Randomized Differential Testing as a Prelude to Formal Verification

Alex Groce, Gerard Holzmann, and Rajeev Joshi
Laboratory for Reliable Software *
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109. USA

Abstract

Most flight software testing at the Jet Propulsion Laboratory relies on the use of hand-produced test scenarios and is executed on systems as similar as possible to actual mission hardware. We report on a flight software development effort incorporating large-scale (biased) randomized testing on commodity desktop hardware. The results show that use of a reference implementation, hardware simulation with fault injection, a testable design, and test minimization enabled a high degree of automation in fault detection and correction.

Our experience will be of particular interest to developers working in domains where on-time delivery of software is critical (a strong argument for randomized automated testing) but not at the expense of correctness and reliability (a strong argument for model checking, theorem proving, and other heavyweight techniques). The effort spent in randomized testing can prepare the way for generating more complete confidence using heavyweight techniques.

1. Introduction

Every space mission generates a large amount of data, which must be stored until it can be downlinked to Earth for analysis (or display on NASA TV and websites). In recent missions, JPL has relied on *flash memory* to store most of this data, as flash uses little power and mass, no moving parts, and has a high information density — making it ideal for space mission use. For convenience and flexibility, most of this data is stored in hierarchical file systems similar to those used in standard operating systems. The data stored is often irreplaceable (e.g., landing telemetry or images of impact with a comet) or critical to spacecraft operation: it is essential that flash file systems provide high reliability. JPL's experience with commercial flash systems has not established confidence that current commercial products provide this level of reliability. In particular, unexpected behavior of the flash file system left one of the Mars Exploration Rovers in an uncontrolled state facing a critical low power situation, and resulted in days of lost science activity [15]. Flash in space operates in a hostile environment: solar flares and radiation levels make hardware faults more likely. File systems must survive the system reboots used as a fault protection response without loss of data integrity.

*The work described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

The Laboratory for Reliable Software has been developing flash file system software for use in future space missions, with the goal of using the full array of available software verification and validation technologies — randomized testing, runtime monitoring, static analysis, model checking, and theorem proving [3] — to establish confidence in the correctness of this file system.

This paper describes our efforts with randomized differential testing, including our approach to hardware simulation with fault injection, randomized test input generation, design for testability, and automated test case minimization. Our eventual goals encompass more formal, heavyweight approaches to correctness, but past experience (and common sense) indicate that during the early stages of development, when requirements and code are fluid and rich in errors, systematic, disciplined *testing* with a high degree of automation offers a much quicker path to finding many bugs. The infrastructure and specification effort applied to testing will pay off in more rigorous approaches (i.e., model checking and theorem proving, as discussed below).

More importantly, the use of modern randomized testing techniques offers an opportunity to improve the state-of-practice in flight software development at JPL. Testing efforts at JPL currently rely on nominal and stress testing on actual flight hardware. This testing is essential to understand behavior and performance in typical scenarios, but is ineffective at exposing low-probability software errors. Experience shows that limited nominal scenarios seldom describe the full set of actual system behaviors, since what

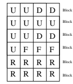

Figure 1. A typical flash configuration
U denotes used pages; **F** denotes free pages; **D** denotes *dirty* pages with no-longer-valid data; **R** denotes reserve blocks used to ensure that garbage-collection is possible.

may be considered "atypical" often does arise in practice. This approach also results in heavy contention for expensive flight hardware testbeds, threatening launch schedules. Better hardware simulation and large-scale randomized testing on developer workstation hardware offers the chance of exposing subtle errors earlier, and is likely to prevent dangerous defects from reaching the launch pad. The review process for our file system offers a chance for JPL's software developers and engineers to evaluate the effectiveness of our approach and the applicability of similar methods in other flight software. Of course, the most important product of this work is (we hope) a much more reliable critical component for operation in deep space missions, contributing to our knowledge of the remote physical universe as well as to the more earthly field of software engineering.

1.1. NAND Flash: Operational Characteristics and Fault Modes

The two primary types of flash memory are NAND and NOR flash [16]. Our efforts have focused on NAND, which presents a greater challenge from the viewpoint of reliability, since blocks may become bad at any time. In addition, because NAND flash has higher density, it is typically the primary storage medium for a mission's *data products*, and is therefore likely to undergo much heavier use than NOR flash (which is typically used to store flight software binaries, which are updated far less often). We are also currently planning an extension of our design to NOR flash.

