Putting Flight Software Through the Paces with Testing, Model Checking, and Constraint-Solving

Alex Groce[1], Gerard Holzmann[1], Rajeev Joshi[1], and Ru-Gang Xu[2]

[1] Laboratory for Reliable Software, Jet Propulsion Laboratory
California Institute of Technology
[2] Department of Computer Science, University of California, Los Angeles

The research described in this publication was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Funding was also provided by NASA ESAS 6G.

Abstract. Fully automated complete verification methods, whether based on constraint solving or other approaches, have not, in our experience, with typical resource limitations, scaled to verification of rich properties of complex software systems such as the flash file systems used in space missions at JPL. Incomplete verification based on aggressive testing serves as our basis for checking functional correctness of the flight software systems we develop, with more heavyweight model checking (and static analysis) reserved either for simpler properties or smaller modules of the system. For more complex properties of programs with complex data structures, it is possibly more beneficial to use constraint solvers to guide execution (i.e., testing, even if performed by a model checking tool) than to translate the program and property into a set of constraints. We describe efforts to verify spacecraft flight software (specifically flash file storage modules) at the Jet Propulsion Laboratory, giving an overview of our experience with practical application of known methods and development of new methods for ensuring the correctness of mission-critical code.

1 Introduction

1.1 Background: Spacecraft File Systems

In January of 2004, the Jet Propulsion Laboratory, NASA, and the world celebrated the landing of the first of two Mars Exploration Rovers. The science that followed the celebration of an engineering triumph was interrupted 18 Martian days later (on "Sol 18") when Spirit abruptly stopped communications with Earth [33]. Over the next few days, Spirit occasionally resumed contact with JPL, but these brief sessions were often mysteriously cut short. JPL's software and fault protection team used the pattern of communication attempts and limited telemetry to trace the problems to a cycle of reboots. Spirit was encountering "fatal" errors either during or just after initialization. On Sol 21, the JPL team

commanded the craft (now running low on battery power, after failing to properly shut down for each Martian night) to go into "crippled" mode, operating without access to the flash file system.

The reboots were due to an unexpected interaction between the flash file system and the core flight software: the file system allocated memory at boot time based on the number of files on the flash storage, *including deleted files*. Files left over from landing and new science and engineering data generated during the 18 sols of normal operations required more memory than was available on the rover, a "fatal" fault resulting in a reboot. After recovering as much data as possible, the JPL team re-formatted the flash device, and Spirit returned to its scientific mission. The full story, presented in detail in an IEEE Aerospace paper by Reeves and Neilson [33] is a classic example of software detective work, high-stakes debugging of a system over 55 million kilometers away.

The Spirit anomaly was not, strictly speaking, due to a bug in either the flight software or the file system. However, the Spirit experience and other (less public) incidents with other file systems convinced JPL's flight software engineers that commercial flash file systems were not ideal for mission critical use. In recent missions, JPL has increasingly relied on flash memory devices [35] to store critical data, as flash uses little power or mass and has a high information density — making it ideal for space mission use. For convenience and flexibility, most of this data has been stored in hierarchical file systems. The data stored is often irreplaceable (e.g., landing telemetry or images of impact with a comet), so it is essential that flash file systems provide high reliability for space missions.

A NAND flash device consists of a set of *pages*, divided into larger units called *blocks*. The basic operations on a NAND device are: *write* a 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, but pages must be erased at the block granularity. Flash file systems must therefore manage invalid and outdated pages and perform garbage collection, rather than relying on overwriting old data. The combined requirements of managing pages, ensuring reasonable wear-leveling of page writes (each page has a limited life cycle), preserving atomic operation across hardware resets, and responding to hardware failures requires a fairly complex implementation. Verifying that an implementation meets these requirements is highly non-trivial.

The Laboratory for Reliable Software (LaRS) [1] took on the task of building an in-house flash file system with static memory allocation and rigorously tested reliability across system resets and flash hardware failures. LaRS has also proposed a verified (flash) file system as a mini-challenge [25] in response to Hoare's grand challenge of a verifying compiler [20].

