environments. For example, a function may work exactly as specified, but not provide an adequate external user interface. A good function tester will identify this and work directly with developers to address the issue, sometimes advocating a design change. System testers must create environments and

use tools and applications that closely mirror the way customers will utilize the functions. In fact, z/OS developers consider the system test team to be their first customer.

Test plan. The foundation for any test begins with a solid test plan, where attacks targeted against new features of the operating system are devised and documented. From FVT onward, the plan is typically a formal document. Input comes from several sources, including product specifications, "postmortems" from prior tests of the same or similar components (where testers reflected upon and documented the strengths and weaknesses of their approach), and test approach reviews (TARs). The TAR is a formal meeting between the FVT, SVT, and development personnel responsible for each new feature to be tested. Planned test scenarios are reviewed, and gaps or overlaps are identified and removed.

SVT phases. Once the test plan has been approved and appropriate entry criteria met, testing commences. Each phase has its own natural flow, but in this paper we focus on SVT. This phase typically starts with regression and migration testing, moves on to basic load and stress testing, then into new function (including scenario-driven) testing, and finally recovery testing. Each is discussed in the sections that follow.

Regression, migration, and load and stress testing. In regression testing, the objective is to ensure new features in the product have not broken, or "regressed," previously existing support or application compatibility. A related objective in migration testing is to validate that a new version can interoperate with older versions of the operating system within the same sysplex cluster. In both cases, automation tools are used to execute a collection of batch and interactive test cases, initially in an essentially single-threaded environment at low system-stress levels, and progressing up to a full load and stress test, with a high degree of concurrency and CPU utilization levels exceeding 95 percent for extended periods of time. Individual test cases target a specific function or set of functions within the operating system, from the allocation and deletion of files, to global resource serialization, to reservation and updating of system storage areas. Individually, each test case exercises only a tiny slice of the operating system code. Taken together, they drive a broad range of functions that span many operating system services.

Where do the regression test cases come from? z/OS regression test cases have been accumulating for over 20 years, with no end in sight. All originated as vehicles for testing new function in past releases; once that role was satisfied, they were saved in "regression test buckets." But in order for these test cases to execute in a fully automated manner, they must be designed to meet certain standards. Specifically, each test case must be:

Good testers enjoy breaking things, especially other developers' software.

- *Self-checking*. It must programmatically verify that the functions it exercises have performed correctly; if they have not, an error return code must be set or an abnormal termination triggered.
- *Debuggable*. When a failure is detected, the test case must externalize as many details as possible about its current status, so that when viewed hours later by a human, the source of the failure can be diagnosed.
- *Well-behaved*. It must not alter control blocks or other resources owned by system components. The test case must also be able to coexist with other completely unrelated test cases (i.e., it cannot require the entire system for itself).
- *Self-operating*. It should not require manual intervention, for example, forcing an operator to respond to periodic messages before proceeding.
- *Restartable*. During testing, the system can fail at any moment, terminating the execution of many test cases prematurely. Therefore, when a test case begins running it must assume it is running in a "dirty" environment, where files or other resources it plans to create have been left around from a previous, aborted execution. So, its first step must be to attempt to delete any such resources, to ensure that its environment is clean before proceeding.
- *Self-cleaning*. This is related to, but distinct from, restartable. In SVT, and particularly in IT, test workloads can often run continuously for days, weeks, or months at a time. If the test cases in those workloads update system resources, such as databases, they must do so in a way that prevents those resources from "aging" ungracefully, or becoming invalid and in need of resetting. For example, a transaction-based workload may consist of many

test cases that read, update, add, or delete database records. To be self-cleaning, when an individual test case updates a record, before it completes it must reset that record to its value prior to the update—and do so within the same unit of recovery context. If the test case deletes a record, it must restore it before terminating. In this way, the data do not age; records are not deleted, nor does the number of records within the database grow. Yet, the required functions of the database system have been exercised.

Test cases that meet these standards can potentially live forever in automated z/OS regression buckets, earning their keep through reuse day after day. Indeed, once the regression phase of SVT is over, the job of regression test cases is not done—it has just begun.

New function. When the operating system has proven that it can withstand the onslaught of thousands of existing batch and interactive test cases running in parallel at high levels of load and stress for an extended period, then these regression test cases shift their role. They become a rich source of complex "background noise," providing the backdrop of heavy load and stress against which *new* test cases and scenarios can be executed. This is a key point. In z/OS SVT, no test of a new function, whether an application-like coded test case or a manually executed scenario, is considered successful until it has been executed in a high load and stress environment with the system pushed to its maximum usable capacity. A rich, varied, high-stress workload can flush out timing and serialization problems that are difficult to find in any other way. What better way to generate this background of load and stress than with a collection of thousands of test cases, built up year by year, designed specifically to exercise all aspects of the operating system with precision accuracy? We say that "good test cases never die—they just increase your stress."

Validation of all the new functions in a release of z/OS may require the creation of hundreds of coded test cases, all written to the exacting standards previously described. The amount of time required for such extensive test case development is frequently challenged by business demands. Reusing test cases saves time, but what about test cases for attacking previously nonexistent function? In SVT, we reuse test cases developed to exploit the new function in a *prior* test phase. The FVT team must spend a great amount of resources on test case development, but much of what these test cases cover are exactly the same functions the SVT team needs to exploit. Once the function testers have completed executing their test cases in a low-stress, single-user environment, if they have followed the SVT test case standards then their work can be reused. FVT test cases make an excellent starting point for exercising the new function in the heavy stress, highly concurrent SVT environment, reducing requirements for unique, SVT-specific test case development significantly while simultaneously broadening test coverage.

But, while such reuse does provide significant test efficiencies, it is still not enough. The FVT team must spend a great deal of time in test case development, and attacks against some new functions often require that SVT-specific test cases must be created. This development work can be slowed by the complexities involved in establishing the proper application environment for homing in on specific functions. The team addresses this issue through techniques such as modularized test case development tooling and inserting testability features directly into the z/OS product. Both are discussed later in this paper.

Scenario-driven testing. When validating new functions, in addition to coded test cases that exercise the function, an important technique is scenario-driven test execution. A scenario is defined as a series of discreet events (such as running test applications or issuing command-driven operator actions) executed in a particular order designed to bring about a particular result. By creating scenarios that simulate customer activities, one is able to focus special testing effort where it has the most value: in finding bugs likely to be pervasive and having a high impact if allowed to escape. A very simple example is dynamically configuring a CPU to be off line and then back on line—against a backdrop of high load and stress, of course. For a more interesting example, consider a z/OS feature, called Hiperbatch,[7] that speeds sequential reads against a certain type of file by a collection of applications accessing it in parallel. The support works by caching data read by the first reader as it works its way through the file, then satisfying subsequent readers of the same data by pulling it from the in-memory cache.

