Annals of Mathematics and Artificial Intelligence manuscript No.
(will be inserted by the editor)

Establishing Flight Software Reliability: Testing, Model Checking, Constraint-Solving, Monitoring and Learning

Alex Groce · Klaus Havelund · Gerard Holzmann · Rajeev Joshi · Ru-Gang Xu

the date of receipt and acceptance should be inserted later

Abstract In this paper we discuss the application of a range of techniques to the verification of mission-critical flight software at NASA's Jet Propulsion Laboratory. For this type of application we want to achieve a higher level of confidence than can be achieved through standard software testing. Unfortunately, given the current state of the art, especially when efforts are constrained by the tight deadlines and resource limitations of a flight project, it is not feasible to produce a rigorous formal proof of correctness of even a well-specified stand-alone module such as a file system (much less more tightly coupled or difficult-to-specify modules). This means that we must look for a practical alternative in the area between traditional testing and proof, as we attempt to optimize rigor and coverage. The approaches we describe here are based on testing, model checking, constraint-solving, monitoring, and finite-state machine learning, in addition to static code analysis. The results we have obtained in the domain of file systems are encouraging, and suggest that for more complex properties of programs with complex data structures, it is possibly more beneficial to use constraint solvers to guide and analyze execution (i.e., as in testing, even if performed by a model checking tool) than to translate the program and property into a set of constraints, as in abstraction-based and bounded model checkers. Our experience with non-file-system flight software modules shows that methods even further removed from traditional static formal methods can be assisted by formal approaches, yet readily adopted by test engineers and software developers, even as the key problem shifts from test generation and selection to test evaluation.

Alex Groce
School of Electrical Engineering and Computer Science, Oregon State University

Klaus Havelund, Gerard Holzmann, Rajeev Joshi
Laboratory for Reliable Software, Jet Propulsion Laboratory
California Institute of Technology
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.

Ru-Gang Xu
Department of Computer Science, University of California, Los Angeles

1 Introduction

1.1 Background: Spacecraft File Systems

In January of 2004, the Jet Propulsion Laboratory, NASA, and the world celebrated the landing of Spirit, the first of two Mars Exploration Rovers. The science mission that motivated the engineering triumph was interrupted 18 Martian days later (on "Sol 18") when Spirit abruptly stopped communicating with Earth [101]. 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 operation 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 [101] 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 rover's flight software or the file system specifically: both behaved as described, but (due to limited testing) the implications of combined behavior were not well understood. However, the Spirit experience and other—less public—bugs or problems with file systems convinced JPL's flight software engineers that commercial flash file systems were not ideal for mission critical use. This was a significant problem, given that in recent missions, JPL has increasingly relied on flash memory devices [106] to store critical data: 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 taken by the craft ejecting the crashing probe), so it is essential that flash file systems provide high reliability for space missions.

A NAND[1] flash device consists of a set of *blocks*, divided into smaller units called *pages*; the number of pages per block (and number of blocks) vary from device to device. The basic operations on a NAND device are: *write* a *p*age, *read* a *p*age, and *erase* a *b*lock. 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 rely on overwriting old data. The combined requirements of managing pages, ensuring reasonable wear-leveling of page writes (each block has a limited life

[1] NAND flash is named for the use of the logical NAND operation, as opposed to the NOR operation used in other flash systems.

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) [6] 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 [79] in response to Hoare's grand challenge of a verifying compiler [71].

1.2 Contributions

In previous papers [14, 53, 56, 59, 60, 76–78] we have described the more technical aspects of our approach to file system development and verification, and the research results inspired by this effort. In this work, we primarily focus on an overview from a more practical point of view: why did we select certain verification and testing approaches, and how well did the various approaches work? What factors appeared to drive effectiveness, and what future directions seem most promising for increasing our confidence in flight software reliability and correctness? We hope that in addition to providing other researchers insight into the current applicability of state-of-the-art techniques to critical software projects, this paper may also provide software practitioners with an understanding of how novel techniques emerging from the research community can be integrated into an overall assurance strategy for non-trivial development efforts. This paper expands on a detailed case study presented at the International Workshop on Constraints in Formal Verification in 2008 [57], placing the efforts in the context of the Mars Science Laboratory (MSL) [7] mission's larger-scale testing efforts and including further details of each approach we took. We also introduce a consideration of how to test the tester, in order to evaluate the condition of the test framework itself, as a prelude to suggesting a broad methodology for similar test efforts. Finally, various (more engineering-side) details omitted from research papers appear here, which may interest many readers. What we describe in this paper is an on-going effort: we are still testing file systems for the Mars Science Laboratory (Curiosity Rover) mission [7], which successfully landed on Mars on August 6th, 2012, and has operated successfully to the date at which we write this introduction (December 2nd, 2013). This paper discusses precursors to our current testing efforts, which are a refinement of the approaches discussed below. The bulk of this paper describes our efforts up until the temporary down-scaling of test efforts for MSL after the delay of launch from 2009 to 2011, with some discussion of work after that point.

1.3 Proof, Analysis, and Testing

We can divide our efforts into three broad classes: proof, analysis, and testing. By *proof* we mean the effort to use mathematical reasoning to formally prove the design and implementation of our system correct. We expect that proof may involve a significant manual effort, assisted by automated checkers. *Analysis* refers to a more automated effort to reason, usually about more limited properties of the design and implementation. We expect that analysis may be less precise than proof in that static analysis tools typically return some false positives, errors related to infeasible program

paths. We also include efforts at more complete symbolic model checking under analysis. Finally, in this paper *testing* refers to dynamic exploration of program behavior, including explicit-state model checking, which can be thought of as systematic testing with program state backtracking and a memory for visited states. We use testing to refer to all of our verification efforts that involve actually running the program.

1.3.1 Proof

In an ideal world, we would have first proved a high-level design (data structures and algorithms described in mathematical terms rather than low-level programming constructs) of a file system correct, and then proved that our implementation faithfully instantiated that correct 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.

We attempted to prove correctness of key parts of the file system design using the ACL2 theorem prover [48]. ACL2 is a powerful first-order theorem prover that can be used for complex proofs of recursive datatypes (such as we have in the file system) [2,80]. However, it is not a fully automatic prover, and requires considerable manual guidance. We spent around three months on a design-level proof, at the end of which we were able to show partial correctness of the function to create a directory. In all, this resulted in proving over 230 theorems in ACL2 (comprising just over 3200 lines of ACL2 input).

The main challenges encountered in the proof were that (a) many of the theorems required generalization, and finding this generalization was often nontrivial, and (b) proving that the file system always remained acyclic turned out to be difficult. One problem was that the file system design was not layered in a way that was suitable for proof. As a result, we needed to add auxiliary variables to various functions to help the prover in finding a suitable well-founded metric to show termination. This motivated us to reorganize the file system design so as to make such proofs easier in the future. These efforts to use ACL2 to prove an initial design correct influenced later designs but never amounted to a basis for confidence in the system. Additionally, connecting the proof artifacts to the flight software implementation would have been a challenge even given a good design-level proof (though, given that ACL2 designs are executable LISP code, we might have performed differential testing as a partial, non-proof-based response to this problem).

Ongoing research projects at various laboratories (e.g., work at NICTA [86]) aim at making user-aided proof a more realistic possibility in more resource-constrained development situations (even with changing specifications) such as ours.

1.3.2 Analysis

As readers would expect, we routinely apply the usual array of static analysis tools to our code, including Coverity [9], Klocwork [10], Code Sonar [4], Uno [72], and some hand-made checkers implemented in the CIL framework [95]. In addition to the checks traditionally performed by static analysis tools, we have attempted to statically show that code satisfies certain coding guidelines, such as containing only constant-bounded loops [74]. As a baseline for good code, and as the earliest and likely most cost-effective method for catching basic coding errors, static analysis tools are indispensable. 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; it requires functional correctness. We also attempted to use a bounded model checker [26] and model checkers that abstract a model from source code on portions of the source code. In our experience, these tools, while theoretically capable of providing automated verification for richer properties, did not scale to either our code or our properties. We consider these tools to provide analysis rather than proof, as they apply to a particular implementation and may only hold for one configuration (size of flash volume, etc.). Moreover, the properties considered are generally far less ambitious than full functional correctness, or equivalence to a mathematical reference model. In practice, we obtained useful results only when considering bounded executions of our implementation, rather than even limited property proof over a particular concrete implementation.

A related approach, in the space between analysis and full proof, is to use a language aimed at the integration of proof/verification and software implementation, e.g. SPARK [3], and prove at least some simple properties (for example, that certain exceptions cannot be produced during execution) with support from the implementation language and its development tools. In many cases, including ours, integration with, or inheritance from, an existing code base in another language, or available developer talent in an organization, makes this approach impossible. We could not have developed file systems for the MSL mission in SPARK; the mission software is exclusively C or C++ code.

1.3.3 Testing

The realities of a flight software development schedule did not prevent us from aggressively applying other state-of-the-art verification technologies, including random testing [65], model checking [37], and constraint-based testing and model checking. None of these technologies, at least as we are using them, are capable of fully verifying a file system's correctness, even in a bounded sense, or under "nominal usage" restrictions. Our goal is to concentrate on *decreasing risk* and *increasing (justified) confidence* in the reliability of the systems, with respect to full functional correctness: in other words, our goals are those of any effective testing effort. 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. 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 using current technology, without effort beyond our means. Automated testing (in some cases via model checking) has revealed hundreds of errors, including very subtle faults that would almost 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 attained a promising maturity.

We next provide a high-level definition of each of the approaches we will be discussing in more detail:

Table 1 Methods and their results when applied by LaRS at JPL to file systems.