A NAND flash device consists of a set of *pages*, grouped into larger units called *blocks* (see Figure 1). The basic operations on a NAND device are: *write* to page, *read* a page, and *erase* a block. Once a page has been written, it may be read any number of times. In general it is impossible or unwise to write to a page once it has been written to, until it has been erased (since a write is actually an AND operation). Erases are only possible at the block granularity. Flash file systems must manage invalid and outdated pages

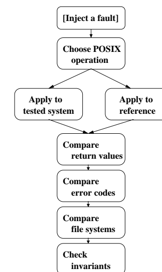

Figure 2. Our inner test loop

and perform garbage collection, rather than relying on overwriting old data.

NAND flash suffers from *bad blocks*: writes and erases to portions of the device may fail, or suffer from a high error rate. A list of initial bad blocks is provided by the card manufacturer, but new bad blocks may appear during operation. Managing the list of known bad blocks is a requirement for a reliable flash file system. Flash blocks have a limited lifetime of erases, and the probability that a block will go bad increases as this threshold is approached. Extended mission life requires a strategy for wear-leveling to prevent early failure of blocks.

Errors in page writes or corruptions of written bits due to radiation (or reset during a write operation) are guarded against by error detection and correction bits (EDAC) on the hardware, but the file system must be robust in the presence of uncorrectable errors.

1.2. Test Strategy

Our randomized test system repeatedly invokes the following strategy (see Figure 2):

1. Optionally inject a fault $\mathcal{F}$ (e.g., write failure or system reset) into the hardware simulation layer.
2. Randomly choose a POSIX operation, $\mathcal{P}$, and vector v of parameters for the operation.
3. Apply $\mathcal{P}(v)$ to the tested system.
4. Check for failure due to injected faults, and take appropriate actions (re-mount after a system reset, etc.).
5. Apply $\mathcal{P}(v)$ to the reference system, if injected faults did not prevent the operation from taking place.

6. Compare return values and error codes for tested and reference systems. Terminate with test failure if results differ in unacceptable ways.

7. Compare file system contents and check invariants.

This is repeated until an error is detected or a maximum test length has been exceeded. A test case (which consists of a sequence of POSIX operations, non-POSIX file system operations, and fault injections) begins with no interesting content on the flash device and builds up an increasingly complex set of directories and files (and thus written pages on the device). The results we report describe the errors found during billions of iterations of this core loop.

1.3. Overview of Results and Lessons Learned

Testing continues to this day, and will continue into the indefinite future, making all results provisional. This report is focused on the first months of testing. At the end of this period, testing was producing no failing test cases. To date, the longest continuous run of successful tests includes over 3.5 billion randomly selected operations. We estimate that mission usage for a year would produce no more than 50 million operations, selected from a more benign set (and with far fewer expected faults) than in our randomized tests.

Figure 3 shows the defects discovered during each of the first 25 weeks of testing. In the months since then, no defects have been discovered by randomized testing. The labels indicate important stages in testing. During the second week of testing, the problematic `rename` operation was added. During the fourth week of testing, bad block faults were introduced into the mix. During the seventh week of testing, a check for usage of bad blocks was strengthened, resulting in the large peak in the number of detected errors (this particular check exposed many obscure, semi-independent corner cases). Multi-partition operations were tested once they were well-defined in requirements, around the eighth week, and "stress" tests focusing on nominal operation sequences were introduced during the tenth week. Finally, read errors were introduced into the fault model during the eighteenth week of testing. The graph shows the expected curve of an initially large number of defects discovered with each additional level of test thoroughness, followed by a period of declining defects as the code stabilizes and regressions prevent check-in of new errors (we report defects in builds, not those discovered during developer regression checks before commit). The final weeks, with no errors, point to the future: despite large numbers of tests and coverage measures meeting our goals, the inability to find errors does not establish that the software performs as expected. Randomized testing cannot explore, even for a small state space, all the interesting configurations of the file system. As noted, we view randomized testing as a prelude to a more rigorous specification and verification effort.

We describe the experience of a group of model checking and formal methods researchers with randomized testing: the effectiveness of randomized testing at bug-finding establishes a challenging baseline for more heavyweight techniques. We also note that the features that we believe made randomized testing effective — design for testability, availability of a reference implementation, and well understood fault model — should make model checking and theorem proving easier. In a sense, the evaluation of our experience is impossible at this point: the real question is whether important errors remain in the code, undetected by our testing. In another sense, the chief lesson is that, even when the *goal* is full correctness, and resources and interest are available to pursue this goal, randomized testing is worth pursuing. Testing and formal verification are not in conflict: both benefit from the same attention to design for error exposure and the use of as much tool-based automation as possible. The difficulty of model checking (and theorem proving) makes randomized testing attractive, when software actually has to be delivered. Moreover, though the uncertainty of randomized testing makes heavyweight techniques attractive when correctness and reliability really matter, randomized testing is crucial for *actually checking the C code that will be used in flight*, as the more heavyweight techniques do not, in our experience, scale to *the C implementations* of a system as complex as a flash file system. Despite the promise of our experiences with a somewhat simpler flash storage system (see Section 7), when model checking or proving theorems for the full file system we have been forced to concentrate on design models, rather than the full implementation.