System testing of this function involved creating a group of cloned test applications that would read the same file, then enabling the caching feature for that file and running the cloned applications in parallel. This simple technique was indeed effective in flushing out bugs in the code, but eventually the group

of applications could run without error. The next step was to run multiple such groups in parallel, each against a different file. This also found problems, but again eventually completed without error. At this point, system testing of this function could have stopped, but it did not. The test team recognized that most users of this support function would not be running applications that were exact clones of one another against a particular cached file, nor would they necessarily run them in a tight group that all began and finished at the same time. In particular, they realized that the applications would likely perform differing amounts of processing against each record before proceeding to the next—meaning that the applications would make their way through the file at different speeds. Also, the applications would likely begin their processing at somewhat staggered time intervals.

Combining these two observations, the team created a scenario in which two groups of applications were run against a particular file. The first group was altered to include a small, artificial delay after each read. This "slow" group was started first. Once it was partially through the file, the next group was started. This second group had no such artificial delay; it read through the file as quickly as possible. Before long, the fast group caught up to and overtook the slow group. At that moment the "lead reader," the application instance for which data were being cached for the benefit of the others, changed.

The team found that precisely when this "changing of the leader" occurred, a data integrity bug hit. In fact, multiple such bugs hit, all caused by very narrow timing windows. Such windows will normally elude single-user tests or code coverage tools, because they depend not on a single errant path through the code, but on multiple occurrences at the same instant across multiple processors on a tightly coupled MP system. That is why it is so important to execute such scenarios under heavy load and stress.

The example shows how a scenario was created to attack data integrity in an environment of high concurrency by rearranging the execution sequence of existing test cases in a way that discovered new and critical bugs. Importantly, this sequence was not picked at random, but was based on the expected customer usage of the function.

Recovery. Since no software engineering process produces defect-free code, how can a goal of continuous availability be achieved? One technique is to accept reality and design, develop, and test software products with the assumption that failures will occur. z/OS programmers are taught to assume that the next instruction in their module may abnormally terminate. z/OS itself provides robust recovery services for programmers to exploit. In fact, nearly 60 percent of z/OS software deals in some way with the inevitability of errors—isolating them, recovering from them, and documenting them.[2] Recovery testing is really another class of z/OS scenario testing, but it deserves special mention. Internal, module-level recovery testing is normally the purview of FVT, where an emulation environment can be exploited to set breakpoints and generate error conditions. Recovery from external failure events fits more naturally into the "real-world" environment of SVT and IT. The external events can range from an unexpected system interrupt (such as a machine check), to the sudden loss of a major subsystem (such as a transaction monitor or database manager), to the failure of a hardware component (such as a disk drive or a zSeries coupling facility), to the complete failure of a partner system in a clustered environment.

In many cases, z/OS has specific support for shifting work away from the failing area, or dynamically rebuilding the failed component on a partner system, in such a way that the end user is unaware that any outage occurred. In these cases, test scenarios are devised where system load and stress are ramped up to extreme levels, then individual failure events are generated (either manually or through tools—see the discussion on the "coupling facility error injection" tool later in this paper) and the system is monitored to ensure that the automated recovery actions operate correctly. For example, one scenario would be to issue an operator command to abruptly terminate a CICS* (Customer Information Control System) transaction monitor region that is processing user work. This would force the region to go through its recovery processing, which in turn would alert the remaining regions to the failure and force new work to be diverted to them. The scenario continues by ensuring that the component failed without unexpected side effects, can be restarted cleanly, and resumes normal processing.

Regardless of the nature of the recovery support under test, it is important that a matrix of all critical failure conditions be created and specific scenarios devised to generate these conditions. Perhaps the failure can be created through a simple operator action. If not, then more sophisticated approaches or tools must be devised. Of course, since failures in-

variably occur at the most inopportune times (i.e., when the system is at its busiest), the scenarios must be executed while the system is running at high levels of load and stress.

The z/OS development laboratory spends a significant portion of each z/OS release cycle testing recovery functions. This special focus has no doubt contributed to the reputation of z/OS for reliability and recoverability, and ultimately to customer satisfaction.

Methodology summary. The z/OS test methodology has several distinguishing characteristics. First is the emphasis on steadily building upon what already exists, for everything from test plan development to workload creation. This philosophy may not be unique to z/OS testing, but it does stand in contrast to a well-known item in many software "best practices" lists, known as "minimizing regression test cases." Minimizing test cases is not the policy for z/OS, where part of the operating system's attractiveness is its ability to support multiple, disparate production workloads on the same server, and where part of a tester's goal is to emulate this wide-ranging environment with highly automated test streams. Building upon existing experiences and test suites has proven to be a very efficient approach to meeting that goal. Second, there is a heavy emphasis on load and stress, both in and of itself as a vehicle for proving systems stability, and as a backdrop for new function- and scenario-based testing. This is critical to validating system operation at maximum concurrency and capacity utilization. Finally, the extensive recovery support within z/OS demands and receives special focus in every phase of the operating system's testing, ensuring that if an error does occur, real-time system recovery will operate as designed and customer impact will be minimized. It is an excellent example of steering test efforts in the direction of the defects that matter the most to customers.

Tools and approaches

A strong tool suite helps a test team to execute its methodologies. Over the years, the z/OS team has developed a set of tools that accomplish several goals. They allow the automation of massive batch and interactive test case streams while directing tester attention to exception conditions. They address the need to simplify test case creation in complex environments in order to optimize that development resource, and do so in a modular way that allows reuse. They interact with specially introduced "testability" features in the operating system product itself. Finally, they focus on particularly difficult, yet critical areas, such as recovery and data integrity. A few of the key tools used by the z/OS test community are described in the sections that follow.

Note that a common thread, reuse, runs through these tools. Some tools have been designed from the beginning to be modular and reusable. Others proved so simple to use and effective for their initial purpose that they were modified and extended many times to meet new challenges. This consistent reuse not only saves time and allows the achievement of time-to-market goals; it also helps to ensure the stability of the base functionality of the operating system during development stages, where significant enhancements are likely.

Automated workloads. A combination of batch and interactive workloads running together forms the basis of z/OS SVT. Different tools are used to automate their execution. For batch test cases, z/OS testers use two primary tools: BERD and JMON.

The BERD (background environment random driver) tool is a simple program that sequentially or randomly submits test cases for batch execution, keeping track of which ones have already been run to avoid resubmitting a test case until the entire suite has been run. User input to the tool includes a list of MVS (multiple virtual storage) partitioned data sets that contain the test cases to submit, a count of the number of passes to make through all the test cases, and the number of seconds to wait between test case submissions (a value of zero tells the tool to submit the test cases as fast as it can). This has proved so effective that, whereas the test cases themselves have continually grown and evolved, the tool for submitting them has been used in the testing of z/OS and its predecessors, and has remained largely unchanged, since the early 1970s!