1.2 Proof, Analysis, and Testing

Proof In an ideal world, we would have first proved our design correct and then proved that our implementation embedded that design in C code. In practice, a full refinement-based proof from the ground up proved impossible given our

resource limitations, time constraints, and shifting requirements and hardware behavior. Our efforts to use ACL2 to prove an initial design correct [12] influenced later designs but never amounted to a basis for confidence in the system — and connecting the proof artifacts to the flight software implementation would have been a challenge even given a good design-level proof. Ongoing research projects at various laboratories aim at making user-aided proof a more realistic possibility in development situations such as ours.

Analysis As readers would expect, we routinely apply the usual array of static analysis tools to our code, including Coverity [2], Klocwork [3], Code Sonar [4], Uno [21], and some hand-made checkers implemented in the CIL framework [30]. The properties checked by traditional static analysis are unfortunately quite limited — establishing reliable data storage across system resets requires much more than the absence of null pointer dereferences. We also attempted to use a bounded model checker and model checkers that abstract a model from source code on portions of the source code, as described below. In our experience, these tools, while theoretically capable of providing automated verification of richer properties, did not scale to either our code or our properties.

Testing The realities of flight software development did not prevent us from aggressively applying other state-of-the-art verification technologies, including random testing, model checking, and constraint-based testing and model checking. None of these technologies, at least as we are using them, are capable of fully verifying the file system's correctness, even in a bounded sense. Our goal, instead, is to concentrate on decreasing risk and increasing confidence in reliability of the system, with respect to full functional correctness. Rather than limit analysis to simple properties for which more complete and automated methods might apply, we have concentrated on aggressive *testing* of file system operation: this paper describes a *bug hunt*, using the best technologies we know of for this purpose. This paper concentrates on the *dynamic* aspects of our efforts to find errors in the file system — in a sense, the title of this paper is misleading: we use both model checking and constraint solving not as exhaustive heavyweight alternatives to testing, but as aids to effective testing by program execution. Complete verification of the file system's correctness does not seem to be possible, without effort beyond our means, using current technology. Automated testing (in some cases via model checking), however, has revealed hundreds of errors, including very subtle faults that would certainly have escaped code inspection or traditional testing. Proof-of-correctness remains beyond our power, given resource and time limitations, at this point, but our experience shows that automated methods for finding errors in programs (including static analysis tools, though these are not the focus of this paper) have finally attained a promising maturity.

In previous papers [14–16, 6, 23] we have described the more technical aspects of our approach to file system testing and the research results inspired by this effort. In this paper, we present an overview from a more practical point of view: why did we select these test approaches, and how well did the various

	Method							
Module	Proof	Static	Random	BLAST	MAGIC	CBMC	SPIN	Splat
NVFS1(MSAP)	P/F	S	S	F	F	F	-	-
NVFS2(MSAP)	P/F	S	S	F	F	F	P	-
NVDS(MSAP)	P/F	S	S	F	F	F	S	P/S
RAMFS(MSAP)	-	S	S	F	F	P	S	-
XFS	-	-	S	-	-	-	-	-
NVFS(MSL)	P/F	S	S	-	-	-	S	-
NVDS(MSL)	P/F	S	S	-	-	-	S	P/S
RAMFS(MSL)	-	S	S	-	-	-	S	-

- = Did not attempt to apply
S = Successful application (bugs discovered or properties proved)
F = Attempted application, failed to scale or caused tool to crash
P = Partial success — very limited results or application to small fragment of code

NVFS is the name for all JPL POSIX-like flash file systems. NVFS1 and NVFS2 are two independently coded versions for a multi-mission software platform (MSAP). RAMFS is the name for all JPL RAM file systems. NVDS is the name for all JPL low-level flash storage modules (not providing a hierarchical POSIX-like file system). MSL denotes module versions to be included in the Mars Science Laboratory flight software. XFS is a contractor-developed flash file system (with a non-POSIX interface) used in a planetary mission managed by JPL. Proof and static analysis were not applicable to "XFS" as we did not have access to design, requirements, or source code.

Table 1. Methods, modules, and results

approaches work? What factors seemed most important to test effectiveness, and what future directions seem most promising for increasing our confidence in file system reliability?