2. Related Work

McKeeman first described the use of randomized differential testing for C compilers [9]. His domain was essentially static: a test case was randomly generated based on some model of valid inputs, and output was compared for a variety of systems (C compilers). We report on differential testing in a more reactive context: the results from POSIX operations on the reference system were used to influence our choice of future operations. In one sense, we report on an easier task: constructing valid, interesting random sequences of POSIX operations proved easier than constructing random C programs satisfying various semantic or syntactic properties. In another sense, we report on a more difficult task, in that fault injection considerably complicated the issue of test evaluation — the reference system was not subjected to any faults.

In the long-term, we hope to use the methods and infrastructure developed in our testing efforts to support more thorough and formal verification of the file systems before

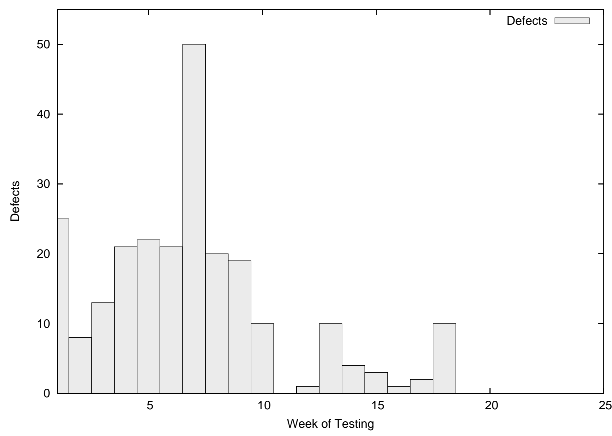

Figure 3. Results for the first 25 weeks of testing

they are used in flight. We believe the reference implementation and hardware simulation will be equally useful for model-driven verification [6] using the SPIN model checker [4]. Yang et al. describe previous efforts to find serious errors in file systems via model checking with CMC [11], and note the utility of testing as a prelude (or postscript) to model checking [17].

3. Hardware Simulation with Fault Injection

The most critical element in testing was a simulator for the flash hardware. Software simulation enabled efficient testing and full control over fault injection.

The simulator is implemented as a library, matching the function signatures of the actual hardware driver. In addition to the core functions provided by the real hardware driver, it provides initialization and configuration (flash devices of arbitrary size can be used in testing) and *fault injection hooks*. Effort invested in building a good simulation layer for testing has paid off in more rigorous approaches, as well: the flash simulator has been essential for model checking a flash storage module (see Section 7), providing a backtrackable "flash device" for use in model-driven verification [6].

3.1 Bad Block Faults

The simplest faults to inject are write and erase failures. The failure of a block of flash memory is detected as a failure of a write or erase call after a trap is set via a countdown mechanism, forcing the nth write (or erase) to fail.

3.2 Read Failures

Sometimes a write call returns success but fails to actually write to the page. The failure manifests itself when a read call is issued for that page. For critical data, the simulator (and driver) provide a *verified-write* call, which writes to a page and immediately reads back the data to confirm that the write worked. Read faults are injected using the same countdown mechanism as write and erase failures.

3.3 System Resets

System resets are simulated by placing the hardware into a *reset mode* in which all write and erase operations fail. A trap is set with a countdown, as with other faults. The file system software continues to operate after a "reset" but is unable to make changes to the flash state. Eventually, when control returns from the file system to the test driver, the in-memory structures of the file system are cleared and reinitialized. It would be possible to directly return control to the test harness from within the driver; we continue execution for two reasons: (1) to obtain useful information from return values (such as the number of bytes successfully written by a `write` call) and (2) to check for sane behavior in the presence of complete hardware failure.

4. Differential Testing: Oracle via Reference Implementation

The most difficult challenge in randomized testing is determining if the tested system is behaving correctly. *Generating* a useful set of random inputs is challenging (though even pure "fuzz" can expose errors [10]), but determining whether a test run that does not crash the tested system has performed correctly over the inputs is even more difficult. Limited ability to detect faults may be derived from system invariant checks and assertions or runtime monitoring of temporal logic properties, but full functional correctness is often difficult to encode as a specification. A *working implementation* is a very useful "specification" for functional correctness. Differential testing works by comparing the behavior of a tested system to another implementation of similar functionality. Every divergence, in theory, exposes an error in either the tested system or the reference system.