	Method							
Module	Proof	Static	Rand	BLAST	MAGIC	CBMC	SPIN	Splat
NVFS1	P/F	S	S	F	F	F	—	—
NVFS2	P/F	S	S	F	F	F	P	—
NVDS	P/F	S	S	F	F	F	S	P/S
RAMFS	—	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	—
RAMFS2	—	—	—	—	—	—	S	—

- = Did not attempt to apply
S = Successful application (bugs discovered or properties proved)
F = Failed application, due to scalability limitations or bugs in the tool
P = Partial success: very limited results or application to small fragment of code
P/F = Partial success on design, failure for implementation-level
P/S = Partial success for full volume as input; success with operation sequence input

- **Random Testing:** In random testing, operations and parameters for a test are generated randomly. The random choices may be biased and depend on past history, but there is no attempt to systematically exclude already-visited program states or reach particular states.
- **Model Checking (Model-Driven Verification):** Model checking [37] is a systematic method for exploring a system's state space, in order to verify properties of the system (or produce an error trace if a desired property does not hold). Because the size of real-world state spaces makes full exploration impossible, model checking usually relies on either a symbolic representation of states (based on SAT or BDDs), an abstraction of the state space, or both of these methods. *Model-driven verification* [75] is an explicit-state model checking method in which a model checker acts as a test harness, choosing operations and parameters for calls to a program. The model checker tracks (some portion of) the state vector and backtracks when exploration reaches an already-visited state.
- **Directed Testing:** Directed testing [52] combines constraint-solving and random testing in order to systematically explore all paths through a program (and thus execute all branches). An initial path is generated randomly, and a constraint-solver is used to repeatedly find inputs to guide the program through new paths.
- **Monitoring and Learning:** In monitoring, a program execution is compared with a specification, to see if it violates any desired properties. Monitoring may be indifferent as to how the execution is produced (and thus combined with one of the above testing techniques). An additional application of monitoring is to *learn* the behavior of a system, rather than enforce desired behavior. In this case a model (typically some type of automaton) is generalized from several executions of the system that exhibit desired behavior.

What we describe in this paper is an on-going effort: we are still testing file systems for the Mars Science Laboratory mission. We have tested three different implementations of a POSIX-like (Portable Operating System Interface [for Unix]) [8] flash file system with hierarchical directory structure, one non-POSIX flash file system with hi-

erarchical directory structure, three POSIX-like 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). POSIX is a standard interface for file storage systems, providing a specification of behaviors for common file system functionality such as `open`, `close`, `read`, and `write` operations. For the purposes of this paper, the primary features of these systems are twofold. First, these systems feature a fairly complicated API with rich behavior in the case of the POSIX-like systems. Additionally, even the considerably less complex API and less rich behavior of the low-level systems results in an unexplorably large state-space.

Table 1 shows a summary of the methods applied, file storage systems tested, and the experienced utility of the method in each case. 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). NVDS is the name for all JPL low-level flash storage modules (not providing a hierarchical POSIX-like file system). XFS is a contractor-developed flash file system (with a non-POSIX interface) used in a planetary mission managed by JPL. RAMFS is the name for all JPL POSIX-like RAM file systems. MSL denotes module versions to be included in the Mars Science Laboratory flight software. Proof and static analysis were not applicable to XFS as we did not have access to design, requirements, or source code. RAMFS2 is a JPL-developed reference file system, not implemented for flight use. These test efforts overlapped, but the order shown in the table is roughly chronological.

Of course, these results may not be typical: we describe our experience, with one particular set of modules, with these approaches. We also report on preliminary efforts to apply more formal specification, learning of specifications, and monitoring to other aspects of MSL flight software. The fine details of both the code and the application of the tools or methods are specific to our circumstances. 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.

1.4 Related Work

The general literature of software verification, program proof, and particularly that of software testing [23, 12] is too well-known and extensive to be addressed here, but has obviously influenced our efforts at complete proof and effective testing. More automated verification of file systems and flash devices is a popular topic in the recent literature. Yang et al. have used model checking [113] and symbolic execution-based [112, 111] approaches successfully on Linux file systems. Some of this work uses CMC [94], a model checker that, like SPIN, directly executes C programs. Kim et al. focus on lower-level verification of flash device drivers, but with considerable overlap in the kinds of behavior being verified [84, 81, 82]; for example, they have shown that heavy use of assumptions may make a bounded model-checking approach with CBMC [87] more feasible than we have come to expect.

Random or stochastic testing dates to the early days of computer programming [65], and has occasionally been endorsed as competitive with partition testing [45, 64]. McKeeman noted that random testing was highly effective for fault detection when

combined with a powerful differential reference oracle [90]. Miller et al. employed a form of random testing to crash system utilities [92]. Recently, the merits of random testing have been more widely acknowledged [66], resulting in a large emerging body of work on sophisticated random testing tools [41,15,13,36,35,114] and key factors in the success or failure of random testing efforts [88,44]. Random testing often performs better than even approaches using sophisticated machine learning techniques to generate tests, when the time to generate each test is taken into account [54].

The earliest model checking algorithms [38] were applied to software, but it is, again, only in recent years that software model checking has emerged as a practically effective method [37]. The blurring of the distinction between model checking (exhaustive, using temporal logic) and testing (partial, typically employing assertions [39] and differential methods) has been a major theme of recent work in this field [69,24]. Godefroid's Verisoft anticipated this concept of a more dynamic and testing view of model checking, where code is actually executed (in place of a model extracted from code) [50]. Java PathFinder 2 [108], Bandera [40], Bogor [102], and CMC [94] typically preserved the possibility of exhaustive model checking, but enabled an approach more focused on bug-finding than full verification. Mercer and Jones showed that even machine code could be model checked using GDB [91]. Visser et al. implemented BET and random testing methods in the JPF2 model checker in order to compare testing approaches, making it particularly clear that explicit-state model checking was a form of software testing [109]. Qadeer and Rehof's context-bounded approach has applied the insights of bounded exhaustive testing to model checking concurrent systems [100], and served (with more traditional BMC) as an inspiration for some of our approaches to *downwards scalability*, a testing approach in which resources are limited to achieve "exhaustive" checks over smaller models. Dwyer et al. proposed exploitation of parallel searches similar to that in the swarm method [77] we employ [46][2].

Beginning with DART [52] testing researchers have explored numerous variations of a methodology combining constraint-solving with random testing [31,103,30,110]. In this work, an initial random execution is followed by tests guided by constraint-solving, with the goal of exploring new paths. The term "concolic" is used to describe this work, which uses *con*crete values from the previous execution whenever constraints produced by sym*bolic* execution are too difficult for a constraint solver to handle. Recent approaches have focused on controlling the path explosion problem [51,28], analogous to the state-explosion problem faced in model-checking. The reader will find further references to tools and approaches of particular relevance to our methods throughout the remainder of the paper.

There has also been substantial recent progress in the effort to generate complete correctness proofs of complex implementations written in the C language. A group at the Australian Information and Communications Technology Centre of Excellence (NICTA) proved correctness of the seL4 microkernel, consisting of around 8700 lines of C, by showing that it refines an abstract specification [86,105]. Their approach involved writing an executable specification in Haskell and proving that the C code was a refinement of the Haskell specification, using the prover Isabelle/HOL [96]. Another ongoing project is the effort to prove correctness of the Microsoft Hyper-V hypervisor [5], which consists of over 60,000 lines of concurrent, highly-optimized, C code. The project has developed a verifier (VCC) for concurrent C [11], which uses the automatic

[2] Swarm replaces a single model checking run with multiple runs using search diversity to explore more of an essentially infinite state space.

prover Z3 [42] to discharge proof obligations generated automatically from the source code. Further progress in this area might make it feasible to considerably extend the proof component of efforts such as that described in this paper.

1.5 Outline

We first present our three primary approaches to testing: random testing (Section 2), model checking (Section 3), and directed testing (Section 4). We briefly discuss a more speculative approach that combines constraint-solving and model checking (Section 5). In order to place our file system work in a larger context, we discuss the problem of testing the integrated MSL flight software, rather than a component in isolation; in particular, we briefly take a look at how, for larger scope test efforts, the problem of specification and test evaluation has tended to overshadow the question of which tests to execute (Section 6). We finally discuss the problem of a faulty test framework (Section 7), and propose a possible methodology for practitioners meeting challenges similar to those we face in the file system efforts (Section 8).

2 Random Testing

Once the LaRS development team produced a minimal running version of a flash file system, the dedicated test engineer developed a random testing [65] system, using a differential comparison [90] with a Linux file system as a test oracle [56]. One of the great challenges of automated testing is the oracle problem: how do we know what the results of an automatically generated test should be, without human inspection, in the absence of a complete formal specification? Differential testing avoids the oracle problem by performing all operations on (at least) two different systems that are supposed to satisfy the same (often informal or unknown) specification, and comparing the results. If the systems diverge in behavior for any inputs, it can be assumed that a flaw in at least one of the systems has been detected. Differential testing requires no complicated evaluation of temporal logic formulas, and can be implemented (usually as a simple set of assertions over outputs) just as easily for explicit-state model checking as for random testing. Differential testing is applicable to systems with slightly different specifications, so long as the intended differences are known and can be handled by an interface layer.

Figure 1 shows the core of the differential test approach we used, which remains (with minor alterations) the heart of our file system testing process (for both random testing and model checking) to this day. Again, the key point is that differential testing compares the results from performing the same operation on two systems with differing implementations. In the case of a file system, we compare the return values of function calls, the error codes set, and the actual resulting file systems (using the "query" functions such as `readdir` and `stat`). Faults (shown in square brackets because a fault is not injected for every operation) are only injected for the tested system, and require a specification of correct behavior with respect to a fault-free system. For file systems, this specification is often an atomicity model: no operation completes partially; when a reset or other fault arises, the observable file system state must either be unaltered or consistent with completion of the operation. Invariant checking, e.g. to ensure that directory structures are never cyclic, is an additional non-differential aspect of testing.

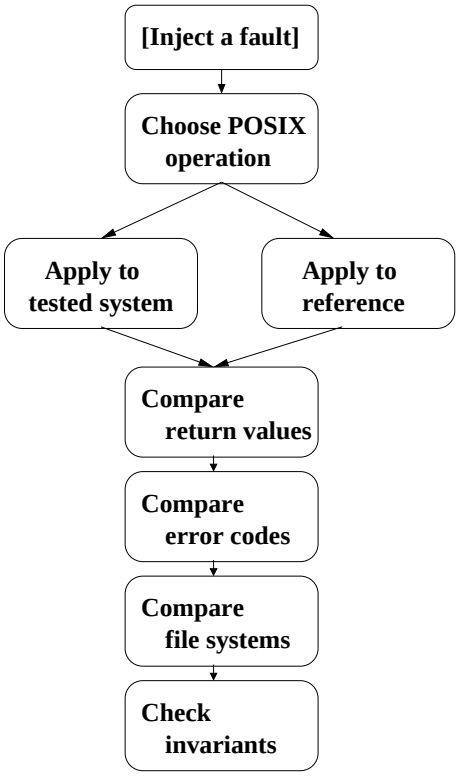


Fig. 1 Differential testing process: comparing two similar systems is a particularly effective test oracle approach for file systems.