What we describe here is an on-going effort: we are now testing our file systems for the Mars Science Laboratory mission, scheduled to launch in September or October of 2009 [5]. This paper discusses that effort, and those that preceded it. We have tested three different implementations of a POSIX flash file system with hierarchical directory structure, one non-POSIX flash file system with hierarchical directory structure, three POSIX RAM file systems, and two low-level flash storage systems (essentially implementing an array with desirable atomic-write properties and bad-block management on flash storage). Table 1 shows a summary of the methods applied, file storage systems tested, and the experienced utility of the method in each case. Of course, these results may not be typical: we describe our experience, with one particular set of modules, with these approaches. The fine details of both the code and the application of the tools or methods are specific to our circumstances, and may not apply in other cases. Our research interests, our expectations, and our initial experiences also influenced the effort given to each approach: in some cases, success may have been a product of greater effort, and in other cases limited success or failure may

have partly been due to limited resource allocation. Every software testing effort is a resource-use optimization problem with limited information as to costs and rewards.

2 Random Testing

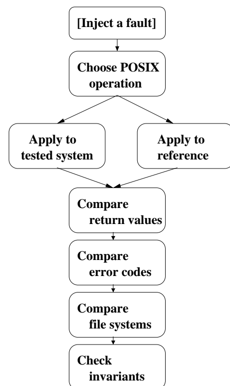

Fig. 1. Differential testing inner loop

Once we had a minimal running version of a flash file system, we developed a random testing [17] system, using a differential comparison [29] with a Linux file system as a test oracle [14]. Figure 1 shows the core of the differential test approach we used, which remains (with minor alterations) the heart of our test process (for both random testing and model checking) to this day. We chose random testing over model checking initially because we assumed (correctly, at the time) that the difficulties of engineering a model checking harness and backtracking the state of our file system and the reference file system would significantly delay the start of testing. As we discuss below, some of these engineering difficulties have been addressed by automatic code instrumentation, leaving random testing preferable (for our applications) only when backtracking the reference system is particularly difficult or when testing must be performed on a system with very limited memory. Hamlet argues that there are cases where "only random testing will do," but we believe that when applicable, model-driven verification (see below) is likely to be at least as effective — Hamlet notes that whether "only random testing will do" when compared with bounded exhaustive testing is essentially unknown [18].

We expected random testing to quickly expose many shallow bugs, especially POSIX error code conformance problems. We also expected that detecting most of the more subtle errors in the design would require more "intelligent" approaches, such as model checking. Random testing proved surprisingly effective

for both purposes, exposing dozens of subtle errors arising only in low probability states.

Keys to Successful Random Testing We attribute this surprising (to us) success to several factors. First, we avoided the primary difficulty of random testing and other automated testing approaches, the test oracle problem. Differential testing, when possible, makes it easy to concentrate most effort on choosing executions to run, rather than determining if those executions are correct.

Second, we used *feedback* [14, 31] to reduce the number of redundant and irrelevant operations randomly generated. An example of using feedback is to limit pathname choices to the set of pathnames provided as arguments to successful `mkdir` or `creat` operations (a set that would initially contain only the root path `/`), with the possibility of adding one additional random pathname component. E.g., if the history set of created paths was {`/, /a, /b`}, path choices would include the members of the set plus `/c`, `/d`, `/a/a`, `/a/b`, and so forth, but *not* `/a/b/c` or `/c/a`. The restriction is based on the observation that if the file system is correct, no POSIX operation can ever succeed on a path that is not of this form. We would not remove paths from the history when they are deleted from the file system, as the "resurrection" of dead files is a common fault. Of course, we cannot assume that our system is correct, but feedback biases the testing towards errors that seem plausible — informally, we can argue that a bug that causes the file system to incorrectly allow an operation on a pathname with a completely invalid prefix would require very peculiar pathname processing code.

Finally, all of our testing and model checking relies on an early emphasis on design for testability [32, 14]. Testing code with many invariants and assertions and the ability to operate on unrealistically small configurations (making corner cases the common cases) is much easier than testing code without such observability and flexibility.