We used heavily-tested and widely available file systems, including the Solaris file system, Cygwin, and EXT3 and tmpfs on Linux, as reference implementations. Most tests compared results to tmpfs, which was the fastest of these implementations.

4.1 Hashing the File Systems

Instead of performing a full comparison of all file contents at each stage of testing, we opted to compute a hash $\mathcal{H}$ over each file system. $\mathcal{H}$ is based on directory structure, file names, and file sizes. It does not take file contents into account. We use a hash for two reasons:

1. It is expensive to perform a complete comparison of file system contents after each operation, but an approach based on sampling would result in late detection of errors. When an operation is not applicable to the reference system, a hash allows us to avoid examining the reference .

2. A hash provides history that direct comparison does not: we can determine if an operation changes flash state (other than file contents) by comparing hashes before and after the operation, without keeping a more detailed history of contents.

4.2 Comparison in the Presence of Faults

The primary challenge in differential testing was handling faults injected into the flash system. In the first week of testing, most effort was spent in dealing with ambiguities of the POSIX standard. In particular, handling the instances where Linux (or Solaris) gave a different, undesired, error code, producing spurious divergences, was a substantial one-time effort. After this was completed, most development in the testing framework was devoted to fault handling.

Injecting bad blocks was easiest, as in most cases, a bad block should not cause an operation to fail. Most write and erase failures are not visible to an observer. The exceptions are (very rarely) when the number of bad blocks detected while trying to complete an operation exceeds a threshold, forcing the file system to fail that operation and return a special error condition, or when space is exhausted on the flash system due to bad blocks. Space-usage computations therefore take into account a conservative approximation of the number of pages unavailable for writing due to bad blocks.

When read errors are injected, determining which differences in behavior indicate software defects becomes more difficult. It is impossible for the flash file system to perfectly mimic the reference system when pages committed to the flash hardware may not actually be present. For testing with read errors, we relaxed the requirement that file contents match those on the reference system. Directory contents were required to match, as the use of verified writes (see Section 3.2) allows the file system to ensure correct storage of metadata.

The test harness checked a strong atomicity claim for resets: an operation in progress at the time of reset must either complete or leave visible system state unchanged (except in the case of multi-page writes, as noted below). This allows us to handle all resets with a simple algorithm, applied after every call to the flash file system (in the context of some operation $\mathcal{P}(v)$):

1. Call into the simulation layer to determine if reset mode has been triggered. If not, continue normally (no special handling for reset is needed; apply the operation to the reference as usual).

2. Turn off reset mode (since it will cause all write and erases to fail) and reinitialize the flash file system.

3. Compute a new hash $\mathcal{H}$ for the flash file system.

4. If $\mathcal{H}' \neq \mathcal{H}$, apply $\mathcal{P}(v)$ to the reference system.

5. Special case: if $\mathcal{P}$ is a `write` call, perform $\mathcal{P}(v')$ on the reference system, where v' replaces the *amount-to-be-written* with the *amount-of-bytes written* returned from the call to $\mathcal{P}(v)$ on the flash file system.

```
   5:: - (creat /gamma)  = 0    *success*
   6::(rename /gamma /gamma)  *EBUSY*
   7::(rename /gamma /gamma)  *EBUSY*
   8::(truncate /gamma offset 373)   *EOPNOTSUPP*
   9::(rmdir /gamma)   *ENOTDIR*
  10::(unlink /gamma)   *success*
  11::(open /gamma RDWR(2))   *ENOENT*
  12::(open /gamma RDWR|O_APPEND(1026))   *ENOENT*
  13::(open /gamma O_RDONLY|O_CREAT|O_EXCL)  *success*
  14::(rmdir /gamma)   *ENOTDIR*
  15:: (creat /alpha)  = 2    *success*
  16::(idle_compact 0 0)   *success*
  17::(idle_compact 0 1)   *success*
  18:: (read 0 (399 bytes) /gamma)  *EBADF*
  19::(rmdir /gamma)   *ENOTDIR*
  20:: (write 0 479 /gamma) Wrote 479 bytes to FLASH
...
*******************************************
Scheduling reset in 1...
*******************************************
   195::(rename /delta/gamma/alpha /gamma)  *ENOENT*
   196::(read -9999 400 /delta/gamma/alpha)  *EBADF*
   197::write of page 7 block 1 failed on reset trap
(creat /delta/gamma/delta)
*******************************************
Reset event took place during this operation.
*******************************************
(mount)  fs_Block 4 bad -- hardware memory
*success*
  *ENOSPC*
Note:  Not comparing results/error codes due to reset.
Clearing file descriptors and open directories...
   198::(write -9999 320 /delta/gamma/delta)   *EBADF*
   199::(rmdir /delta)  *EROFS*
```