The BERD tool can drive the execution of many thousands of test cases within a given 8-hour period of machine time. Manual review of the output from each test case in search of failures is untenable, so a method of output reduction is needed. The z/OS team uses the JMON (Job MONitoring subsystem) tool for this purpose. JMON hooks into the z/OS job entry subsystem (in particular, it takes advantage of the subsystem interface of z/OS to establish itself as a secondary subsystem) to intercept the actual return codes issued by each test case and compare them against the expected return codes for that test

case—an oracle, if you will. When the expected result is achieved, the test case's output is flushed; when the expected result is not achieved, the output is retained and a highlighted message is sent to the operator console. During the run, JMON can be queried on the overall success/fail ratio for the workload; at the end of the run, JMON can provide a more detailed accounting for each test case. Furthermore, JMON exploits z/OS system services to establish itself, monitor activity, and purge or retain output. So JMON itself tests pieces of z/OS.

But batch processing is only part of the z/OS workload. Interactive test cases (which require simulated users to, say, submit an input form and process the response received) are also key. Interactive workload tests have in the past been driven through the TPNS (teleprocessing network simulator[8]) tool, which can simulate thousands of end users banging on keyboards to process work through various on-line systems (such as CICS and IMS* [Information Management System]). Many of these test cases run COBOL programs that update legacy databases within the context of a transaction monitor.

As zSeries customers have moved into e-business, they typically have not started by creating new applications "from scratch." A common first step is to adapt their traditional, 3270 interface-based transactional applications for the Web environment. In z/OS testing, we took the same approach. Our traditional COBOL transaction programs were originally written with no thought of Web processing. We treated them as our "legacy" applications and added new Web-based front ends, utilizing everything from CGI (Common Gateway Interface) to EJB** (Enterprise JavaBeans**) technology, just as our customers have done. But TPNS was unable to drive the resulting HTTP (HyperText Transfer Protocol), so new tools were needed to generate load and stress. Fortunately, workstation-based tools that simulate HTTP users coming in over a network are widely available in the industry and we were able to use one for this purpose. This is another example of continually finding ways to build upon existing test suites and exploit them for new environments.

Component Test Tool. When testing in the z/OS environment, there are many environmental and operational requirements that a test case must meet to be considered a comprehensive and useful test tool on the platform. The concept behind the Component Test Tool (CTT) was to create a testing framework within which testers could quickly and efficiently create and modify test cases using a highly structured programming language and built-in, easy-to-use services. The major strength of CTT is its support for building a wide variety of z/OS operating environments, quickly, consistently, and, most importantly, correctly. Traditional test case development in other programming languages, over time, can spread incorrect code models across many software test groups, which can lead to invalid test cases and allow defects to escape. CTT encapsulates, in callable services and other framework functions, the error-prone code required to establish the complex system environments in which test cases often need to execute. This technology allows the creation of environments and test models in a consistent manner, a very important aspect of code reuse. Once these code models and constructs have been validated, they can be quickly and confidently populated across many test cases, across many testing groups on the z/OS platform. This implementation of reuse has been a critical aspect of increased testing productivity. Additionally, as we will explain, user updates to CTT test cases do not require compile, assemble, and link. Therefore, test case modification time is significantly reduced and is, in some sense, interactive.

Overview. CTT provides a set of functions, or "verbs," that allow the tester to very easily create test cases in extremely complex environments. For example, executing in SRB mode (a service request block is a unit of work that carries higher than normal priority) in z/OS can require system locks, serialization, and other complex processes in order to be established. CTT performs all of the complex tasks of establishing these environments with the single invocation of a verb. This simplicity allows the tester to focus on the services being tested in SRB mode, rather than expend time and effort on the nontrivial coding of the SRB mode environment.

Figures 3 and 4 show the two approaches that can be used to create the SRB mode environment. The code in Figure 3 implements the scheduling of the SRB using a high-level programming language. We see from the amount of code and complexity of the services that must be called that there is opportunity for error. The program must perform the following tasks: enter a system-authorized state to perform the schedule operation, load the actual program that will execute when the SRB is dispatched, initialize the SRB control structure that outlines the system characteristics under which the SRB will execute, and exit the system-authorized state.

Figure 3 Traditional z/OS SRB schedule

```
/* Code fragment to schedule SRB (Service Request Block) routine */
  DCL CVTPTR PTR(31) LOCATION(16);
  DCL SRBREG REG(6) PTR(31) RSTD;
  DCL SRBPTR PTR(31);
  DCL ASCBPTR PTR(31);
  DCL SRB_ADDR PTR(31);
  DCL MYSRB CHAR(LENGTH(SRBSECT)) BDY(WORD);  /* SRB storage */
     /* SRB parameters */
  DCL SRB_PARMLIST BDY(DWORD);
    ?MODESET KEY(ZERO) MODE(SUP);
 /* GET NON-SWAPPABLE  */
    GEN(SYSEVENT DONTSWAP);
 /* LOAD THE SRB */
    ?LOAD EP('SRBRTN') LOADPT(SRB_ADDR)
        GLOBAL(YES,F) ERRET(BADLOAD2);
BADLOAD2:
  IF SRB_ADDR=0 THEN /* If load failed  */
   DO;               /* Terminate      */
   RETCODE=16;
   ?WTO('Load failed for SRB routine SRBRTN') ROUTCDE(11);
   END; /* End of load failed */
  ELSE DO;     /* SRB loaded */
      SRB_PARMLIST=''B; /* Clear out parameter list */
     /* SETUP SRB SRBRTN */
        SRBREG=ADDR(MYSRB); /* Get addressability to SRB */
        RFY SRBSECT BASED(SRBREG);
        SRBSECT=''B;  /* Clear out SRB */
        /************************/
        /* Initialize the SRB   */
        /************************/
        SRBPTR = ADDR(SRB);          /* Set address of SRB */
        SRBID = 'SRB ';              /* Initialize acronym */
        SRBPASID = PSAAOLD -> ASCBASID; /* SRB is to run in Home address space */
        SRBASCB = PSAAOLD;
        SRBPTCB = PSATOLD;     /* Purge tcb affinity */
        SRBEP = SRB_ADDR I '80000000'X; /* Indicate AMODE 31*/
        SRBPARM = ADDR(SRB_PARMLIST);/* Set up SRB parm list */
        SRBRMTR = ADDR(CVTBRET);    /* No resource manager  */
  /*************************************************************/
  /* Actual call to the system to schedule the SRB for execution */
  /* Additional complexity on scheduling the SRB is introduced   */
  /* when different types of system locks are required           */
  /* This is the most simplistic case                            */
  /*************************************************************/
     ?SCHEDULE SRB((SRBPTR)) SCOPE(LOCAL) LLOCK(NO)
             FRR(NO) MODE(NONXM);
/*   GET SWAPPABLE  */
 GEN(SYSEVENT REQSWAP);
/*   Get unauthorized  */
 ?MODESET KEY(NZERO) MODE(PROB);
```