In principle, most faults detected by checking invariants of the tested file system will eventually be exposed by differential testing, but it is always best for debugging to find errors as early as possible, and invariant checks often reveal faults many operational steps before differential outcomes appear.

We chose differential random testing over differential model checking initially because we assumed (correctly, at the time) that the difficulties of engineering a model checking harness and backtracking the state of the 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 our work on 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. Although Hamlet [66] argues that "only random testing will do" in cases of large structured input domains and persistent state, we believe that, when it can be applied, model-driven verification (see below) is likely to be at least as effective. Indeed, Hamlet himself does not argue that "only random testing will do" when compared with bounded exhaustive testing [3].

[3] A possible exception may be in cases where there is a need to randomly explore inputs with very large ranges (i.e., drawn from the full set of 32-bit integers) [60].

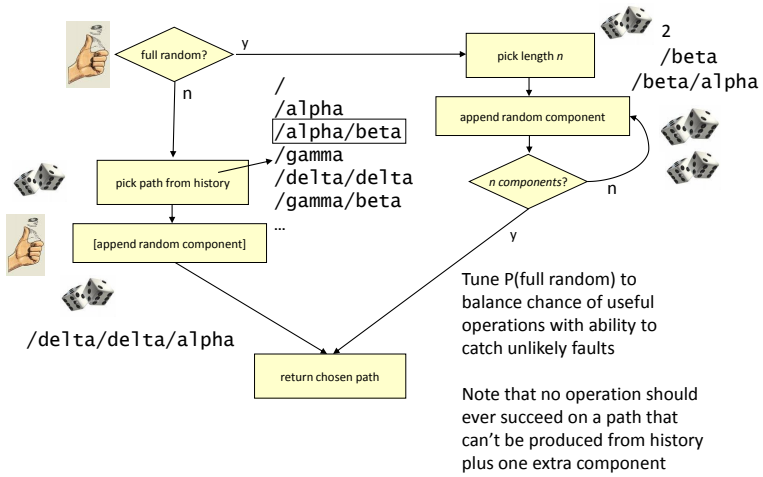


Fig. 2 Feedback approach for path name selection based on file system history: dice and coin flips indicate random selection of numeric/boolean values.

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 sophisticated approaches, such as model checking. Random testing proved surprisingly effective for both purposes, exposing dozens of subtle errors arising only in very low probability states, and producing a very compact, high-coverage, regression suite (composed of the 200+ minimized failing test cases for the full fault set found during testing) [56]. The errors detected included a checksum collision resulting from a warm-reboot taking place at the vulnerable point in a block `memcpy`, bad block information lost due to very complex sequences of writes and reboots, and numerous errors resulting from very low probability interactions between the `rename` operation and write failures. The sheer number of tests generated allowed us to find even very low probability events. Executing a million tests overnight ensures that software errors exposed in only one out of a million tests will almost certainly be found in a few nights of testing; it is only when error propagation is so constrained as to occur in only one in a billion or fewer tests that random testing becomes less effective.

2.1 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 effort on choosing executions to run, rather than determining if those executions are correct.

Second, we used *feedback* [56,98] to reduce the number of redundant and irrelevant operations randomly generated. In feedback, a weak model of system state is used to

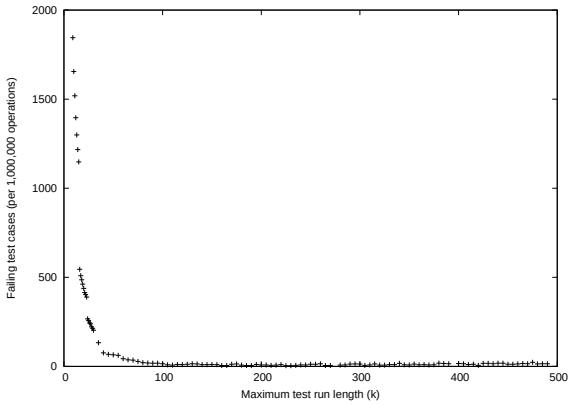
bias test operation and parameter selection, with the goal of reducing the high rate of invalid or redundant operations that random testing can produce (i.e., deleting files that do not exist, creating the same file over and over, accessing closed file descriptors). An example of using feedback is to 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 (Figure 2). 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 (any of which might result in another successful `mkdir` or `creat`), but *not* `/a/b/c` or `/c/a`—where a prefix of the path did not exist. 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 (with a prefix taken from a previous successful operation). 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, so we would prefer frequent attempts to access those files. Of course, we cannot assume that our system is correct, but feedback biases the testing toward 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 (e.g., some kind of embedded special case for certain path fragments).

Finally, all of our testing and model checking relies on an early emphasis on design for testability [99, 56]. 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. The preference for testable, predictable, behavior continued to guide design choices in all of the file system work that followed our initial efforts.

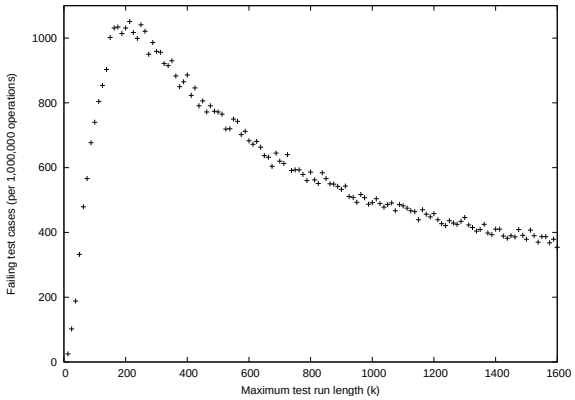
2.2 Lessons Learned in Random Testing

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 to "build one to throw away" [29], 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. We also implemented a simple mechanism for automatically producing stand-alone C test cases, consisting of a `main` function initializing the system and calling all operations involved in the failure. This mechanism simplified communication with the non-JPL developer of the file system being tested in that effort, and improved confidence that erroneous behaviors were not an artifact of the test generation process. The triviality of producing stand-alone executable test cases independent of any framework or abstraction is a very useful feature of more dynamic verification approaches. The mechanism for generating stand-alone test cases may be as simple as printing the literal C code that executes,



A fault appearing only in system initialization can result in a dramatic decrease in test effectiveness as the length of tests increases.



For this version of the file system, very short tests produced very few failures, but increasing test length to more than 200-300 operations also significantly decreased test effectiveness.

Fig. 3 Failure detection, early versions of NVFS: how failures/operation changes with maximum test length, for 1,000,000 test operations.

with constants in place of variables, for all operations and assertions needed by the test. Such code can be easily inserted in a standard boilerplate body that handles system initialization.

We additionally discovered that the length of each random test is a significant factor in the effectiveness of the testing [14], with a change in maximum test length in many cases resulting in a one or two orders-of-magnitude change in the number of failing test cases produced per test operation. The probability of a fault manifesting from any operation is not independent of the number of operations previously performed, nor is it always monotonically increasing; consider the case of a bug that relies on no other code initializing an uninitialized variable. That is, given a fixed test budget of operations, B, performing $\frac{B}{k_1}$ tests of length k_1 can, in real systems, produce 10 or

more times as many failures as performing $\frac{B}{k_2}$ ($\neq k_1$) tests of length k_2 (e.g., as in Figure 3). The optimal length for tests varied as the software matured, suggesting an experimental, iterative approach to test length selection.

Our experiences also confirmed the importance of minimizing [115] randomly produced test cases before debugging [88]. In general, we found that understanding and debugging non-minimized random tests was not an effective use of developer time, while debugging minimized tests in our setting was easy enough that we did not require any kind of bug triage or fuzzer taming [34] system. The only such effort needed was to add the requirement to the delta-debugging loop that the final POSIX operation of any reduced version of the test case be the same as in the original test case (otherwise certain "ubiquitous" bugs embedded in all test cases tended to obscure more interesting faults, in some cases). One test case for each failing operation was provided to the developer.

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 typically more obvious upon 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 contributions to the implementation of some better-known tools for checking C code [87,32]. We selected a very small (50 line) function with no dependence on the larger code base as a case study: if the more ambitious tools proved unable to handle such a small and self-contained fragment of critical code, we would limit our investment of time in a perhaps doomed effort. The function in question, given a string, *canonizes* the string as a pathname by removing extraneous "/" and "." characters and returns 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 [70] and MAGIC [32] either failed to extract a model from the code or failed to find a proof or a counterexample.

A bounded model checker for software checks a program for correctness by producing a SAT (or SMT) formula that is satisfiable only by a bounded-length trace demonstrating that the program does not satisfy a given property. If the formula is unsatisfiable, the program cannot produce an execution of the given length that violates the property. The CBMC bounded model checker [87] was able to verify the important properties of the canonizer up to a maximum path length of 6 characters (the length controls the number of executions of each loop in the code). We attempted to increase the bound, but CBMC timed out (after 24 hours or more) for larger bounds, with a variety of SAT solvers, including ZChaff, limmat, and MiniSAT [93,25,47]. We made limited efforts to apply BLAST and CBMC to other code fragments, but in general found that the tools did not scale to the 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 research tools to low-level device driver software [17] 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.

Perhaps with assistance from the tool authors we might have been able to handle the canonizer function, but the turnaround for such a process is often impractical, especially for flight code where export-control concerns are also a factor. That is, with current abstraction-based tools, it seems to be the case that obtaining good results without the ability to actually share source code with tool authors is at best unlikely; this is not unexpected in an emerging research area, but makes practical exploitation of these methods very difficult. In general, most available abstraction based tools have been developed and applied by research teams, rather than production teams, with a resulting emphasis on producing results for publication on a limited set of known benchmarks, rather than an emphasis on reliability over a wide range of program targets. The Microsoft SLAM project's eventual transformation into a tool (the Static Driver Verifier [1]) for driver developers suggests that technology transfer is possible with sufficient resources, but perhaps only as part of an institutional effort, not as a testing activity of an individual project in an organization. The contrast between the abstraction tools and static source code analyzers is interesting: in general, the static analysis tools work on a much larger range of programs, and require configuration of far fewer highly esoteric parameters in order to produce acceptable results. In part this is due to differences in the underlying technologies (e.g., the abstraction tools are more ambitious in terms of properties checked), but we believe that the eventual emergence of a commercial market, and the origin of source code analysis tools in "developer-driven" utilities rather than academic and industrial research also explains the difference in usability.