Lessons Learned in Random Testing Because random testing was only intended to be a stopgap measure, the first version of the test framework was hastily implemented and lacked a coherent architecture. We re-designed and implemented the system as part of a black box acceptance testing effort (at mission request) for a file system developed by an outside contractor. The improvement in adaptability and ease of debugging test code suggested that the cost of a slight delay in initial testing would have been wise to accept in order to improve the rest of the testing experience. It is, of course, conventional wisdom that "building one to throw away" [9] is often a wise idea, as it is difficult to know how best to build a system without experience with a prototype, but the temptation to dive into test code without any effort to make it maintainable is a serious threat to effective testing.

In particular, we found that making the language in which test cases were stored human-readable made it much easier to debug the system and the test harness. It was also very helpful to be able to automatically generate stand-alone C test cases to pass to the (non-JPL) developer of the file system.

We also discovered that the length of each random test is a significant factor in the effectiveness of the testing [6], with a change in maximum test length in many cases resulting in a one or two order-of-magnitude change in the number of failing test cases produced per test operation. Our experiences also confirmed the importance of minimizing [37] randomly produced test cases before debugging [28]. Minimization became much less important when we switched to model checking as our primary test method, as test cases were generally much shorter and irrelevant operations were often obvious to cursory inspection.

3 Model Checking

3.1 Bounded and Abstraction-Based Model Checking

We hoped to use model checking to fully verify certain critical components of the file system. We have considerable experience with bounded model checking and abstraction-based model checking, including the contributions to the implementation of some better-known tools for checking C code [27, 11]. We selected a small (50 line) function with no dependence on the larger code base as a case study. The function in question, given a string, *canonizes* the string as a pathname — removing extraneous "/" and "." characters and returning an error code if any illegal characters appear in the string or if it exceeds the maximum path length. To our surprise, the abstraction-based tools, including BLAST [19] and MAGIC either failed to extract a model from the code or failed to find a proof or a counterexample. Perhaps with assistance from the tool authors we might have been able to handle this small function, but JPL restrictions on releasing flight software made such an effort impractical. The bounded model checker CBMC [27] was able to verify the properties of interest up to a maximum path length of 6 characters, but the SAT solver timed out for larger bounds. We made limited efforts to apply BLAST and CBMC to other code fragments, but in general found that the tools did not scale to verifying interesting properties of file system code, and that slicing the code to push it through the tools was an unrewarding effort. The success of various groups in applying similar tools to low-level device driver software [7] suggests that our problems with more complete model checking may arise from the heavy use of more complex data structures in the file system: even involved manipulation of strings seems to pose a challenge for the abstraction-based tools.

3.2 Model Checking via Program Execution: Model-Driven Verification with SPIN

In a sense, most of our "model checking" efforts are closer to aggressive systematic testing with backtracking than to traditional model checking. We actually execute implementation code, rather than building a model or abstracting a model from source code, and we do not expect to explore the entire state space of the system. This *model-driven verification* approach [22] is based on two observations: 1) for critical applications, it is essential to test actual implementation

code and not just design models and 2) as noted above, for most real programs, complete verification of rich properties is not possible with current "complete" model checking technologies.

Model-driven verification with SPIN [24] relies on the fact that SPIN is a model-checker generator. Given a model written in the PROMELA language, the SPIN tool generates a customized explicit-state model checker written in C. In model-driven verification, PROMELA is extended to allow embedded fragments of pure C code, which are executed during transitions of the model. The PROMELA model (now essentially a test harness for a C program) includes information on the memory locations of the state of the C program, enabling SPIN to backtrack the running C program. The model checker runs the C program, providing inputs and choosing function calls as dictated by the nondeterminism expressed in the PROMELA test harness. When the model checker reaches a state that has been previously visited, it backtracks both the harness and the C program.

We used model-driven verification to check the pathname canonizer discussed above. After introducing a relatively obvious abstraction (limiting characters in the strings to the set of special characters plus one valid component character), we were able to verify the properties of interest for a much larger depth than with the bounded model checker CBMC.