Figure 4. Portions of a typical test scenario

6. Otherwise, if $\mathcal{H}' = \mathcal{H}$, continue to the next operation in the test sequence.

4.3. Choosing Operations and Parameters

The details of selecting $\mathcal{P}(v)$ are complex, but the basic principles are somewhat similar to those suggested by Pacheco et al. for testing object-oriented libraries using feedback-directed random testing [13]. The results of past POSIX operations are used to extend the test in choosing random parameters (file descriptors and pathnames) for new operations — the use of feedback helps to strike a balance between choosing "interesting" operations likely to modify system state and thorough random testing for unexpectedly state-altering operations. Figure 4 shows part of a typical test run (for one configuration – to stress different functional scenarios, we make use of four different configurations, designed to produce different mixes of POSIX operations and faults.)

Choosing operations in this manner — on-the-fly, while keeping a loose history of generated files and directories, balances complete randomness (operations no human is likely to intentionally produce) with high probability of altering file system state. The details and exact probabilities, including such subtleties as how often to format the flash partition, are in no way optimal; they are an adaptation

to changing test results, an educated guess at an optimization problem impossible to solve (as we wish to optimize for *error-detection*, and do not have the computational resources to explore the space of strategies in any depth).

5. Design for Testability

The reference implementation made it possible to do large-scale randomized testing, by providing a (partial) test oracle. The ability to determine whether a test execution revealed any errors was only part of successful testing, however: possibly more important is the ability to actually "flush out" errors present in the software. Compliance with the POSIX-like interface enables the first; for the second requirement, we made a conscious effort to *design the system to be testable* [14].

5.1. Assertions

The heavy use of assertions [2] made it possible to reveal errors even when future actions of the test driver prevented a faulty system state from causing observable failure (consider a format chosen just after a file system corruption), or simply failed to reveal a failure as a mismatch with the reference system. The aggressive placement of assertions was critical enough to the development effort to be included in one of the authors' list of ten essential rules for coding safety-critical software, published in *IEEE Computer* [5].

5.2. Downwards Scalability

Errors are often caused by complex interactions between multiple resource and boundary conditions (often called the "corner cases"). Stress testing typically reveals some of these errors by exceeding expected limits in system resource use (memory, storage, etc.) or parameters, but often fails to explore the complex *interactions* of boundary conditions that programmers have failed to consider. Model checking and randomized testing are unlikely to reach the resource-exhaustions that can only be produced by long system executions. Model checking faces the state-space explosion problem, and randomized testing is unlikely to generate only resource consumptions without resource releases. One way to expose such errors is to change the resource limitations, by scaling the system downwards — for instance, for a flash file system, by testing on simulated hardware with *few blocks, few pages per block, and small page sizes*. The diminished size makes it possible to reach complex corner-case combinations of used/bad/free blocks and relative page positions with small test cases requiring few fault injections.

Experience with applying model-driven verification [6] on the flash file system used on a previous JPL mission revealed that some commercial systems impose hard limits on downwards scalability, failing to operate on unrealistically small hardware. In contrast, our file systems were tested from early stages on very small systems as well as real-life configurations. For testing, we typically used a configuration with 6 blocks of flash memory, 4 pages per block, and 200 bytes per page (32 bytes for header information and 168 for data) (see Figure 1) for most partitions, much smaller than the flight hardware (2048 blocks of 32 4-kilobyte pages). Smaller configurations made it difficult to perform more than a handful of operations before running out of space, while larger configurations were not as effective for exposing corner-case errors. It is always possible that errors exist that can only be detected on a larger flash configuration (or smaller). We rely on the "folk version" of the small model property, invoked in bounded model checking [1], protocol verification, and other fields: it is typically the case that correctness for a sufficiently large small configuration implies correctness for all larger configurations. In any case, the goal of testing is to expose bugs, not to prove correctness.

5.3. Designing System Behavior for Testability

In a few cases, the choice of system behavior was motivated primarily by testability requirements. In particular, the *rename* operation requires multiple directory updates, and introduces the possibility of duplicate entries in directories. Competing designs were proposed for dealing with the possibility of duplicates: some allowed the test driver to predict whether a rename operation interrupted by a system reset would result in a state prior to or after the rename. These designs were preferred because of their testability: they made strong guarantees about the behavior of rename, beyond those strictly required for correctness.