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 [75] 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 [73] 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, see Figure 4) includes information on the memory locations of the state of the C program, enabling SPIN to backtrack the running C program. The PROMELA harness resembles a C program for selecting test operations and checking that results match expectations, but provides explicit syntax for nondeterministic selection (e.g., the `if / :: / fi` construct), improving the readability of the test definition. Using the specialized model construction language of SPIN makes the *structure* of the test itself clear, while the infrastructure for random choice selection, systematic exploration, test case storage and replay, etc., in C or Python often

```
:: choice == UNLINK -> /* unlink */
   pick(pathindex, NUM_PATHS);  /* Choose a path */
   c_code { enter_nvfs(); /* Allow memory access to NVFS region */
            now.res = nvfs_unlink (path[now.pathindex]);
            now.nvfs_errno = errno;
            leave(); /* Disallow memory access */ };
   check_reset(); /* Check for system reset and reinitialize/mount NVFS if needed */
   if
   :: (res < 0) && (nvfs_errno == ENOSPC) -> /* If there was an out-of-space error */
      check_space();
   :: ((!did_reset) || (res != -1)) && !((res < 0) && (nvfs_errno == ENOSPC)) ->
      c_code{ enter_ramfs(); /* Allow memory access to RAMFS region */
              now.ramfs_res = ramfs_unlink (path[now.pathindex]);
              now.ramfs_errno = errno;
              leave(); /* Disallow memory access */ }
   :: else -> skip
   fi;
   ...
   assert (res == ramfs_res);
   assert (nvfs_errno == ramfs_errno);
```

Fig. 4 Simplified PROMELA code for file system testing: note mix of C calls with nondeterministic choice operations in PROMELA (::).

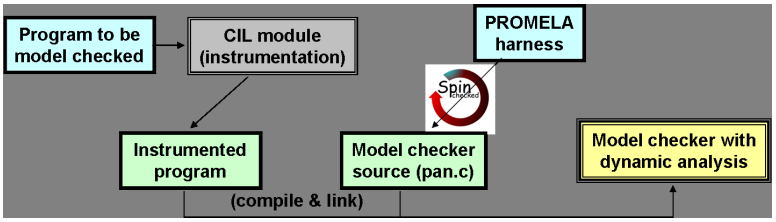


Fig. 5 Using SPIN to model check a program, with CIL annotation to integrate dynamic analyses into model checking via source instrumentation.

obscures the test process. 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 tested for in the code, 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, as 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, a complex space usage problem, 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 [59]. Essentially, this approach relies on using automatic code instrumentation via CIL [95] to automate memory safety checks and overcome the primary limitations of model-driven verification for sequential software (Figure 5). CIL, the C

Intermediate Language, supports instrumentation and static analysis for C programs, after transformation to an unambiguous canonical form. The CIL instrumentation augments the C program we are model checking with automatically inserted function calls that both support features of general utility—such as checking memory safety and correctness of model-checker backtracking, supporting novel heuristic search approaches, etc.—and features more specific to flight software, such as the simulation of software resets during operation.

This tool set improved the ease of model-driven verification (by greatly reducing the time spent debugging backtracking) so much that we found it more convenient to work *only* with model checking, rather than building both a model checker and a random tester [60]. 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. 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 case of comparison with the other MSL file system, an advantage is that both systems are required to present identical interfaces, including error codes, making any divergence an important error. The second case, comparing to an independently developed file system, with a slightly different interface, serves as a guard against 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. Moreover, backtracking the state of most widely used file systems is a non-trivial problem.

Model checking has been quite effective at finding subtle flaws in the file system, as expected. Complete verification, on the other hand, even for very small flash configurations, has proven impossible: week long runs on a 32-GB machine have confirmed that even after unsound abstraction, there are (at minimum) trillions of reachable states in the system. Finding bugs is possible; proving that there are no bugs via complete state space exploration is not.

This has forced us to move away from pure model checking, in the direction of extensive testing. For instance, in order to deal with the large range of possible parameters to function calls, we use feedback [56] just as in our random testing. We also make heavy use of *unsound* abstractions [59]. In model-driven verification we typically apply abstraction by computing an abstract state $a(c)$ from each concrete state c, and considering a state to be visited if we have seen the *abstract* state before. The goal, as with many abstraction approaches, is to choose an abstraction such that if we visit every reachable abstract state (every possible valuation of $a(c)$ where c is a reachable concrete state), and show that the properties of interest hold for these states, we know the property holds for the concrete system. In order to guarantee that we visit every reachable abstract state, the abstraction function and state-space structure together must satisfy a critical requirement:

Soundness: An abstraction function a is **sound** iff:

$$\forall c_1, c_2 : (a(c_1) = a(c_2)) \Rightarrow \forall c_3 : (T(c_1, c_3) \Rightarrow \exists c_4 : T(c_2, c_4) \wedge a(c_3) = a(c_4))$$

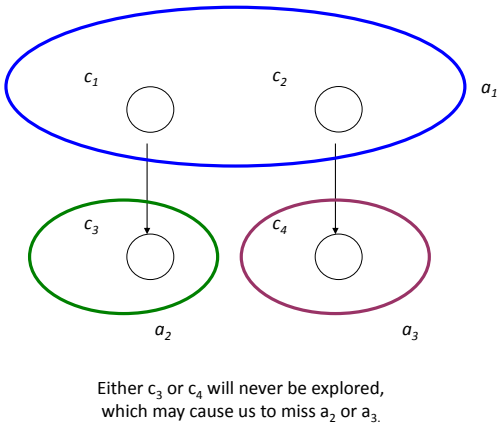


Fig. 6 An example of an unsound abstraction.

where $T(c_1, c_2)$ means that there is a transition from state c_1 to c_2. We require that if two concrete states have the same abstraction, then for each successor of the first state there must exist a successor of the second state with the same abstract value. Why? Recall that explicit-state model checking is typically a depth-first search. Abstraction, then, simply increases the set of states for which a DFS visited-check will succeed, causing the search to backtrack. If two concrete states have the same abstraction but successors with different abstractions, we will visit either c_1 or c_2 first (as the graph transitions may be explored in any order), and thus backtrack when we visit the other (as the abstract states will match), without visiting one of c_3 or c_4, which may be the only way to reach the abstract state $a(c_3)$ or $a(c_4)$. Figure 6 shows an example of this problem. Assuming c_3 and c_4 are the only concrete states for a_2 and a_3, we are guaranteed to miss at least one reachable abstract state here.

Unfortunately, abstractions satisfying this requirement, in the case of the file systems, either fail to reduce the state space sufficiently or do not relate well to any interesting properties (e.g., the abstraction where $a(c)$ is a constant function is "sound" in this sense but useless for verification purposes). In part this is because the file system has surprisingly little symmetry: without going into design details, we simply note that pages on flash are ordered, and that the contents of invalidated pages remain potentially relevant until erased[4]. We therefore rely on unsound abstractions, with the result that even if SPIN's search terminates, we may not have visited all reachable abstract states. This is an acceptable trade-off, especially given that in many cases the search does not terminate after days of computation, even after aggressive abstraction; recall that the system has more than 10^{12} states, even for a very small configuration. The key point is that an unsafe abstraction acts less as traditional model checking abstraction

[4] We do make use of one trivial abstraction that we believe to be sound, the assumption that the precise characters and lengths of path components are irrelevant.

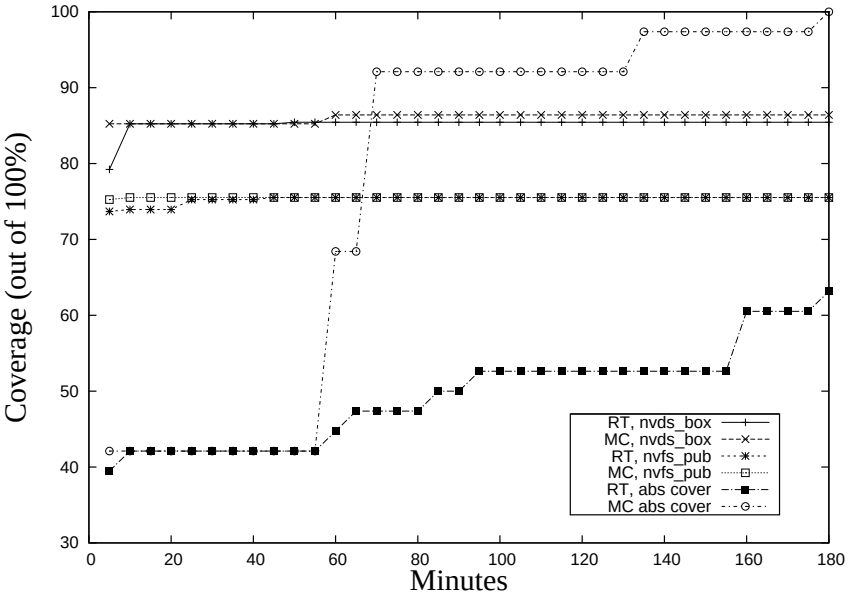


Fig. 7 Coverage for model checking and random testing compared.

guaranteeing a verification result over a smaller state space, and more as a combination of search heuristic and coverage criteria for testing.

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 (depth boundings, etc.) will often reveal the same error in a matter of seconds [77,76,78]. Experiments show that 1 hour (10 minutes on a 6-core machine) of swarm exploration produces better branch and path coverage (by more than a factor of two, in the case of path coverage) than 12 hours of sequential SPIN exploration of the same model [53]. This search strategy further moves us in the direction of aggressive systematic testing, and to a large extent makes the soundness of abstractions an even less relevant factor in our verification.

3.3 Model Checking and Random Testing in One Framework

In recent versions of the test harness, we have integrated model checking and random testing [60]. 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. This lets us compare the testing effectiveness of (1) a systematic exploration with backtracking and (2) a series of random walks. The results of such a comparison are not obvious when the state space is far too large to fully explore, either deterministically model checking with backtracking or in a stochastic fashion. Both approaches to exploring the behavior of a program may leave a large portion of the state space unexplored. The

essential question for each program is whether systematic bias limits the effectiveness of model checking more than redundancy limits random testing.