We next applied SPIN to a low-level flash storage module we were developing for flight use in storing critical engineering parameters. We expected this to be easier than using SPIN for a full POSIX file system — writing a backtrackable C reference was a trivial exercise, and all parameters were small integers, rather than complex pathnames. To our surprise, model checking revealed only one interesting error that had not been detected by random testing.

In order to apply model checking to the full flash file system, we developed a suite of engineering tools for debugging test harness backtracking and checking properties inside C code [15]. This tool set improved the ease of model-driven verification to the point that we found it more convenient to work *only* with model checking, rather than building both a model checker and a random tester [16]. For the MSL project, all our file system testing has been based on model-driven verification with SPIN. This has forced us to abandon the use of a Linux file system as a reference, due to the difficulty of backtracking such a system. We instead compare the POSIX flash file system (NVFS) to both the RAM file system (RAMFS) and an independently developed RAM file system. In the first case, an advantage is that both systems are required to present identical interfaces, including error codes, making any divergence an important error. The second case serves as a semi-independent confirmation that we are not missing errors common to both NVFS and RAMFS. Of course, a more thoroughly tested and widely-used file system would serve as a better differential basis, but the alterations we have made to POSIX in order to suit spacecraft usage forced our original random tester to complicate test code in order to translate errors and call parameters into standard POSIX terms. We believe the power of model checking, in addition to the convenience and improvement in the simplicity of

the test framework outweigh even the known problems of what is (arguably) a case of N-Version programming [8].

Model checking has been quite effective at finding subtle flaws in the file system, as expected. Complete verification, even for very small flash configurations, has proven impossible: week long runs on a 32GB machine have confirmed that even after unsound abstraction, there are trillions of reachable states in the system. The size of the state space has forced us to implement feedback [14] in the model checking parameter selection and to rely on a number of unsound abstractions [15] to further reduce the state space. We have also come to rely on a diversified search approach, rather than monolithic model checking runs. While a SPIN run using bitstate hashing and a large memory array may run for many hours before detecting an error, we find that a series of independent runs with different orderings for nondeterministic choices and different search strategies will often reveal the same error in a matter of seconds [23].

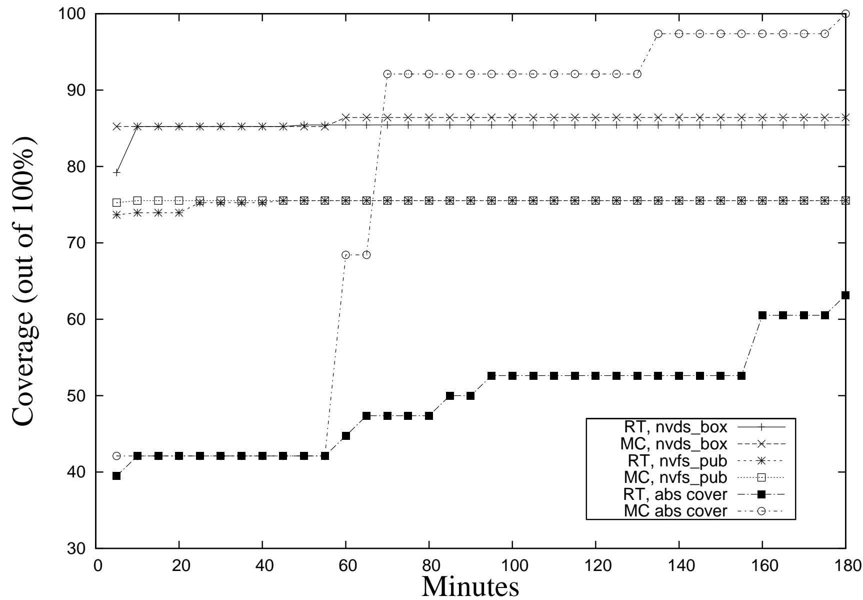

Fig. 2. Coverage for model checking and random testing compared

Model Checking and Random Testing in One Framework In our most recent versions of the test harness, we have integrated model checking and random testing [16]. We use a macro call for all nondeterminism in the SPIN model checking harness, and compile the model for either model checking or true random testing (a series of random walks through the state space). Our experiments so far confirm our suspicion that random testing is generally less effective at cov-