Figure 4 CTT z/OS SRB schedule

```
INIT TESTCASE=SCHEDULE,ASID=MAINASID,
       MAINID=MAINTASK;  /* Identify the test case */
/*******************************************************/
/* Sample CTT deck to Schedule an SRB and wait for the  */
/* SRB to end its processing.                           */
/*******************************************************/
STARTDEF TYPE=MAIN,ID=MAINTASK; /* Start of MAIN task */
  SCHEDULE SRBID=SRB1,ASID=MAINSID;  /* Schedule an SRB */
  WAIT WAITID=WAITER1;  /* Wait for SRB to complete */
ENDDEF TYPE=MAIN,ID=MAINTASK;  /* End of MAIN task */

STARTDEF TYPE=SRB,ID=SRB1; /*Start of the SRB*/
  POST WAITID=WAITER1,ASID=MAINASID; /* Post main task */
ENDDEF TYPE=SRB,ID=SRB1; /* End of the SRB */
```

Figure 5 Multisystem CTT test

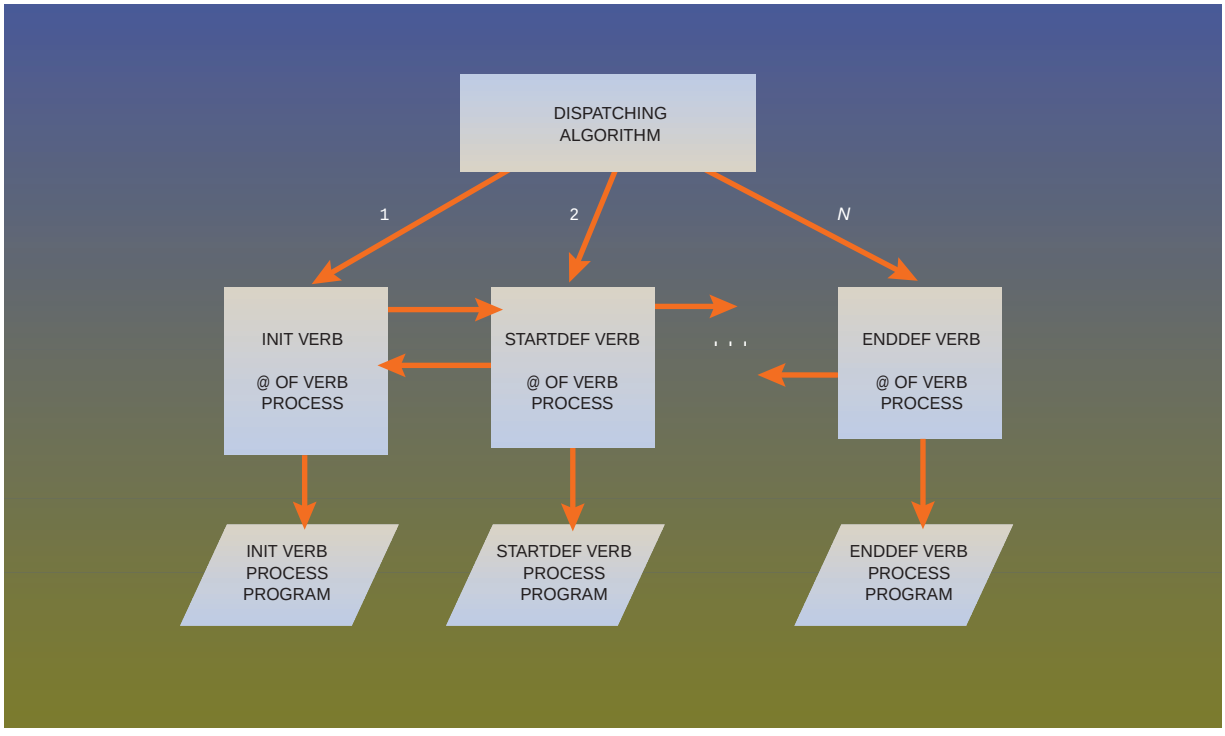
```
//CTTDECK JOB .....
  //CTTIN DD *
    INIT TESTCASE=CTTDECK,SYSTEMID=SYS1,
         CTT functions (VERBS) to be performed on SYS1
    INIT TESTCASE=CTTDECK,SYSTEMID=SYS2,...;
         CTT functions (VERBS) to be performed on SYS2
    INIT TESTCASE=CTTDECK,SYSTEMID=SYS3,...;
         CTT functions (VERBS) to be performed on SYS3
```

The code in Figure 4 makes use of the CTT test case infrastructure. The interesting part is the function named SCHEDULE. This single statement, by passing control to a CTT functional routine, performs the entire set of tasks outlined in Figure 3 without requiring the programmer to understand all of the details. Clearly, CTT makes it easier to develop test cases and eliminates worry about the complexities of the underlying environment.

Figure 5 shows another example of establishing environments using CTT. Here the tester has spread the test case across multiple systems in a z/OS Parallel Sysplex. This multisystem capability allows testers to validate cross-system functions and processes that are, otherwise, very complex to build and implement.

A CTT test case that is developed for multiple systems can execute in any multiple system Parallel Sysplex regardless of the number of systems currently active. If the CTT test case has been coded for more than the current number of active systems in the cluster, CTT will intelligently distribute the portions of the test case across the currently active systems. This helps in CTT's portability to, and reuse across, different test environments.

Figure 6 CTT dispatching algorithm



An additional capability allows other programming language modules and programs to be called from within the CTT environment, giving programmers the flexibility to code in familiar languages as extensions to CTT.

Implementation details. The design and implementation of CTT allows quick and convenient extendability. CTT is comprised of two processes, parsing and processing. During the parse phase the input is deciphered and interpreted. CTT completely parses the entire input stream and, if there are errors, reports on all of them and the execution stops. If the parse is successful, CTT builds control structures representing each individual input statement. These control structures are linked into a double-threaded queue structure that is an ordered representation of the input statements. Once the parse phase has completed, CTT begins processing each individual control structure element to perform the function that it provides. It processes these control structures using a simple dispatching algorithm that traverses the queue and examines the control structure representing a particular input statement. One of the data items within the control structure element is the address of a program to be called to perform the functions associated with the element. The control structure element contains all of the parameters and options specified on the input statement. The process program, which is called by the dispatching algorithm, interprets the data passed via the control structure element and performs the actual function requested. Figure 6 shows the CTT dispatching approach.

This implementation is key to the extendability of CTT. Both the parse program and process program that are called by CTT are simple plug-ins. Testers wishing to implement new or improved CTT verbs or functions can simply specify an input pointer to a set of verb definition files that specify the name of the parse and process routines that CTT is to load and utilize for the specific verb. At any time, testers can then develop their own verbs or functions for use under the CTT framework. The CTT user community reviews provided functions to determine their value to the overall tool and its users. If there is suf-

ficient value in the new functions, the support is added to the base CTT.