6. Automated Test-Case Minimization

The randomized approach was very successful in discovering errors during the early stages of testing. This very success posed a problem: on a typical day, testing would produce hundreds (or thousands) of failing test cases, representing several independent errors. Due to randomization, few (if any) of these test cases *succinctly* exposed a failure: most test cases contained hundreds of unnecessary operations, obscuring a small number of critical fault injections and file system operations. For example, between 10:00 AM and 8:00 PM, on February 22nd, 2006, the test driver generated 610 failing test cases, ranging in length from 12 steps to 1,011 steps, with average length of 520 steps. Delivering this mass of indigestible material to a developer would have been counterproductive.

The solution was to present only *minimized* test cases for inspection and debugging. As each failing test case was generated, delta-debugging [18] was applied to produce a *minimized test case*. Unfortunately, in early stages of testing, large errors resulting from complex causes often proved to contain an embedded instance of a different, simpler error. Because certain simple errors proved difficult to fix and obscured more important problems, we modified Zeller's delta debugging scripts to use simple heuristics for preserving the cause of error — e.g., requiring that the failing POSIX operation remain unchanged, and only considering assertion failures to be valid minimizations of assertion failures.

Hand inspection (based on searching for similar failing operations and/or asserts) was used to construct an approximate breakdown of the errors by their different causes. One minimized test from each set was delivered to the developer, and the original test cases (and minimizations) were re-executed after fixes for the proposed errors were delivered. In some cases, it was necessary to re-minimize a long test case because the first minimized version no longer failed due to the fixes for other errors. The process was iterated until all original and minimized test cases succeeded.

For the February 22nd tests, the final set of representatives derived from the minimizations of the 610 test cases consisted of 17 tests, ranging in length from 1 step to 93 steps, with an average length of 30. In all cases these tests were shorter than the shortest test in the original set of 610 exhibiting the same failure. The average length of a minimized test case for the full February 22nd set was 39 steps — a 92.5% reduction over the original test cases. Our experience confirms the results of previous research indicating that automatic minimization is critical in randomized testing [8]. Performing a similar classification by hand over the *original* set of test cases would have required much more developer effort, as the operational patterns indicating different failures would have been difficult or impossible to extract from the noise of other operations. We expect that minimization would also assist automated statistical error classification approaches [19].

7. Reuse of the Test Framework

In addition to the flash file system, we have been developing two other systems for potential mission use. The first of these is a RAM file system, similar in interface to the flash file system, but operating on system memory. This system poses an interesting challenge, since one of the requirements is to be reliable across a "warm" system reset. In a warm reset, the software is restarted, and all data (and code) on the program stack is cleared, but any data on the heap (which stores the contents of the RAM file system) is not cleared, though it may be corrupted (for instance, the last word being written at the time of the warm reset may have not been written completely). In this case, the memory for the file system needs to be recovered, even after the software has been terminated and re-started in mid-operation. The second effort is to produce a low-level interface for manipulation of raw data on the flash memory, accessed as an array, rather than as a file system.

In both cases, we reused the framework developed for testing the flash file system. Our experience has been that initial efforts to develop an effective test system pay off in re-use on similar projects. The significant differences (fault model and injection in the case of the RAM file system, and reference suitability in the case of the low-level flash interface) were less important than the similarities. For the RAM file system, the strategy adopted for simulating system resets in the flash file system was not available: "writes" are modifications to main memory, assignment statements and `memcpy` calls in C. Replacing all such calls with macros or function calls would decrease code readability and reduce system performance. We used CIL [12] to rewrite the source to instrument each global memory access with a call to a function that checked for reset (and decremented a reset counter if reset was pending). The `check_for_reset` function optionally corrupted the location of the last write and returned control (via `setjmp`) to the test driver, in the event of a reset trap. Overhead was less than 1% for this instrumentation. For the low-level interface, we implemented our own array-based reference implementation. Our hardware simulation layer was reused in model-driven verification of the low-level storage module using SPIN [6] (model-checking exposed a subtle flaw undetected by our randomized testing efforts).

Adapting our infrastructure to the (non-POSIX) interface of another flash file system at the request of a flight project was a relatively low-effort task, and the test framework has demonstrated the ability to detect important problems with this *independently-developed* file system, for which we have not received source code. This experience increases our confidence that the framework developed may be generalized to other testing applications.