ering both source code and file system states than model checking, if more than an hour of compute time is available for testing. Figure 2 shows a comparison of random testing and model checking, using the same framework and probabilities for choices (recall that the model checking is incomplete) [16]. Two of the measures indicate source code coverage over modules of the flash file system, while the third shows coverage of an abstraction of the flash device state (the file system types of pages stored on the flash device). The graph shows how coverage increases as time for testing increases. Clearly, more time for exploration produces better coverage. In one source-cover coverage case, random testing and model checking both attain maximum coverage quickly. Coverage of a lower-level (and more state-based) module is initially better for random testing, but model checking begins to improve on those results at around the 50 minute mark and thereafter remains superior. The effectiveness of model checking for coverage is best shown, however, by the abstract state coverage: model checking covers 100 percent of the states that we know to be reachable (we cannot guarantee that this is true exhaustive coverage, as we are unable to complete model checking for this system), while random testing never visits more than 65 percent of those states, even after three hours of performing random walks.

4 Directed Testing via Constraint Solving

A fundamental criticism of random testing is that it is difficult to find "needles in the haystack" – when a branch is guarded by specific input values, the chances of randomly selecting those values (and thus exploring the branch) are often very low. Hand-tuning the ranges of random choice can address this problem in some cases, but reduces test automation and scales poorly. Moreover, when the guard depends on other inputs, or when various guards obligate different random bias functions, hand-tuning may be essentially impossible. One solution is to use symbolic execution and a constraint solver to produce inputs that satisfy guards [26]. Unfortunately, this approach is limited to the rare cases in which all expressions in guards are suitable for constraint solving. In particular, pointer dereferences, operating-system calls, hash value computations, and other "difficult" expressions tend to defeat the constraint solver and thus the symbolic execution engine.

Directed random testing combines random testing with symbolic execution to avoid this problem [13]. Expressions that cannot be handled symbolically are reduced to concrete values (taken from a particular execution) before calling a constraint solver. The "static" symbolic evaluation is assisted by the results of dynamic execution.

Recent work in directed testing has shown that path enumeration with a fixed sized input is effective in uncovering bugs and exploring branches that are extremely unlikely to be found with pure random testing [13, 34, 10]. Each element of the fixed sized input is represented by a symbolic value. The input is symbolically executed as the program is run. At each branch, the predicate representing whether the branch has been or not taken is noted. The conjunc-

tion of these predicates over the symbolic input (called the the *path constraint*) represents an unique path within the program. To generate a new unique path, one predicate in the path constraint is negated. The solution to the modified path constraint, generated by a constraint solver, yields a new input that will follow a different path. Repeating this procedure over all possible branch points results in enumeration of all paths.

We applied the directed testing tool Splat [36] to the pathname canonizer, and observed much better scalability than with the bounded model checker CBMC. Of course, the results are not directly comparable: CBMC explores all possible executions (including data values) up to a bounded depth of loop unrollings, while Splat explores all control flow paths (with the limit defined by the size of symbolic inputs). In the case of the canonizer, we hypothesize that complete path coverage without error essentially guarantees correctness of the code. For the CBMC limits, Splat requires only 2 seconds to generate 137 paths. Increasing the maximum path size (equivalent to the loop bound in this case) to 12 characters Splat needs a little over an hour to generate 36,857 paths.

We also applied Splat to NVDS, the low-level storage module of the file system. Initially, influenced by the successful experiments performed with EXE on Linux file-systems [10], we defined the input as an 504 byte buffer that represented the smallest formattable flash memory: 3 blocks of 3 pages per block with 56 bytes per page. This 504 byte buffer was used to enumerate paths in the `mount` function followed by a `write` operation. Splat generated 79,548 "flash volumes" over a period of 13 days (on a 2.8 GHz P4 with 1 GB of RAM). However, none of these disks were mountable: although many paths leading to a failure in the `mount` function were explored, the `write` operation was never called. We were facing a problem analogous to the state space explosion problem in model checking: a path explosion problem. The number of ways to fail to mount a flash volume effectively hid the few paths leading to a successful `mount` and the possibility of a `write`. Of course, any path might reveal an unknown error — but, practically speaking, we were not finding new bugs or improving on the coverage results for model checking or pure random testing. We investigated a number of approaches to path abstraction or pruning, but the various algorithms proposed resulted in the loss of many of the benefits of aiming at complete path coverage. At this point, we reconsidered our definition of the test input.