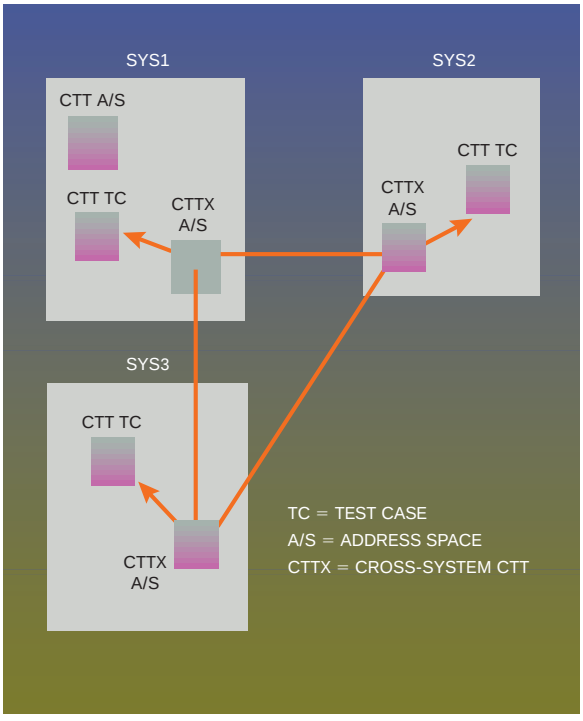
The conceptual implementation of the multisystem CTT test case initiated as shown in Figure 5 is shown in Figure 7. The CTT address space executing on SYS1 gets initiated. The main CTT address space communicates with the address spaces resident on all systems in the cluster to initiate the different portions of the test case on the target systems. At the conclusion or abnormal termination of the test case (on one or any of the target systems), all of the output is routed back to the initiating system (SYS1, in this case) where it is presented to the user.

CTT conclusion. CTT is a testing tool that was developed to be used by multiple testing phases in the development process. Its primary use is in the FVT phase, where system and application interfaces are validated and verified. However, it is also integral to z/OS SVT efforts where heavy load and stress conditions are exercised. Therefore, it is critical that test cases that are coded in this technology have self-verifying and self-cleaning characteristics. The system verification test team can itself make use of CTT's capabilities to build stress test cases that exercise the new functionality of zSeries.

CTT has evolved, over the past 15 years, along with the operating system and its new technologies. From the basic implementation of z/OS application program interfaces, to support for z/OS UNIX** systems services and other environments, this tool has moved with the platform to attack the most complex environments and problems. This is another example of adapting existing tools to a constantly changing business environment.

Software testability and the coupling facility error injection tool. In order to ensure recoverability of components of the z/OS operating system, constant interaction between the test and development communities is required. By working closely with developers, testers get a better appreciation for the actual code implementation and how it can be tested. Introducing "testability" or "ease of test" into the software that is being verified allows the tester to validate the software more efficiently and effectively. "How easy will it be for our testers to verify and validate this software solution?" is a question that all designers and developers should be asking themselves. Independent testing tools that attempt to hook into product code can interfere with the overall behavior of the product if not properly implemented by the testers. It is critical for the testers to work directly with the development and design teams to see where there are opportunities to improve the testability of the product code. One successful example is the coupling facility (CF) error injection program.

Figure 7 Multisystem CTT test case



The CF error injection program was developed in order to inject failures in S/390 coupling facility structures[9] without interfering with normal processing (i.e., without setting traps, modifying actual code, etc.). To do this, system operation codes were defined within the coupling facility support code to allow a tester to cause the next operation on the coupling facility structure to fail. This enhancement required help from architects and developers.

The user of the program targets a specific structure name and requests that a failure be injected against that structure. On receipt of this request, the coupling facility returns a failure response code on the next operation requested against that structure. The capability to inject errors at random allows the test organization to validate all of the appropriate recov-

ery actions of z/OS and all of its subsystem products. To date, there have been no field-reported problems with regard to recovery from a coupling facility structure failure condition.

This program has become an integral part of the S/390 testing approach and, most recently, it has been released for use by customers to help create recovery scenarios to test their applications and product solutions.[10] Each time new software making use of the zSeries coupling technology enters test, the use of the CF error injection tool should be considered.

To use this tool, the z/OS system programmer or operator creates a scripted routine to invoke the test tool. Then the operator or tester can invoke the tool with the following z/OS command:

```
S INJERROR,PARM='strname,XXX'
```

where INJERROR is the name of the tool, strname is the name of the coupling facility structure into which the error is to be injected, and XXX is an optional parameter that indicates whether an old or new instance of the coupling facility structure should have the error injected. This parameter is applicable in certain recovery cases.

The user receives the following messages to indicate the result of the command:

```
*INJERROR: PROCESSING STARTED
  - VERSION 2.0 - 09/01/99
*INJERROR: INPUT RECEIVED
  - STR=strname, STRTYPE=N/A
*INJERROR: STR FAILURE INITIATED
  AGAINST STR=strname
*INJERROR: PROCESSING COMPLETE
```

After receiving the "processing complete" message, the application structure into which the coupling facility failure condition is introduced will enter into its recovery actions. Simple to use and very effective, this tool allows the z/OS test team to ensure the recovery of the coupling technology platform on zSeries executing z/OS. This tool illustrates the kind of technology needed to ensure continuous availability on the zSeries platform.

Thrashers. Maintaining data integrity is an essential role of an operating system. Data placed in a storage location (either in memory or on disk) is expected to be valid and intact when it is later retrieved. This is so fundamental that most applications programmers never even think about it; they know that the operating system will handle it. But, as those who develop operating systems know, the code that moves data back and forth is just as likely to have bugs as any other code. In fact, given the complexities involved in managing storage, bugs in such code are quite likely.

Compounding the difficulty, data integrity problems are among the most difficult to debug. They are often caused by narrow timing windows related to serialization. Since normal test applications (or customer applications) assume data integrity is being maintained, they likely will not immediately notice when something has gone wrong. Instead, they simply continue processing the invalid data. Eventually, they may be affected in some way (such as dividing by zero) that will crash the application and bring it to the attention of the tester. Or, the program may end normally—but with invalid results. In any case, by the time the error is detected, the system has long since covered its tracks, the timing window has closed, and prospects for debugging it are slim.

Validating data integrity requires testers to think differently from other programmers. Specialized attacks are required. Test cases must be written that do *not* take it for granted that when a value is written to storage, that same value will later be retrieved. In fact, the test cases must assume the opposite—and be structured in a way that will facilitate debugging data integrity problems when they strike. In z/OS, we use a set of tools for this purpose that we refer to as "thrashers."

Our first thrasher was written in the 1980s in response to a data integrity bug in a prerelease model of the 3090 mainframe processor. The bug turned out to be related to how the machine was handling a type of memory used for fast paging and swapping, called expanded storage. At the time, of course, no one knew where the problem lay, only that data were being corrupted, and after weeks of analysis the culprit had proved elusive. A thrasher was written; it almost immediately caught the problem, which was then quickly debugged. Since then, variations on this short, deceptively simple program have been used during the testing of many products and features related to storage management, whether on disk or in memory. Easy to create and run, thrashers have proven themselves to be a very powerful and valuable tool in the z/OS test arsenal.