Our experiments so far indicate that random testing is generally less effective at covering both source code and file system states than explicit-state model checking, if more than about an hour of compute time is available for testing. That is, in cases where test time is very limited, random testing may be more effective than model checking, but we can find no instances in which, given at least an hour to test, random testing outperforms model checking in any aspect of coverage or fault detection. Figure 7 shows a comparison of random testing and model checking, using the same framework and probabilities for choices (recall that the model checking is incomplete) [60]. The first two measures indicate statement 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. Naturally, more time for exploration produces better coverage. In one source-code 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 difference in statement coverage is probably a good predictor of a fundamental difference in fault detection effectiveness, as branch and statement coverage are known to predict mutation detection fairly effectively [49].

The effectiveness of model checking for coverage is best shown, however, by the abstract state coverage. In this case, abstract state coverage is particularly useful, as the low-level block and page management of the system should be equivalent for all concrete states represented by the same abstract state (higher level behavior, such as file and directory structure, will not be equivalent). We can therefore consider good coverage of the abstraction as a sign that this aspect of the file system's behavior (low-level flash management) is well tested. Model checking covers 100 percent of the states that we know to be reachable, while random testing never visits more than 65 percent of those states, even after three hours of performing random walk. We cannot guarantee that model checking achieves truly exhaustive coverage, as we are unable to perform complete model checking for this system, with this abstraction, but the final results match what the design predicts: that is, coverage of the unreached abstract states would violate an intended invariant of the implementation. In general, we suspect that model checking needs a certain "ramp-up" time to obtain net benefit from the overhead of state storage, while random testing faces no such initial cost to overcome. The hour cutoff is, of course, an artifact of our particular benchmarks, processor speed, and other empirical accidents, but indicates an expected general trend of "fast vs. thorough" methods, which will be further discussed shortly, in the context of our last major testing approach, directed testing.

The difference between random testing and model checking becomes even more clear if we change our approach from a single depth-first search to a swarm verification. A swarm run explores 240,166 unique program paths during a 10 minute run (using 6 cores), while the same swarm execution using random testing mode only explores 43,128 unique program paths [53]. With the more efficient swarm verification, the "ramp up" time is collapsed (in part due to search diversity, in part because we make use of multiple cores to perform "an hour of exploration" in 10 minutes).

```
int f (int x, int y) {
  ...
  if (hash(x) == y)
        ERROR;
  ...
  return 0;
}
```

Fig. 8 A "needle in a haystack" motivating example for directed testing.

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, is susceptible to tester bias, 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 [85]. 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. Consider Figure 8, where `x` and `y` are derived from program inputs. On the one hand, randomly producing inputs such that the hash value of `x` is `y` is extremely unlikely, unless the input generation is manually altered to include such a case. On the other hand, using a constraint solver to solve for `x` and `y` is unlikely to work, unless the hash function is unusually weak [5].

Directed random testing combines random testing with symbolic execution to avoid this problem [52]. Expressions that cannot be handled symbolically are reduced to concrete values (taken from a particular execution) before calling a constraint solver. Symbolically solving for `y`, *given the concrete value of* `hash(x)`, however, may be easy. 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 [52,103,31]. 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 is been taken or not is noted. The conjunction of these predicates over the symbolic input (called the *path constraint*) represents a 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 (after replacing any unmanageable expressions with their concrete values), yields a new input that will follow a different path. Repeating this procedure over all possible branch points results in enumeration of all paths. Of course, the exponential cost of such a procedure is problematic, and approaches based on procedure summaries or directed search for new branch coverage first have attempted to mitigate this cost [51,28].

[5] We have actually experimented with using CBMC to solve for hash collisions for checksums, but find that the computational expense is too high for repeated use during testing.

We applied the directed testing tool Splat [110] 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. As with CBMC, we must choose some maximum input size before applying the tool (as otherwise there are an infinite number of potential program paths). For the CBMC input limit of 6 characters, 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 [31], we defined the input as a 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 (the test harness only writes if the volume successfully mounts). 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`, a different kind of needle in a haystack, resulting from simple combinatorics (there are more invalid volumes than valid volumes).

Why the large number of paths? Consider the structure of a flash volume, as the file system reads in pages during the mount operation. Each permutation of pages (depending on page type, version number, and status) may create a different path through the `mount` operation. For as few as 24 pages, given the ordering of version numbers and other distinctions, the number of paths becomes extremely large, especially considering the high overhead per path produced, due to solving complex constraints. Of course, any new path *might* reveal an unknown error, but we were not finding new bugs or improving on the coverage results. We didn't even improve path coverage, since most paths produced were also produced easily by random testing or model checking. 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. Rather than generating volumes, we generated operations (as in random testing and model checking) and applied them to a fixed initialized volume.

This second attempt to apply directed testing was much more successful, and revealed previously undetected arithmetic overflow bugs in the `read` and `write` operations. The much smaller input buffer now 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, maximum buffer sizes,

and other (we thought) safe constraints. 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 (where $x + y$ overflows, yet results in a value that is smaller than the size of the file). 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 successor states for SPIN to explore in model checking.

Unfortunately, after revealing this difficult-to-find error (later 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 undetected 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. In general, we expect that the efficiency, in terms of new states/paths per second for directed testing will be much worse than that for pure random testing or model checking, due to the high overhead of constraint-solver calls. The key to effective use of directed testing would seem to be a focus on the ability to "close the gaps" by finding precisely those "needles-in-the-haystack" that other methods miss. In this instance, we independently applied directed testing, largely duplicating previous efforts, but with a critical payoff in terms of one error.

One promising approach is to explicitly integrate directed testing with a more generally efficient method [89]. In general, we note that the three primary testing methods we used differ, perhaps most importantly, in terms of the balance between short-term efficiency and long-term thoroughness. A testing method must pay a price, in overhead, to store the information or perform the computation to ensure that, over the long run, it will cover a system's behavior well. Random testing produces new paths very efficiently, and exposes high probability faults with ease, but pays a price in redundant and irrelevant activity over longer test periods, and lacks any non-probabilistic guarantees of coverage. Explicit-state model checking pays a penalty for state storage and backtracking, but quickly outpaces random testing in terms of paths or coverage when longer test runs are feasible. Directed testing pays a very high price in overhead, but is the only method that will find certain paths and certain faults. In future work, we plan to make use of the faster methods to obtain solid basic path and branch coverage, then analyze the test cases stored and use directed testing to complete 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, e.g., "states in which the flash volume is almost full" or "states in which there are two bad blocks with valid data"[6]. 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

[6] A block may be bad in the sense that writes to it are not reliable, but correctly written information may still be read.

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 [87] (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 has proven difficult, 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 Testing, Monitoring, and Learning

Testing requires two equally important efforts: (1) generating inputs that produce executions showing interesting program behavior, and (2) determining whether the executions produced conform to expectations. The first problem was the topic of previous sections on random testing, model checking, and constraint-directed testing. We now turn to the second effort, often referred to as the *oracle problem*: the challenge is to produce an oracle that, for a given input, monitors and determines whether a given execution is correct. In previous sections the oracle problem was solved by using *differential testing* where executions of the system under test are compared with executions for the same input of an alternative, more trusted system. Testing the file system posed no significant oracle problem, thanks to the existence of several other trusted file systems, including the various Linux file systems. Often, however, there is no such alternative system to compare against, and the oracle problem becomes a question of defining expected behavior in a formalized machine-readable manner. Traditionally, such oracles are written as test scripts in a high-level scripting language, e.g., Python. A typical test script will emit a sequence of commands or call a program, and subsequently check that the resulting observed behavior is correct.

LaRS joined the task of testing a broader selection of MSL flight software components, in both a workstation simulation environment and on the actual hardware test-beds, as part of the MSL team's Flight Software Internal Test (FIT) team. Prior to our involvement, both the test inputs and evaluation produced by FIT members were written as Python programs. As a result, the specifications were obfuscated and

difficult to apply to other tests, resulting in considerable duplication of effort and weak regression suites.

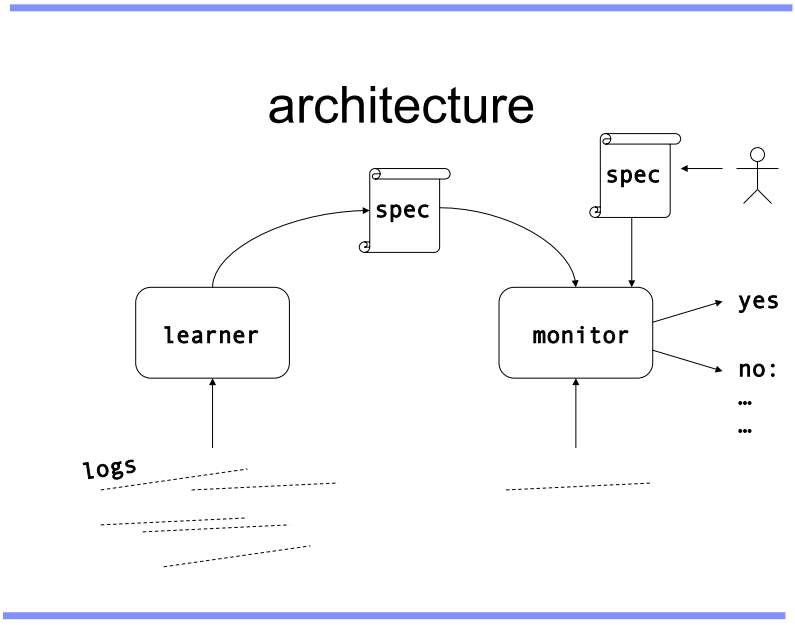


Fig. 9 LogScope consists of two modules: (1) a learner that produces specs, and (2) a monitor that checks specs against log files.

The MSL ground software stores all visible events in an SQL data base, interpretable as a chronologically ordered sequence of events. We have developed a Python framework, called LogScope, for post-processing this information and checking that it conforms to expectations formalized in a specification language [19,18,55]. LogScope consists of two parts: (1) a *log extractor*, which extracts a log file for a particular run from the SQL database, identified by a test key ID, and (2) a *log monitor*, for checking logs against formal specifications (Figure 9). LogScope supports both manually written specifications and specifications learned from recorded, correct, logs.