8. Errors Exposed by Testing

No experience report would be complete without some discussion of the kinds of errors discovered. The file system errors exposed during our testing can be grouped into three categories, in order of increasing criticality: POSIX divergences, errors in handling hardware faults, and major file system integrity losses. Figure 3 shows a "history" of the defects discovered during testing. Our final regression set consists of 255 test cases.

8.1. POSIX Divergences

The first week of testing exposed a number of incorrect POSIX error codes. The only difficulty in resolving these was in differentiating between genuine errors and cases where POSIX allowed multiple possible return codes. These errors were very quickly eliminated, and during the remainder of development few were re-introduced (in two occasions, a rewrite of parameter-checking code did produce a new POSIX divergence).

8.2. Fault Interactions

Hardware fault interactions that did not compromise file system integrity were a heavily-populated category of errors, responsible for the majority of errors between the fourth week of testing and the 13th week. Most of these errors consisted of the file system incorrectly managing its internal list of bad blocks — either attempting to write to a bad block or failing to use space on a good block due to an erroneous assumption that it was bad. This functionality was the source of considerable difficulty in testing, as it is not part of the reference system.

8.3. File System Integrity/Functionality Losses

A substantial number of the errors discovered involved a loss of file system integrity or functionality. These errors were almost all very low-probability scenarios, involving well-placed system resets and precise flash configurations, but of very high severity. We estimate the probability of discovering any of these errors using standard JPL hardware testbed procedures to be very slim indeed. Scenarios included complete loss of the file system, loss of file contents (for a file not involved in the operation causing the error, in some cases), null pointer dereference, inability to unmount the file system, and abortion of file system operation due to an incorrect assertion about system state.

These errors justify the testing effort — at a low cost, critical errors in implementation or design were exposed that would likely have been impossible to detect by more conventional testing approaches. Even if certain errors could have been discovered in the testbed, a very subtle, low-probability error exposing design issues may be left in place as a known problem. The cost of fixing the error would be so high, and the uncertainty introduced by a late-stage design modification so great, that the error would likely be deemed "unfixable." Our randomized testing framework exposed over 100 such faults during the test period. While some of these would have been revealed by normal testing, and some would have been discovered during review of the design, we believe that a large number would have made their way into delivered software, lurking as potential threats to mission success.

9. Test Coverage

Over our current regression set, statement coverage of the file system ranges from a low of 62.60% on the file containing "extra" functionality not covered by mission requirements to 89.06% coverage on the file containing core algorithms for manipulating pages on flash, the most critical component of the system. Coverage over 10,000 random tests produces lower coverage numbers, as would be expected (20% lower for non-core functionality, but only 3% lower for page manipulation) . Coverage for 100,000 random tests is roughly unchanged from that for 10,000 random tests.

We are not particularly concerned with these coverage *numbers*. Our interest in coverage focuses on the actual source statements covered. In particular, we performed hand reviews of the coverage measures for each statement of source code, and established that all statements not covered fell into two categories:

- **Defensive coding:** Most non-covered code has the structure shown in Figure 5. Certain conditions are not expected to occur during execution, but cannot be *proven* impossible. During testing, assertions detect these violations of expected behavior. During operation, it is not reasonable for the file system to simply abort operation; the file system must take action to prevent file system corruption and return an error indicating the anomaly. In every such case not covered during testing, *failure of the assertion guard would indicate an error in the file system, or at least in our understanding of system invariants*. Lack of coverage is therefore a sign of reliability. We do plan to cover the "impossible" code via software fault injection in order to ensure that the defensive actions work properly, but this is a low priority sanity check.
- **Trivial parameter checking:** The remainder of the code not covered during random testing consists of checks for minimal requirements on input parameters to POSIX calls (e.g., that the read buffer is not NULL or a request for a negative number of bytes, as in Figure 6). Producing trivially incorrect input parameters is a bad use of randomized testing: the behavior of the code in such cases does not depend on the system state and does not change the system state. Whether testing or model checking, introducing transitions that do not alter state is an ineffective approach to bug-hunting.

  Of course, "trivial" parameter checking is occasionally incorrect, missing, dependent on system state, or capable of modifying system state. We plan to use bounded

```
531914:  780:  if (!FS_ASSERT((dp->type & FS_G) == FS_G))
 #####:  781:     {       fs_handle_condition(dp->type);
 #####:  782:             FS_SET_ERR(EEASSERT);
     -:  783:     }
```

Figure 5. Defensive coding in the file system

The numbers on the left indicate statement coverage over a large set of randomized tests. In this case, the conditional on line 780 was executed over 500,000 times, but was never satisfied (0 is indicated by ##### in the tool to simplify searching for un-executed lines).

```
15007634: 1844:  if (want < 0 || b_in == NULL)
   #####: 1845:  {       fs_i_release_access(Lp);
   #####: 1846:          FS_SET_ERR(EINVAL);
   #####: 1847:          return FS_ERROR;
       -: 1848:  }
```

Figure 6. Trivial parameter checking in the file system

model checking [7] or other static analysis to discharge the assumption of triviality for each such parameter check.