There are a few basic rules involved in creating a good thrasher. First, the processing of the thrasher code should be kept to the absolute minimum, to ensure that system code, not the thrasher code, becomes the bottleneck. Second, the thrasher should be designed in such a way that multiple copies of it can be run in parallel, as separate address spaces or processes. In z/OS, virtual storage is managed on an address space basis, and one possible bug involves pages from one address space being exchanged with those of another—running multiple thrashers in parallel is the way to catch such problems. Finally, the golden rule is: trust nothing.

Implementation details. Figure 8 shows a pseudocode representation of a virtual storage thrasher. Parameters to control the thrasher's execution are passed in from the user, including the number of pages of storage to thrash through and any delays desired for throttling the thrasher's speed. A template for what will be stored in each page is defined. Note that the first two fields here, PADDR and PASID, are used to uniquely identify each page in a way that the program can independently validate. For PADDR, the page's own virtual address is stored into itself. For PASID, the identifier for the address space within which this instance of the thrasher is executing is saved. These two values will be used to determine if corruption has occurred. The remaining data fields serve a dual purpose. On the one hand, they provide useful debugging information. On the other hand, they provide fields that the thrasher can change, forcing real storage frames to be updated.

Next, the thrasher dynamically obtains the storage table and initializes it. Note that the size of each entry in the table is related to how the underlying operating system and hardware manages virtual storage. z/OS manages storage on a 4096-byte page basis, so each entry is 4096 bytes long.

Finally, the thrasher goes into an infinite loop and begins working its way through the table. For each page, it first checks for corruption. If any is detected, it immediately forces abnormal termination. Typically, the tester sets a system trap for this abend, which upon detection will immediately freeze the entire system so that storage can be dumped and the failure analyzed. If no corruption is detected, then the program updates the table entry to force a write to the real storage backing the virtual page. It performs any delays requested by the user, then proceeds to the next page. Note that with this flow, after a page has been updated, it is allowed to "brew" for a while before it is rechecked for corruption. This is necessary in order to give errant timing windows sufficient opportunity to arise, but of course this means that there will be a slight delay between the time an error occurs and when it is detected.

Execution. The thrasher program is started, and it runs until it is canceled or detects a defect. As stated earlier, multiple instances of a given thrasher are normally run concurrently. Similarly, streams of unrelated thrashers are often run in parallel. In fact, a secondary benefit of thrashers is that they provide an easy method for generating high load and stress, with minimal setup requirements, and can provide good background "noise" while other tests are run in the foreground. Furthermore, the design described here of that first thrasher written two decades ago has been reused and extended over the years for verifying data integrity in many additional technologies as they have come along, including such z/OS features as data spaces, hiperspaces,[11] coupling facilities, BatchPipes,[12] Hiperbatch, and the UNIX Systems Services hierarchical file system.

Thrasher conclusion. Data integrity problems are certainly at the top of the list of defects that matter the most to customers, for any operating system. It is no surprise that companies call their information technology buildings "data centers." Thrashers have proved themselves to be an easy-to-use, yet powerful tool for validating data integrity. Their simplicity encourages frequent use and reuse by testers. This is another example of taking a testing tool that has demonstrated its effectiveness and finding ways to build on that strength for the benefit of subsequent projects.

Measurements

How effective are the methodologies, tools, and techniques used by the z/OS test team? In some sense, the widespread reputation of z/OS speaks for itself—and the bar set by our customers can be very high. For example, one grocery store chain in the United Kingdom has run a six-system Parallel Sysplex, spread across two sites, for as long as 550 days without a single minute of sysplex-wide outage, planned or unplanned. A major Canadian bank has kept their six-system Parallel Sysplex, which runs their critical banking applications, available for over 4 years. This has been achieved while making numerous upgrades to all hardware and software components to maintain currency. And a retail business in the United Kingdom has achieved 100 percent application avail-

Figure 8 Pseudocode representation of a virtual storage thrasher

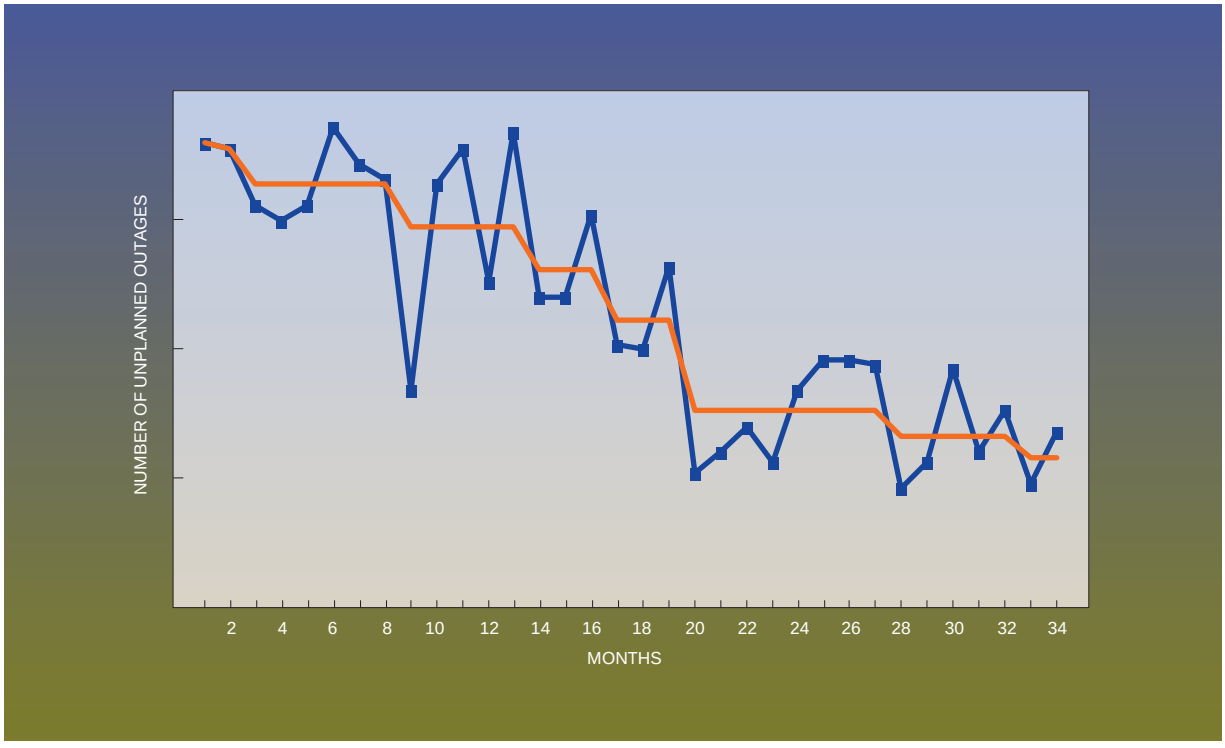
```
PROGRAM THRASHER(PAGES FIXED(31), WAIT1 FIXED(32), WAIT2 FIXED(32));

Declare PAGESIZE = 4096;           /* One page equals 4096 bytes */
Declare STORPTR, PAGEPTR PTR(31);  /* Pointers to beginning of obtained storage table, and individual pages */
Declare PAGENUM FIXED(31);         /* Loop counter, corresponds to pages in storage table */
Declare REFNUM FIXED(31);          /* Counter of number of times we've looped through the storage table */
Declare 1 PAGEMAP BASED(PAGEPTR),  /* Template for each page in table */
          2 PADDR   PTR(31),       /* Address of this page in virtual storage */
          2 PASID   FIXED(16),     /* Address space ID which owns this page */
          2 *       CHAR(2),       /* Dummy halfword,so what follows will be on word boundary when viewed in dump */
          2 PTIME   CHAR(8),       /* Clock from last time this page was updated  */
          2 PSYSNAME Char(8),      /* Name of system */
          2 PJOBNAME Char(8),      /* Name of job running this instance of the thrasher */
          2 PCOUNT FIXED(32);      /* Number of times this page has been updated */
GET STORAGE ADDRESS(STORPTR) LENGTH(PAGES*PAGESIZE) BOUNDARY(PAGE); /* Get storage table to be thrashed through */
PAGEPTR=STORPTR;
DO PAGENUM=1 TO PAGES;                          /* Initialize storage table */
   PADDR=PAGEPTR;
   PASID=MyASID;                                /* MyASID obtained from system control block */
   STCK(PTIME);                                 /* Store value obtained from current clock */
   PCOUNT=0;                                    /* Page not trashed through yet */
   PJOBNAME = MyJob;                            /* MyJob obtained from system control block */
   PSYSNAME = SysName;                          /* SysName obtained from system control block */
   PAGEPTR=PAGEPTR=PAGESIZE;                    /* Go to next page in storage table */
END;

REFNUM=1;
DO FOEVER;                                      /* Loop through storage area until job is cancelled */
     PAGEPTR=STORPTR;
     DO PAGENUM=1 TO PAGES;                     /* Make a pass through the storage area */
       IF PADDR¬=PAGEPTR | PASID¬=MyASID THEN   /* Data integrity error detected */
          Force the program to abend;           /* Force an OC3 abend */
       ELSE
          DO
          PADDR=PAGEPTR;                        /* Else, update page again... */
          PASID=MyASID;                         /* MyASID obtained from system control block */
          PJobName = MyJob;                     /* MyJob obtained from system control block */
          PSysName = SysName;                   /* SysName obtained from system control block */
          STCK(PTIME);                          /* Store current clock value in this page */
          PCOUNT=REFNUM;                        /* Update reference count for this page */
          PAGEPTR=PAGEPTR+PAGESIZE;             /* Go to next page */
          IF WAIT1¬=0 THEN                      /* Delay between pages */
             Wait for WAIT1 Seconds;
          END;
     END;

     REFNUM=REFNUM+1;
     IF WAIT2¬=0 THEN                           /* Delay between passes through table */
         Wait for WAIT2 Seconds;
END;
```