Our second attempt to apply directed testing was much more successful, and revealed previously undetected arithmetic overflow bugs in the `read` and `write` operations. The 32 byte input buffer represented parameters to three `write` operations and a `read` operation that followed a `mount` of a *freshly formatted volume*. Splat quickly generated an input that caused a buffer overrun due to an arithmetic overflow in bounds checking. After these arithmetic overflow bugs were fixed, all paths were generated within an hour. The overflow bugs were not detected by model checking or random testing. In both cases, we had limited the range of inputs to "reasonable" choices, based on the maximum file size for the flash volume. Simply adding an additional choice, rather than enlarging the range, would not have sufficed to find the error: the buffer overrun required the

values to overflow into a specific range. Obviously extending the range of inputs to the full 32 bit values would result in very low probability of performing any interesting operations (or finding the overflow) for random testing, and create a prohibitive number of successors states for SPIN to explore in model checking.

Unfortunately, after revealing this difficult-to-find error (since discovered in various other systems, after we knew the pattern of parameter-checking to look for), directed testing with a small operation set revealed no new errors not detected by model checking or random testing. Increasing the number of operations rapidly increased the time before completion without an equivalent increase in code or abstract state coverage.

5 Work in Progress: Using a Constraint Solver to Select an Initial State

We are now experimenting with an alternative use for constraint solvers in testing: using a constraint solver to find an initial state for the system, and then starting model checking (or random testing) from that initial state. Rather than relying on path coverage to produce a large set of starting flash volumes (with the attendant difficulties noted above), we intend to rely on developer/tester definition of interesting starting states, i.e, "states in which the flash volume is almost full" or "states in which there are two bad blocks with valid data." The key motivation is to begin model checking from deep states that may not be easily reached during the depth-first search, even with search diversification. Ideally, constraints define system states that are both hard to reach and likely to be near (as measured in the number of operations that must be applied) to states exposing a fault in the file system.

The novelty of the approach is that our constraints are defined by executable C code, including a `rep_ok` function to determine if a volume embeds a valid file system and a series of abstraction functions that take a concrete volume and produce an abstraction. We use CBMC [27] (and a SAT solver, called by CBMC) as our "constraint solving" engine. The advantage of this approach is that a developer can write invariant-checking functions and abstract coverage functions, then use these functions to guide model checking. The approach allows us to stage generation of concrete states — we can use CBMC to find an abstract state matching a specification, then use faster, more scalable hand-coded generators to produce random concretizations of the abstract state. For example, CBMC may only determine the type of each flash page, and a second tool may populate pages with random bytes. In general, for smaller flash configurations this staging is not required, but checking for resource-limit based errors may require larger flash volumes than CBMC can directly handle.

Our current implementation of this constraint-based approach works within a SPIN test harness for the file system: the harness calls CBMC to generate an initial flash configuration, embedding a specified structure of files and directories (we have modified CBMC to produce counterexamples as executable C code fragments that assign values to variables).

Unfortunately, evaluating the utility of this method is difficulty, as it is not a fully automated approach. The value of the initial states lies in the skill of the developer or tester in finding deep corner cases that are not easily generated by model checking or random testing.

6 Conclusions

At this time fully automated verification methods, whether based on constraint solving or other approaches, do not, in our experience, easily scale to verification of rich properties of complex software systems such as flash file systems. Verification approaches more akin to aggressive testing, with more guidance by the tester or developer than push-button model-checking, have served as the basis for checking functional correctness of our software modules, with more heavy-weight model checking (and static analysis) reserved either for simpler properties or small modules of the system. For the more complex properties of programs with complex data structures (that is, programs and properties relying on more than control structure and simple integer relationships), we believe it may be, at present, more practical to use constraint solvers to guide execution than to translate the program and property into a set of constraints. Given more resources and time (and better tools), user-assisted proof would be the ideal approach for ensuring correctness, but did not prove feasible in our circumstances.