10. Test Status and Conclusions

As noted in our overview, testing continues. To date, we have a sequence of over 3.5 billion operations with no detected divergences from expected behavior. This exceeds expected mission lifetime use by a considerable amount. We plan to continue testing. Unfortunately, no amount of randomized testing can establish software correctness. This leads into our plans for the future, which build on the (minimal) confidence in correctness established by our testing efforts.

10.1. Future Work

The most important future work, from JPL's perspective, is that we will continue to test mission file systems, including a likely re-design of our implementation to meet requirements for smaller memory footprint for large flash devices, and will extend our efforts to other software development efforts where differential testing is possible. More interesting from a research perspective are efforts to use the testing framework as a basis for more formal verification.

The "technology transfer" effort would require the definition of a language for generating test inputs, a framework for comparing aspects of tested and reference system behavior, and an easy-to-use method for handling fault-induced differences. An early effort in this direction relies on generating programs to generate tests, a convenient framework for the developers who are most likely to use such a system.

As formal verification researchers, our primary interest at this point is to go beyond the reliability established by randomized testing and improve our confidence that the file system is actually correct. Model-driven verification is a natural next step: in a sense, model driven verification can be seen as systematic testing guided by state-space coverage. For large state-spaces such as the file system, complete coverage may be impossible, but coverage of well-designed state-space abstractions can provide much more confidence in correctness than random exploration. Our experience with model checking and testing the low-level flash storage module suggests that considerable effort can be saved by reusing nondeterministic drivers for both purposes, and confirms our suspicions that model-driven verification may find some errors that escape randomized testing. The ability to detect defects (not) discovered during randomized testing will serve as a useful measure of abstraction effectiveness, and the simulation layer will be directly re-usable. Our final goal is to use our experience to help others make a smooth transition from low-effort randomized testing in early development to rigorous abstraction-based model checking and the use of theorem provers and bounded model checkers to establish invariants before use in mission-critical situations.

References

[1] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207, 1999.

[2] L. A. Clarke and D. S. Rosenblum. A historical perspective on runtime assertion checking in software development. *ACM SIGSOFT Software Engineering Notes*, 31(3):25–37, May 2006.

[3] J. Erickson and R. Joshi. Proving correctness of a Flash filesystem in ACL2. Unpublished manuscript in preparation, 2006.

[4] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.

[5] G. J. Holzmann. The power of ten: Rules for developing safety critical code. *IEEE Computer*, 39(6):95–97, June 2006.

[6] G. J. Holzmann and R. Joshi. Model-driven software verification. In *SPIN Workshop on Model Checking of Software*, pages 76–91, 2004.

[7] D. Kroening, E. M. Clarke, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176, 2004.

[8] Y. Lei and J. H. Andrews. Minimization of randomized unit test cases. In *International Symposium on Software Reliability Engineering*, pages 267–276, 2005.

[9] W. McKeeman. Differential testing for software. *Digital Technical Journal of Digital Equipment Corporation*, 10(1):100–107, 1998.

[10] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 105(33(12)):32–44, 1990.

[11] M. Musuvathi, D. Park, A. Chou, D. Engler, and D. Dill. CMC: A pragmatic approach to model checking real code. In *Symposium on Operating System Design and Implementation*, 2002.

[12] G. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *International Conference on Compiler Construction*, pages 213–228, 2002.

[13] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *International Conference on Software Engineering*, 2007. To appear.

[14] B. Pettichord. Design for testability. In *Pacific Northwest Software Quality Conference*, October 2002.

[15] G. Reeves and T. Neilson. The Mars Rover Spirit Flash anomaly. In *IEEE Aerospace Conference*, 2005.

[16] Various. A collection of NAND Flash application notes, whitepapers and articles. Available at `http://www.data-io.com/NAND/NANDApplicationNotes.asp`.

[17] J. Yang, P. Twohey, D. Engler, , and M. Musuvathi. Using model checking to find serious file system errors. In *Operating System Design and Implementation*, pages 273–288, 2004.

[18] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

[19] A. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken. Statistical debugging: Simultaneous identification of multiple bugs. In *International Conference on Machine Learning*, 2006.