Figure 9 Number of unplanned outages by month over a three-year period



ability for a critical depot management application running in a cluster spread across two sites for the last *eight years*. Indeed, in the z/OS mainframe environment, system reliability has been proven under fire, time and time again.

Now we dig deeper and look at some data. Several different classes of data could be examined, but here we focus on indicators of defects that escaped during test and were found in the field. In particular, we look at system outage reports and high-impact defects reported by customers.

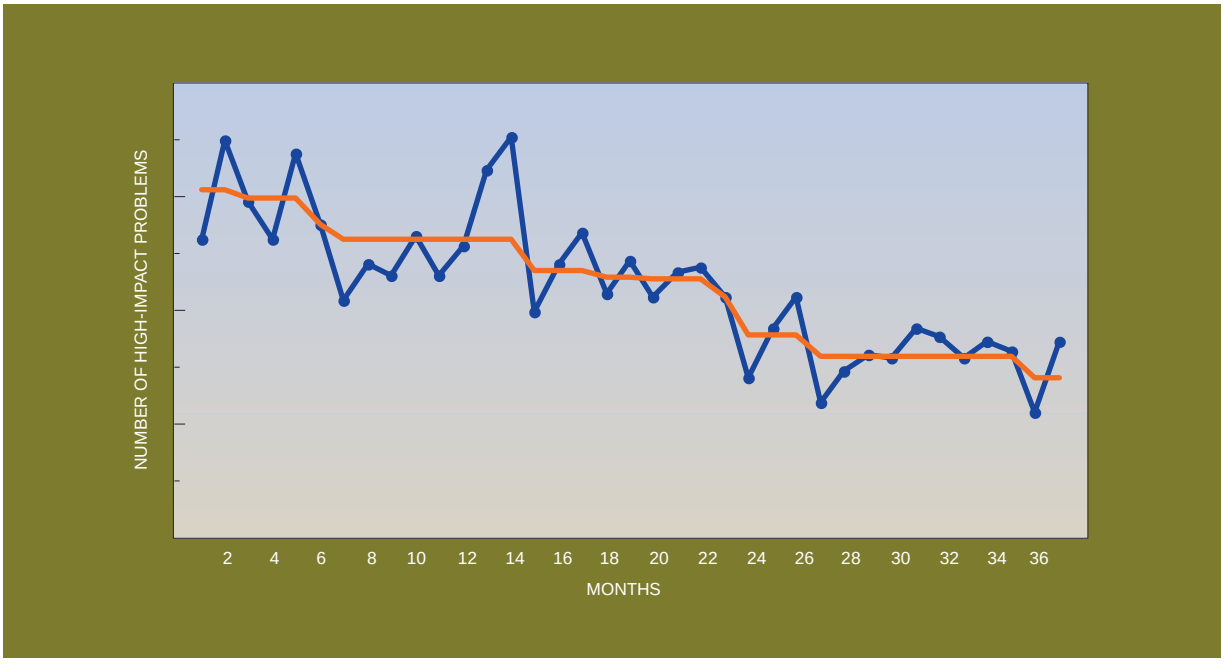
In Figures 9 and 10, we used *isotonic regression* [13] to smooth the data, and the smoothed curves are shown as red lines. This procedure finds the monotonic data sequence closest to the actual data.

Figure 9 shows customer-reported unplanned outages across more than one member of a sysplex cluster. The data points are by month over almost three years, and demonstrate a steady improvement in system availability. This resiliency is the system characteristic targeted by the test team's focus on real-time recovery. As described earlier, FVT tests module-level recovery of internal failures and SVT tests system recovery from external failures while the system is performing multiple tasks under load and stress conditions.

Figure 10 shows field-reported defects identified by the customer as having significant impact to their system. Each of these, of course, represents a problem that escaped detection during test. The data are by month over a three-year period and show an ongoing reduction in high-impact defects. This reduction parallels our testing objective: to target the defects that matter most to customers.

So, the data show a clear trend toward the reduction of both system outages and high-impact defects discovered in the field over time. Drawing definitive correlations between field data and test effectiveness is hardly an exact science. Certainly a number of other factors contribute to these results, not the least of which are changes in the product itself. Nonetheless, a key goal of the testing approaches described in this paper (the steady buildup of reused test cases

Figure 10 Number of high-impact problems by month over a three-year period



combined with new ones that accurately attack complex environments, a relentless focus on the testing of recovery and data integrity features, cooperation between testers and developers to ensure the testability of new features, etc.) is to achieve a downward trend in the defects that adversely affect customer availability. The data show that actual results are consistent with that objective.