6.1 Log Extraction

The *log extractor* generates a Python list object, where each element is an *event*: a record implemented as a Python *dictionary* mapping field names to their values. A special field indicates what kind of event it is. There are four kinds: *commands* being issued to the system, *internal state* readings, *internal logging events* called EVRs [58] produced by the system, and *data products* produced by the system which contain the

```
pattern CommandSuccess :
  COMMAND{Type: "FlightSoftwareCommand", Stem: x, Number: y} =>
    {
       EVR{VC1Dispatch: x, Number: y},
       [EVR{Success: x, Number: y},  !EVR{Success: x, Number: y}],
       not EVR{DispatchFailure: x},
       not EVR{Failure : x, Number : y}
    }
```

Fig. 10 A generic specification for flight software commands, in the pattern language.

outputs of the system, for example science data and images. The log extractor sorts events according to spacecraft event times, since the order in which events are received by ground communications software does not correspond to the order in which events are generated on-board, due to varying communication priorities. The log extractor handles all MSL-specific event semantics, producing a generalized log format applicable to other missions and systems.

6.2 Monitoring

The *log monitor* takes two arguments: (1) a log generated by the log extractor, and (2) a specification written in a textual format. The specification language consists of an expressive rule-based language, which includes support for state machines, and a higher-level, more convenient but less expressive, *pattern language*, which is translated into the more expressive specification language before interpretation.

Specifications in the pattern language are easy for test engineers and software developers to read and write. We are working with test engineers to build a library of patterns for common flight software behaviors checked in tests, e.g., that shown in Figure 10.

First, recall that `EVR` indicates a flight software event, a kind of "printf" used to monitor spacecraft behavior. Therefore, this pattern, named `CommandSuccess`, states that in case a command event occurs (meaning a command is issued to the flight software), where the `Type` field has the string value `"FlightSoftwareCommand"`, the `Stem` field (the name of the command) has a value `x` (`x` will be *bound* to that value), and the `Number` field has a value `y` (also a binding variable), then (`=>`), we want to see (in any order, as indicated by set brackets `{...}`): (1) a dispatch of the command `x` with the number `y`; (2) a success, and *after that* no more successes. The square brackets `[...]` indicate an ordering of the event constraints. Furthermore, (3) we do not want to see any dispatch failures for the command; and finally (4) we do not want to see any failures for the command.

The interesting features of this language are its mixture of ordered and unordered sequences of event constraints, including negations, and its support for capturing data values embedded in events and using them later in the test. The pattern language is translated into the core rule-based language, derived from the RULER specification language [20,21]. A useful subset of this language defines state machines with parameterized events and states, where a transition may enter many target states, essentially supporting alternating automata with data (making it possible to specify a family of events with differing details in one pattern). The core language is also inspired by previous state-machine oriented specification languages for monitoring, such as RCAT

[104] and RMOR [67]. The rule-based specification language style is itself a synthesis of previous work on runtime verification as documented in numerous publications [107], and implemented in tools and languages such as MAC [83], MOP [33], and JPAX [68].

6.3 Learning

The specification and monitoring framework just described has been well-received by test engineers, and was integrated into some MSL flight software test suites shortly after release. Feedback from users with no formal methods expertise has guided the language development and resulted in a system that we suspect may be more practical than most monitoring systems developed without the context of an on-going mission critical test effort. One important result has been to alert us to the burden of writing specifications that are more specific than the kind of generic rule shown above. In order to ease this burden we are experimenting with *learning* specifications from runs. The intended procedure consists of running the flight software one or more times, refining a specification after each run to capture an abstraction of the system's behavior. If these runs have been "*good*" runs from the perspective of the test engineer, he/she can "endorse" (perhaps after making manual modifications) the specification, and it can be used to monitor subsequent executions. If these later runs differ, for example due to code modifications, warnings highlighting the discrepancies are issued.

Learning requires that the system can determine when two events are "equal", and users can define which fields should be compared for testing event equality (for example, exact timing is usually abstracted away). We have implemented and applied an *exact learner* which learns the set of all execution sequences seen so far. We are working on a mechanism for learning a mapping from commands to events expected in all execution contexts. We hope to incorporate classic automata-learning results [16] in order to generalize these specifications. This poses a significant challenge, in that we cannot determine that an event sequence is impossible without including the timing of events in the query language, which would make the size of the alphabet prohibitive. Using an offline finite state machine learning algorithm (e.g., Biermann's [27]) would address this query problem. Unfortunately, the SAT queries required to solve the constraint problems produced may be too large to solve.

6.4 Finding an Appropriate Role for Verification Technology

The relevance of the MSL integration testing effort to this paper resides in the selection of where to apply more formal or automated methods. Members of the FIT team have used limited random testing, but LaRS has not pursued more sophisticated approaches (i.e., model checking, random testing with feedback, or directed testing) to test generation for the integrated MSL flight software. The complexity, size, and fragility of full builds makes simply running the software a challenge: maintaining a fully automated testing system and implementing heavyweight oracles is not a reasonable allocation of resources when simply evaluating the results of simple scripted tests is a significant challenge. Here, the utility of formal approaches and constraint-solving lies in *test understanding and disposition*. In our experience, while the oracle problem remains significant for hand-constructed tests, efforts to automate test generation provide little benefit. Even random tests to detect crashes may provide limited information, as

early versions of these systems are often so brittle that triage and fault-isolation based on unexpected operation combinations can overwhelm the more systematic effort to understand faults in simple, expected behaviors. It is critical to identify the dominant problem in a testing effort: if specification is highly problematic, the payoff for automated test generation is fundamentally limited by the lack of a useful test oracle. Moreover, the complex or large systems that most often make specification difficult tend to have extremely complex input structures and poorly understood state spaces, making abstraction and parameter/operation generation difficult to automate. In such cases, more formal methods may still be useful for evaluating tests designed by humans.

7 An Open Problem: Testing the Tester

Complex, frequently modified software systems have bugs. This truth motivates automated testing; unfortunately, it also applies to automated testing systems. Our SPIN framework for testing the various MSL file systems is a complex, frequently modified software system. Over a period of four months, after our test framework had reached a significant level of maturity (and had detected important faults in the file system), we discovered, after long delays and painful debugging, configuration and implementation faults in the tester, introduced during efforts to improve it and adaptations to changes in the file system's interface. We were fortunate that new changes to the file system introduced faults that the LaRS development team became aware of through independent testing. These were very basic faults that should certainly have been detected by our tester. Without this "alarm" we might have continued to proceed with a testing program that inspired false confidence in the file system and our test framework for checking future modifications of the file system.

The problem of detecting faults in a test framework remains an open problem: a test framework is a complex, extremely difficult-to-specify software artifact. The most critical faults in the framework will manifest as missed faults in the system under test, and these will obviously be hard to detect as we typically lack an independent list of faults in the system we are testing!

As a minimal defense against tester faults, we propose that testers examine path coverage statistics [53, 49]. In particular, if path coverage unexpectedly decreases after a modification to the tester, this is a warning sign that a fault may have been introduced. Branch coverage may serve the same purpose, but is a coarse enough measure that even random testing and model checking do not dramatically differ in branch coverage for very short runs. The advantage of path coverage is that a significant decrease can be detected with even very short test runs, rather than only appearing as an anomaly after an overnight test run. The overhead for collecting path coverage, even if implemented in the most simplistic and inefficient fashion, is only around 12-15% for model checking, inexpensive enough to be a default option for shorter test runs [59]. Table 2 shows differences in path coverage of key functions for a variety of versions of our MSL file system test harness, after only 10 minutes of testing with swarm. We mark the "known bad" versions with an **X** in the table: these are actual faulty versions of the tester, applied at least once during our testing efforts. We highlight the best path coverage results for each function. Observe that no *faulty* version improves on more than one function, over the standard tester configuration. If we consider total path coverage rather than per-function coverage, every faulty version decreases path coverage by at

Table 2 Detailed coverage results from file system testing: total unique function paths explored by different test harness versions.

	Function							
Version	**mount**	**write**	**read**	**lseek**	**mkdir**	**rmdir**	**creat**	**unlink**
+No hw fault	**31,958**	18,222	89	131	**19,072**	**19,353**	**16,411**	2,072
+No lseek	25,433	19,934	93	0	16,980	12,482	12,319	3,020
+R/W step >	21,394	16,837	92	141	13,418	11,865	10,633	3,252
Standard	21,246	16,203	97	140	13,252	11,615	10,960	2,935
+No close	21,097	18,545	87	135	13,111	11,087	10,721	2,832
+No read	21,202	16,324	0	140	11,955	10,038	9,971	2,943
X+ No resets	19,200	19,687	74	89	12,132	9,542	10,254	1,655
X Abs. bug	18,685	16,120	86	142	11,905	10,016	9,427	2,358
X+ FD bug	18,880	12,148	64	151	11,942	9,110	9,009	2,617
+ No unlink	18,077	15,288	99	153	10,064	8,618	8,144	0
+ 1 hw fault	18,689	15,712	82	120	9,858	7,295	8,440	2,347
- With rename	17,169	15,564	78	142	9,506	7,161	7,517	1,726
- R/W step <	15,277	13,771	93	134	9,432	6,729	7,345	1,775
X Choice bug	7,738	7,693	91	126	4,577	4,275	4,730	2,743
+ No dirs	18,116	**59,850**	**160**	**206**	0	0	3,803	**4,707**
X Path bug	12,903	12,641	23	104	1,243	1	1,266	1
X Rand. test	11,026	1,741	61	66	2,197	1,599	2,542	1,246
X 2 erases	10	2	3	2	2	1	2	1
No swarm	7,324	823	9	10	9,062	8,749	6,131	91

least 5,000 paths. Increasing focus, e.g., by simply not testing directory operations, however, may considerably improve path coverage for certain functions.

7.1 Test Focus

Examining path coverage as test framework configuration parameters change also suggests that altering the *focus* of tests, as swarm alters the model checker's search mechanism, may improve test efficiency. In the table, versions with a + indicate increased test focus (more thorough testing of a smaller range of behaviors) while those with a - involve testing *more* behavior than the standard configuration for testing. As a simple example of change in focus, consider adding or removing a test operation: if we have a fixed test budget, and only call 5 operations, each of the 5 operations will be called more often than if we test 6 operations for the same test budget. However, the 5 operation test may miss interactions with the 6th operation, which may result in missed faults. As the table shows, however, the payoff for missing those interactions may be a significant increase in path coverage over the tested operations. There is a delicate tradeoff between potentially missing some errors due to a narrow focus, and only shallowly covering complex behaviors due to a very broad focus. Given that our framework relies on 130 configuration parameters, automation is essential if we hope to exploit this alternative source of test diversity.