References

1. http://eis.jpl.nasa.gov/lars.
2. http://www.coverity.com.
3. http://klocwork.com.
4. http://http://www.grammatech.com/products/codesonar.
5. http://mars.jpl.nasa.gov/msl.
6. James H. Andrews, Alex Groce, Melissa Weston, and Ru-Gang Xu. Random test run length and effectiveness. In *Automated Software Engineering*, 2008.
7. Thomas Ball and Sriram Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN Workshop on Model Checking of Software*, pages 103–122, 2001.
8. Susan Brilliant, John Knight, and Nancy Leveson. The consistent comparison problem in n-version software. *IEEE Transactions on Software Engineering*, 15(11):1481–1485, 1987.
9. Frederick Brooks. *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition*. Addison-Wesley Professional, 1995.
10. Cristian Cadar, Vijay Ganesh, Peter Pawlowski, David Dill, and Dawson Engler. EXE: automatically generating inputs of death. In *Conference on Computer and Communications Security*, pages 322–335, 2006.
11. Sagar Chaki, Edmund M. Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. In *International Conference on Software Engineering*, pages 385–395, 2003.
12. John Erickson and Rajeev Joshi. Proving correctness of a Flash filesystem in ACL2. Unpublished manuscript in preparation, 2006.

13. Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Programming Language Design and Implementation*, pages 213–223, 2005.
14. Alex Groce, Gerard Holzmann, and Rajeev Joshi. Randomized differential testing as a prelude to formal verification. In *International Conference on Software Engineering*, pages 621–631, 2007.
15. Alex Groce and Rajeev Joshi. Extending model checking with dynamic analysis. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 142–156, 2008.
16. Alex Groce and Rajeev Joshi. Random testing and model checking: Building a common framework for nondeterministic exploration. In *Workshop on Dynamic Analysis*, 2008.
17. Richard Hamlet. Random testing. In *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.
18. Richard Hamlet. When only random testing will do. In *International Workshop on Random Testing*, pages 1–9, 2006.
19. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Principles of Programming Languages*, pages 58–70, 2002.
20. Tony Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, 50(1):63–69, 2003.
21. Gerard Holzmann. Static source code checking for user-defined properties. In *Conference on Integrated Design and Process Technology*, 2002.
22. Gerard Holzmann and Rajeev Joshi. Model-driven software verification. In *SPIN Workshop on Model Checking of Software*, pages 76–91, 2004.
23. Gerard Holzmann, Rajeev Joshi, and Alex Groce. Tackling large verification problems with the swarm tool. In *SPIN Workshop on Model Checking of Software*, 2008.
24. Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.
25. Rajeev Joshi and Gerard Holzmann. A mini-challenge: Build a verifiable filesystem. In *The Conference on Verified Software: Theories, Tools, Experiments*, 2005.
26. James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
27. Daniel Kroening, Edmund M. Clarke, and Flavio Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176, 2004.
28. Yong Lei and James H. Andrews. Minimization of randomized unit test cases. In *International Symposium on Software Reliability Engineering*, pages 267–276, 2005.
29. William McKeeman. Differential testing for software. *Digital Technical Journal of Digital Equipment Corporation*, 10(1):100–107, 1998.
30. George Necula, Scott McPeak, Shree Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *International Conference on Compiler Construction*, pages 213–228, 2002.
31. Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *International Conference on Software Engineering*, pages 75–84, 2007.
32. Brett Pettichord. Design for testability. In *Pacific Northwest Software Quality Conference*, October 2002.
33. Glenn Reeves and Tracy Neilson. The Mars Rover Spirit Flash anomaly. In *IEEE Aerospace Conference*, 2005.

34. Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *Foundations of Software Engineering*, pages 262–272, 2005.
35. Various. A collection of NAND Flash application notes, whitepapers and articles. Available at `http://www.data-io.com/NAND /NANDApplicationNotes.asp`.
36. Ru-Gang Xu, Rupak Majumdar, and Patrice Godefroid. Testing for buffer overflows with length abstraction. In *International Symposium on Software Testing and Analysis*, 2008.
37. Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.