Conclusion

The z/OS mainframe operating system and its predecessors have been part of commercial computing for decades. z/OS includes certain characteristics that FORTUNE 500 companies rely on, such as highly usable capacity, strong concurrency, consistent application compatibility, pervasive real-time recovery, and robust data integrity. Validating these attributes poses a set of challenges. This paper has explored those challenges and described methodologies and tools for meeting them. Examples were provided based on implementation experiences at the IBM z/OS development laboratory in Poughkeepsie, New York, along with data showing the results of those efforts. The intent was not to perform an exhaustive review, but rather to provide a practical guide to several key approaches that have proven themselves over many years.

Not all techniques described here will be universally applicable. However, we assert that use of the focused methodologies and standards, full exploitation of easy-to-use, modular tools that encourage extension and reuse, cooperation between test and development organizations to ensure the testability of new features, and a relentless emphasis on heavy load and stress in the system test phase should be key aspects of any real-world software test strategy. Experiences in testing z/OS have shown that key targets for tooling are areas such as workload automation, simplification of complex programming environments, error injection, and data integrity. Not coincidentally, these are areas of concern to customers who insist on rich functionality and high availability. Exploiting techniques and tools that zero in on these areas is a way to find defects that matter.

Acknowledgment

We thank Ram Biyani for help with isotonic smoothing.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Sun Microsystems, Inc., or The Open Group.

Cited references

1. *Enterprise Server Essentials*, GF22-5122-00, IBM Corporation (May 1999).
2. F. Bothwell, *Meeting the Business Challenges of the 21st Century: Comparative Large System Capabilities and Attributes*, Enabling Technologies Group, Inc. See http://www.etginc.com/services/publications/index.shtml.
3. *OS/390 V2R10.0 MVS Setting up a Sysplex*, Chapter 4, "Managing Coupling Facility Resources," GC28-1779-09, IBM Corporation (July 2000).
4. N. S. Bowen, J. Antognini, R. D. Regan, and N. C. Matsakis, "Availability in Parallel Systems: Automatic Process Restart," *IBM Systems Journal* **36**, No. 2, 284–300 (1997).
5. The z/OS integration test team's experience reports are available on the Internet at http://www-1.ibm.com/servers/eserver/zseries/zos/integtst/.
6. J. A. Whittaker, "How to Break Software," *Proceedings, The International Conference on Software Testing Analysis & Review*, Orlando, FL (May 1–5, 2000).
7. *MVS Hiperbatch Guide*, GC28-1470-00, IBM Corporation (March 1994).
8. *TPNS V3R5 General Information*, GH20-2487-08, IBM Corporation (October 1996).
9. J. M. Nick, B. B. Moore, J.-Y. Chung, and N. S. Bowen, "S/390 Cluster Technology: Parallel Sysplex," *IBM Systems Journal* **36**, No. 2, 172–201 (1997).
10. The CF error injection tool is available on the Internet at http://www-1.ibm.com/servers/eserver/zseries/zos/integtst/injerror.html.
11. *OS/390 V2R7.0 MVS Extended Addressability Guide*, GC28-1769-04, IBM Corporation (January 1999).
12. *IBM BatchPipes/MVS Introduction*, GC28-1214-02, IBM Corporation (July 1995).
13. R. E. Barlow, *Statistical Inference Under Order Restrictions: The Theory and Application of Isotonic Regression*, John Wiley & Sons, Inc., New York (1972).

Accepted for publication September 25, 2001.

Scott Loveland *IBM Server Group, 2455 South Road, Poughkeepsie, New York 12601-5400 (electronic mail: d10swl1@us.ibm.com).* Mr. Loveland is a senior software engineer in the z/OS development laboratory. He joined IBM in 1982 after receiving his B.A. degree in computer science from the University of California, Berkeley. His entire career within IBM has been spent in test for z/OS and its predecessors, MVS™ and OS/390®. His work has spanned the function, system, and integration test disciplines, with specialization in real storage management, Hiperbatch, dynamic I/O management, Parallel Sysplex, networking, and Web-based e-business transaction processing. He has authored two inventions in the area of testing for data integrity, and teaches a class within IBM on performing effective system testing. Most recently, Mr. Loveland has been working as the Linux test architect in the Poughkeepsie zSeries software test organization.

Geoffrey Miller *IBM Server Group, 2455 South Road, Poughkeepsie, New York 12601-5400 (electronic mail: geoffm@us.ibm.com).* Mr. Miller is a Senior Technical Staff Member in the z/OS development laboratory. He joined IBM after graduating in 1982 with a B.S. degree in mathematics from Muhlenberg College. He has held various technical leadership positions in MVS, OS/390, and z/OS product packaging and test. Mr. Miller worked closely with the first customer of Parallel Sysplex technology, playing a key role in the joint IBM-customer study and prototyping efforts. He has designed and implemented many large-scale system and integration test efforts for the zSeries platform and the z/OS software stack. Mr. Miller is a core member of the zSeries Business Leaders Council and also an original member of IBM's corporate-wide Software Test Community Leaders (STCL) group. Most recently, he has been working as the chief test architect for the z/OS Solution and Integration Test organization.

Richard Prewitt *IBM Server Group, 2455 South Road, Poughkeepsie, New York 12601-5400 (electronic mail: prewitt@us.ibm.com).* Mr. Prewitt, a senior software engineer in the z/OS development laboratory, joined IBM in 1984 after graduating with a B.S. degree in computer science from Pennsylvania State University. He has worked in test over his entire career within IBM. He spent a number of years working as a function component tester on MVS/XA™, MVS/ESA™, and OS/390. His area of expertise includes base components of the operating system, Parallel Sysplex, and coupling technology. He spent a significant amount of time in designing and developing test-tool technology, and was one of the original creators and authors of the Component Test Tool (CTT), which is discussed in this paper. After leading in many of these areas, he took on a leadership role in the area of integrated testing. He was the overall leader for the OS/390 COMBAT (OS/390 Community Build and Test) team. This is a worldwide team of product packagers and testers who built and tested an integrated testing platform for each release of z/OS. Most recently, Mr. Prewitt has been working as the test project manager in the IBM License Manager area.

Michael Shannon *IBM Server Group, 2455 South Road, Poughkeepsie, New York 12601-5400 (electronic mail: mshannon@us.ibm.com).* Mr. Shannon is a senior software engineer in the z/OS development laboratory. He joined IBM at Essex Junction, Vermont, in 1969. From 1969 until 1993 he was an MVS system programmer at an internal data center. In 1993 he transferred to the OS/390 System Test group in Poughkeepsie as a software debugger. From 1993 to the present time he has worked as both a software debugger and z/OS system programmer in the z/OS System Test group.