Exploiting focus diversity in model checking is complicated by the lengthy runtime of each exploration, even in swarm verification. In pure random testing, however, a different "swarm" strategy that simply randomly selects a new configuration before each test case has proven extremely effective for testing compilers as well as an open source flash file system [62,34]. Because the test effectiveness gains for file systems are generally not as large as those for compilers, in the MSL testing we have thus far chosen the advantages of swarmed model checking over the alternative swarmed random testing approach, but in settings where less model checking expertise is available, the

random testing approach with diversified test focus may be preferable. Additionally, the gains from configuration swarming, if it could be applied to model checking, are probably more modest than in random testing: a major source of swarm's power is the problem that some API calls or test features *suppress* other behavior (i.e., `close` calls make it harder to explore `write` behaviors) [61]. In model checking, however, paths without `close` calls will be explored frequently, due to state tracking and DFS behavior, while in random exploration such paths are almost never taken.

8 A Goal-Oriented Approach to Methodology

We do not claim to extract a detailed verification methodology from our experiences with JPL flight software. However, our experience (both positive and negative) suggests that constructing a methodology *around certain goals* may be beneficial, in situations similar to ours. By "situations similar to ours" we encompass cases where (1) the chief purpose is to establish reliability of a critical software system, and the resources, expertise, and commitment exist to benefit from aggressive automated testing and verification technologies, (2) the system is self-contained and well-defined enough that the specification problem does not dominate the verification problem, (3) resources, particularly time, are constrained, (4) the software in question is written in a language with significant verification tool support, such as C or Java, but not in a language, e.g. SPARK [3], aimed at integrated development and verification and (5) the system is sufficiently large or complex that complete proof of correctness is not possible. For software systems larger than 1,000 lines of code, (5) seems likely to be the case in most realistic projects. For very large systems, such as our integrated MSL flight software build, (2) seems unlikely to be the case. Our core proposals are:

1. **Run tests.** As soon as a software system, or a executable design prototype, exists, it is time to begin running tests. Even when a completely test-driven development [22] is not suitable, reliable systems should be designed with testing in mind [99, 56]. Early testing can reveal critical design flaws before they become embedded in too many aspects of an implementation to cleanly modify, and *early testing gives the test effort time to mature with the implementation.* Even a very limited automated testing system will help to understand the specification and behavior of the system, and will, if created early in the life-cycle of the program, grow with the implementation. Early testing also forces test engineers and developers to face any major limitations in testing: highly complex input structures, very large input ranges, nondeterministic behavior, and the like. If these problems are only discovered after the implementation is mature, solutions will be much more limited than if the software can potentially be adapted to be more testable. This principle also applies to the last stages of development. When a critical module has been integrated into a working system and passed a large number of sophisticated automated tests, and has accumulated many pages of good coverage statistics, it is tempting to stop testing. A good automated testing system, by removing human effort from the test loop, should allow us to resist this temptation: if the system is working, the test effort will continue as a background task, without consuming resources other than computing power. If continued testing begins to consume

human resources again, it is presumably because faults are being detected, and this is better than the alternative of not testing[7].

2. **Do not make the perfect the enemy of the useful.** If a method *could* produce powerful verification results, but in practice is not working out, i.e., is not exposing faults, or even *running tests* that *could* expose faults, it is reasonable to abandon it. Approaches providing no test benfit are perhaps best put aside until a less powerful method can be matured to the point of executing tests while human resources concentrate on the more difficult approach. Simple random testing can serve as a mitigation for the risk of test efforts that fail because they rely on immature technologies or unknown properties of the code being tested (small model properties, exploitable abstractions, sensitivity to search strategies, etc.). There are no clear reliability gains from determining that a software system is not a suitable target for a verification technology, but the risk-reward balance becomes less unpleasant if some form of automated testing is already in place and executing tests without using many human resources.
3. **Know the implications of faults, and use them.** While we believe that path and branch coverage may help detect gaps in a testing process, we *know* that known, undetected faults can expose problems with a test framework. Assuming that a test effort will involve multiple approaches, including mental execution, by independent testers and developers, it is critical to make sure that an understanding of every detected fault is communicated to the entire team. It is obviously important for developers to understand faults, in order to produce bug fixes or workarounds, but it is also critical that test engineers know as much as possible about faults. Most importantly, knowledge of faults that were not detected by a test system may imply a weakness of the test system. Understanding detected faults may also expose test framework problems: if the existence of a fault implies the existence of a simple test case that detects the fault, then if that test case has never been observed, the test framework may be missing other faults, or at a minimum producing overly complex (and thus hard-to-debug) test cases. Without endorsing full mutation testing [43, 63, 97], which is effective but may be too expensive or difficult for resource-constrained test efforts, we also suggest mutation testing a test framework by introducing simple, well-understood faults that both developers and test engineers believe should be detected.
4. **Exploit computing power.** Automated testing is effective in large part because the computing power available to execute tests is such that even if each test is highly unlikely to expose a fault or contribute to coverage, the overall result may be effective. *Random testing can exploit as much compute power as is available: there is no reason to have idle hardware, when more random tests can be generated.* With model checking, exploiting massive parallelism is slightly more involved than choosing a new set of random seed ranges, but swarm technology extends to many machines as well as many cores on a single machine, as there is no communication overhead between searches. Even if much of the additional exploration is redundant, using commodity hardware rather than specialized clusters makes this a very cost effective way to improve testing.

[7] It may also be because the automated tester is poorly designed and fragile, which is often a consequence of not beginning testing early enough, once again emphasizing the importance of running tests early, often, and until a project is complete and the code will never execute again.

5. **Use static analysis.** It is difficult to imagine any good reason not to make static bug detection tools a routine part of the build process for any critical software project. In fact, rewriting code to remove even false positive warnings produced by static analysis can be a helpful discipline, re-enforcing the need for clean compilation and analysis before code is considered ready to execute. Static analysis tools are generally mature and well supported, requiring few project resources beyond the purchase price of the commercial tools (the resources required for *fixing bugs* detected by the tools are not a liability!).

9 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 heavyweight 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, 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, even for a self-contained and well-specified module such as a file system. For now, we can only suggest that you test early, and test often.

Acknowledgments: We are indebted to the members of the Mars Science Laboratory Flight Software Internal Test (FIT) team for contributions to the pattern language design and the early efforts at learning: in particular, Cin-Young Lee, Chris Delp, Margaret Smith, Hyejung Yun, Lisa Tatge, and Hui Ying Wen all suggested features for the pattern language or applied the tool in its early stages. The authors would also like to thank Martin Leucker for input regarding the learning of automata. Finally, we would like to thank the anonymous reviewers of earlier drafts of this paper.

References

1. About Static Driver Verifier. `http://www.microsoft.com/whdc/DevTools/tools/SDV.mspx`.
2. ACL2 Version 6.3. `http://www.cs.utexas.edu/~moore/acl2`.
3. AdaCore: SPARK Pro. `http://www.adacore.com/sparkpro/`.
4. CodeSonar: GrammaTech static analysis. `http://www.grammatech.com/products/codesonar`.
5. Hyper-V - Microsoft. `http://www.microsoft.com/servers/hyper-v-server/default.mspx`.
6. JPL Laboratory for Reliable Software (LaRS). `http://lars-lab.jpl.nasa.gov`.
7. Mars Science Laboratory. `http://mars.jpl.nasa.gov/msl`.
8. Open Group Base Specifications Issue 6. `http://www.opengroup.org/onlinepubs/009695399/`.
9. Software development testing and static analysis tools: Coverity. `http://www.coverity.com`.
10. Source code analysis tools for software security & reliability: Klockwork. `http://klocwork.com`.

11. VCC: A C verifier - Microsoft Research. `http://research.microsoft.com/en-us/projects/vcc/`.
12. Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.
13. James Andrews, Felix Li, and Tim Menzies. Nighthawk: A two-level genetic-random unit test data generator. In *Automated Software Engineering*, pages 144–153, 2007.
14. James H. Andrews, Alex Groce, Melissa Weston, and Ru-Gang Xu. Random test run length and effectiveness. In *Automated Software Engineering*, pages 19–28, 2008.
15. James H. Andrews, Susmita Haldar, Yong Lei, and Chun Hang Felix Li. Tool support for randomized unit testing. In *Proceedings of the First International Workshop on Randomized Testing*, pages 36–45, Portland, Maine, July 2006.
16. Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87106, 1987.
17. Thomas Ball and Sriram Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN Workshop on Model Checking of Software*, pages 103–122, 2001.
18. Howard Barringer, Alex Groce, Klaus Havelund, and Margaret Smith. Formal analysis of log files. *Journal of Aerospace Computing, Information, and Communication*, 7(11):365–390, 2010.
19. Howard Barringer, Klaus Havelund, David Rydeheard, and Alex Groce. Rule systems for runtime verification: A short tutorial. In *Proc. of the 9th Int. Workshop on Runtime Verification (RV'09)*, volume 5779 of *LNCS*, pages 1–24. Springer, 2009.
20. Howard Barringer, David Rydeheard, and Klaus Havelund. Rule systems for run-time monitoring: from Eagle to RuleR. In *Proc. of the 7th International Workshop on Runtime Verification (RV'07)*, volume 4839 of *LNCS*, pages 111–125, Vancouver, Canada, 2007. Springer.
21. Howard Barringer, David E. Rydeheard, and Klaus Havelund. Rule systems for run-time monitoring: from Eagle to RuleR. *J. Log. Comput.*, 20(3):675–706, 2010.
22. Kent Beck. *Test Driven Development: By Example*. Addison Wesley, 2002.
23. Boris Beizer. *Software Testing Techniques*. International Thomson Computer Press, 1990.
24. D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. Generating tests from counterexamples. In *International Conference on Software Engineering*, pages 326–335, 2004.
25. Armin Biere. The evolution from Limmat to Nanosat. Technical Report 444, Dept. of Computer Science, ETH Zűrich, 2004.
26. Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207, 1999.
27. A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behaviour. *IEEE Transactions on Computers*, 21:592–597, 1972.
28. Peter Boonstoppel, Cristian Cadar, and Dawson Engler. Rwset: Attacking path explosion in constraint-based test generation. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 351–366, 2008.
29. Frederick Brooks. *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition*. Addison-Wesley Professional, 1995.
30. Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Operating System Design and Implementation*, pages 209–224, 2008.
31. 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.
32. 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.
33. Feng Chen and Grigore Roşu. MOP: An efficient and generic runtime verification framework. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'07)*, 2007.
34. Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. Taming compiler fuzzers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 197–208, 2013.

35. Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Experimental assessment of random testing for object-oriented software. In David S. Rosenblum and Sebastian G. Elbaum, editors, *International Symposium on Software Testing and Analysis*, pages 84–94. ACM, 2007.
36. Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of haskell programs. In *ICFP*, pages 268–279, 2000.
37. Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 2000.
38. Edmund M. Clarke and E. Emerson. The design and synthesis of synchronization skeletons using temporal logic. In *Workshop on Logics of Programs*, pages 52–71, 1981.
39. 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.
40. James Corbett, Matthew Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.
41. Christoph Csallner and Yannis Smaragdakis. JCrasher: an automatic robustness tester for Java. *Softw., Pract. Exper.*, 34(11):1025–1050, 2004.
42. Leonardo de Moura and Nikolaj Bjorner. Z3: An efficient SMT solver. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.
43. R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 4(11), 1978.
44. Roong-Ko Doong and Phyllis G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 3(2):101–130, April 1994.
45. J. W. Duran and S. C. Ntafos. Evaluation of random testing. *IEEE Transactions on Software Engineering*, 10(4):438–444, 1984.
46. Matthew B. Dwyer, Sebastian G. Elbaum, Suzette Person, and Ragul Purandare. Parallel randomized state-space search. In *International Conference on Software Engineering*, pages 3–12, 2007.
47. Niklas Een and Niklas Sorensson. An extensible SAT-solver. In *Symposium on the Theory and Applications of Satisfiability Testing (SAT)*, pages 502–518, 2003.
48. John Erickson and Rajeev Joshi. Proving correctness of a Flash filesystem in ACL2. Unpublished manuscript in preparation, 2006.
49. Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Amin Alipour, and Darko Marinov. Comparing non-adequate test suites using coverage criteria. In *International Symposium on Software Testing and Analysis*, pages 302–313, 2013.
50. Patrice Godefroid. Verisoft: a tool for the automatic analysis of concurrent software. In *Computer-Aided Verification*, pages 172–186, 1997.
51. Patrice Godefroid. Compositional dynamic test generation. In *Principles of Programming Languages*, pages 47–54, 2007.
52. Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Programming Language Design and Implementation*, pages 213–223, 2005.
53. Alex Groce. (Quickly) testing the tester via path coverage. In *Workshop on Dynamic Analysis*, 2009.
54. Alex Groce, Alan Fern, Jervis Pinto, Tim Bauer, Amin Alipour, Martin Erwig, and Camden Lopez. Lightweight automated testing with adaptation-based programming. In *IEEE International Symposium on Software Reliability Engineering*, pages 161–170, 2012.
55. Alex Groce, Klaus Havelund, and Margaret H. Smith. From scripts to specifications: the evolution of a flight software testing effort. In *32nd Int. Conference on Software Engineering (ICSE'10), Cape Town, South Africa*, ACM SIG, pages 129–138, 2010.
56. 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.
57. Alex Groce, Gerard Holzmann, Rajeev Joshi, and Ru-Gang Xu. Putting flight software through the paces with testing, model checking, and constraint-solving. In *International Workshop on Constraints in Formal Verification*, pages 1–15, 2008.
58. Alex Groce and Rajeev Joshi. Exploiting traces in program analysis. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 379–393, 2006.

59. 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.
60. Alex Groce and Rajeev Joshi. Random testing and model checking: Building a common framework for nondeterministic exploration. In *Workshop on Dynamic Analysis*, pages 22–28, 2008.
61. Alex Groce, Chaoqiang Zhang, Amin Alipour, Eric Eide, Yang Chen, and John Regehr. Help, help, I'm being suppressed! the significance of suppressors in software testing. In *IEEE International Symposium on Software Reliability Engineering*, pages 390–399, 2013.
62. Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. Swarm testing. In *International Symposium on Software Testing and Analysis*, pages 78–88, 2012.
63. R. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3(4), 1977.
64. R. Hamlet and R. Taylor. Partition testing does not inspire confidence. *IEEE Transactions on Software Engineering*, 16(12):1402–1411, 1990.
65. Richard Hamlet. Random testing. In *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.
66. Richard Hamlet. When only random testing will do. In *International Workshop on Random Testing*, pages 1–9, 2006.
67. Klaus Havelund. Runtime verification of C programs. In *Proc. of the 1st TestCom/FATES conference*, volume 5047 of *LNCS*, Tokyo, Japan, June 2008. Springer.
68. Klaus Havelund and Grigore Roşu. Efficient monitoring of safety properties. *Software Tools for Technology Transfer*, 6(2):158–173, 2004.
69. M. P. E. Heimdahl, S Rayadurgam, W. Visser, D. George, and J. Gao. Auto-generating test sequences using model checkers: A case study. In *International Workshop on Formal Approaches to Testing of Software (FATES)*, 2003.
70. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Principles of Programming Languages*, pages 58–70, 2002.
71. Tony Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, 50(1):63–69, 2003.
72. Gerard Holzmann. Static source code checking for user-defined properties. In *Conference on Integrated Design and Process Technology*, 2002.
73. Gerard Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.
74. Gerard Holzmann. The power of ten: Rules for developing safety critical code. *IEEE Computer*, 39(6):95–97, June 2006.
75. Gerard Holzmann and Rajeev Joshi. Model-driven software verification. In *SPIN Workshop on Model Checking of Software*, pages 76–91, 2004.
76. Gerard Holzmann, Rajeev Joshi, and Alex Groce. Swarm verification. In *Automated Software Engineering*, pages 1–6, 2008.
77. Gerard Holzmann, Rajeev Joshi, and Alex Groce. Tackling large verification problems with the swarm tool. In *SPIN Workshop on Model Checking of Software*, pages 134–143, 2008.
78. Gerard J. Holzmann, Rajeev Joshi, and Alex Groce. Swarm verification techniques. *IEEE Transactions on Software Engineering*, 37(6):845–857, Nov./Dec. 2011.
79. Rajeev Joshi and Gerard Holzmann. A mini-challenge: Build a verifiable filesystem. In *The Conference on Verified Software: Theories, Tools, Experiments*, 2005.
80. Matt Kaufmann, Panagiotis Manolios, and J. Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.
81. Moonzoo Kim, Yunja Choi, Yunho Kim, and Hotae Kim. Formal verification of a flash memory device driver — an experience report. In *SPIN Workshop on Model Checking of Software*, pages 144–159, 2008.
82. Moonzoo Kim, Yunja Choi, Yunho Kim, and Hotae Kim. Pre-testing flash device driver through model checking techniques. In *International Conference on Software Testing, Verification, and Validation*, pages 475–484, 2008.
83. Moonzoo Kim, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Java-MaC: a runtime assurance tool for Java. In *Proc. of the 1st International Workshop on Runtime Verification (RV'01)*, volume 55(2) of *ENTCS*. Elsevier, 2001.
84. Moonzoo Kim, Yunho Kim, and Hotae Kim. Unit testing of flash memory device driver through a SAT-based model checker. In *Automated Software Engineering*, pages 198–207, 2008.

85. James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
86. Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an OS kernel. In *ACM Symposium on Operating Systems Principles*, 2009.
87. 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.
88. Yong Lei and James H. Andrews. Minimization of randomized unit test cases. In *International Symposium on Software Reliability Engineering*, pages 267–276, 2005.
89. Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *International Conference on Software Engineering*, pages 416–426, 2007.
90. William McKeeman. Differential testing for software. *Digital Technical Journal of Digital Equipment Corporation*, 10(1):100–107, 1998.
91. Eric Mercer and Michael Jones. Model checking machine code with the GNU debugger. In *SPIN Workshop on Model Checking Software*, 2005.
92. 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.
93. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Linao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference*, pages 530–535, 2001.
94. Madanlal Musuvathi, David Park, Andy Chou, Dawson Engler, and David Dill. CMC: A pragmatic approach to model checking real code. In *Symposium on Operating System Design and Implementation*, 2002.
95. 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.
96. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Springer, 2002. Volume 2283 of LNCS.
97. J. Offutt and A. Abdurazik. In *Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, 2000.
98. 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.
99. Brett Pettichord. Design for testability. In *Pacific Northwest Software Quality Conference*, October 2002.
100. Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107, April 2005.
101. Glenn Reeves and Tracy Neilson. The Mars Rover Spirit FLASH anomaly. In *IEEE Aerospace Conference*, 2005.
102. Robby, Matthew Dwyer, and John Hatcliff. Bogor: An extensible and highly-modular model checking framework. In *European Software Engineering Conference / Symposium on the Foundations of Software Engineering*, pages 267–276, 2003.
103. Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *Foundations of Software Engineering*, pages 262–272, 2005.
104. Margaret Smith and Klaus Havelund. Requirements capture with RCAT. In *16th IEEE International Requirements Engineering Conference (RE'08)*, IEEE Computer Society, Barcelona, Spain, September 2008.
105. Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In *Principles of Programming Languages*, pages 97–108, 2007.
106. Various. A collection of NAND Flash application notes, whitepapers and articles. Available at `http://www.data-io.com/NAND /NANDApplicationNotes.asp`.
107. Runtime Verification. `http://www.runtime-verification.org`.
108. Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, April 2003.
109. Willem Visser, Corina Păsăreanu, and Radek Pelanek. Test input generation for Java containers using state matching. In *International Symposium on Software Testing and Analysis*, pages 37–48, 2006.

110. Ru-Gang Xu, Rupak Majumdar, and Patrice Godefroid. Testing for buffer overflows with length abstraction. In *International Symposium on Software Testing and Analysis*, pages 19–28, 2008.
111. Junfeng Yang, Can Sar, and Dawson Engler. EXPLODE: a lightweight, general system for finding serious storage system errors. In *Operating System Design and Implementation*, pages 131–146, 2006.
112. Junfeng Yang, Can Sar, Paul Twohey, Cristian Cadar, and Dawson Engler. Automatically generating malicious disks using symbolic execution. In *IEEE Symposium on Security and Privacy*, pages 243–257, 2006.
113. Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. In *Operating System Design and Implementation*, pages 273–288, 2004.
114. Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 283–294, 2011.
115. Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.