(Quickly) Testing the Tester via Path Coverage

Alex Groce
Laboratory for Reliable Software
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109, USA
Oregon State University
School of Electrical Engineering and
Computer Science
Corvallis, OR 97331 USA
agroce@gmail.com

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

ABSTRACT

The configuration complexity and code size of an automated testing framework may grow to a point that the tester itself becomes a significant software artifact, prone to poor configuration and implementation errors. Unfortunately, testing the tester by using old versions of the software under test (SUT) may be impractical or impossible: test framework changes may have been motivated by interface changes in the tested system, or fault detection may become too expensive in terms of computing time to justify running until errors are detected on older versions of the software. We propose the use of path coverage measures as a "quick and dirty" method for detecting many faults in complex test frameworks. We also note the possibility of using techniques developed to diversify state-space searches in model checking to diversify test focus, and an associated classification of tester changes into focus-changing and non-focus-changing modifications.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms

Measurement, Verification

Keywords

test frameworks, evaluation of test systems, regression testing

ACM acknowledges that this contribution was authored or co-authored by a contractor or affiliate of the U.S. Government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
WODA '09 July 20, 2009, Chicago, Illinois, USA.
Copyright 2009 ACM 978-1-60558-656-4/09/07 ...$5.00.

1. INTRODUCTION

The benefits of effective large-scale automatic test generation are increasingly accepted in the software engineering community. Random testing [14, 10, 3, 21], constraint-based testing [9, 22, 24, 4], and model checking-based methods [23, 15, 11] all provide the ability to automatically generate a very large number of tests for a software system. Unfortunately, most of these systems rely on the implementation of a test harness and a set of configuration options that control the harness and any associated automated testing tool (e.g., DART [9], Splat [24], SPIN [18]). We refer to the test harness code plus any configuration options generally as the "tester" — the test framework plus option choices that govern its behavior.

The test harness will, in many cases, be occasionally modified: it must adjust to changes to the interface of the SUT (software under test). Even if the interface is constant, configuration changes (which may or may not require changes to the test harness itself) are likely to be frequent. In random testing, operational profiles may change, or the pursuit of errors may lead to adjustments in the probabilities of operations and parameters and the introduction or modification of feedback mechanisms. The definition of inputs may need to be altered to improve the scalability of constraint-based approaches. Model checking-based approaches will often present an even wider array of tunable options, given that they will almost certainly combine user-defined state-space abstractions with a choice of search strategy.

Complex, frequently modified software systems have bugs. This is a truth known to all who are likely to use automated testers. In the case of a tester, the most obvious consequences of faults are (1) false positives and (2) inability to detect otherwise detectable faults in the SUT. False positives are relatively harmless: they are a visible symptom, and in many cases it requires little effort to determine that they do not relate to actual faults in the SUT. Failure to detect faults, however, may have grave consequences: an ineffective automated tester may produce a dangerous false confidence in the quality of the SUT, and result in a high rate of faults detected later in the development cycle (with the related increase in the cost of correction) or in operation (in which case the consequences may be catastrophic).

These issues are not merely theoretical concerns, in our case. As part of our ongoing efforts to apply aggressive automated testing to the file systems for the next JPL Mars

rover mission, the Mars Science Laboratory [1], we have developed a large test framework, providing model checking with unsound abstractions and random testing [13]. In the last four months, we have 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 SUT's interface. We believe that we have restored our tester to its former effectiveness (and in some ways improved it), but are greatly disturbed by our complacence during weeks of tests reporting no errors. In our case, we were fortunate that new changes to the SUT introduced faults that the developer became aware of through independent testing, 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 correctness of the SUT, in the test team, the developer, and users of the file system modules.

How do we detect the non-detection of errors? What kind of test oracle can be used to expose this kind of problem?

1.1 Traditional Regression Testing

We can (perhaps) run the tester on older versions of the SUT, which might expose a fault that prevents detection of already known bugs. It may (and often will) be the case, however, that older faults in the SUT will have a higher probability of detection (known to likely be the case in our applications [2]), and so a faulty tester may still uncover known faults. Even if fault detection probabilities remain roughly constant, they may be low enough to make regression testing very expensive. If finding an error requires many hours of testing, checking the ability to detect that fault may be too time-consuming to be practical.

Additionally, as noted above, in many cases the tester changes in response to interface changes to the SUT, making regression maintenance a costly and error-prone process.

A more general concern is that, in searches of large state spaces, the ability to (quickly) find a given fault or set of faults is not a very good predictor of the ability to find other, unknown faults. Dwyer, Person, and Elbaum show the limitations of finding a given needle in a haystack as a predictor for the value of model checking heuristics, a case that strongly resembles our "testing a tester" problem [7].

1.2 Differential Testing

We clearly cannot rely on differential testing [19] to compare *testers*: the only comparison would be with an equally powerful (or better) tester. If we had access to a better tester, we would presumably use it in the first place. An equally powerful tester will, clearly, require execution time roughly equal to that of the tested tester. Given that large scale automated testing runs often, in our experience, require hours or days to detect faults that are not dense in the state-space (the faults we are most concerned with), this is at least as impractical as regression testing. We do expect independent test efforts to exist in many cases, and for this to serve (as in our experience) as a kind of "differential testing," but to operate without any particular automation or systematization.

1.3 Coverage Metrics

We investigated the feasibility of introducing a regression test system for our tester, but (thus far) abandoned this idea in light of the analysis above. We then noted that our tester already provided the option to capture coverage information, the most common measure of test effectiveness. Why not rely on coverage metrics as our proverbial canary in the coal mine?

We therefore hoped that branch or statement coverage could be adopted as a first-order defense against tester regression. Such coverage is inexpensive to compute and, when there are unexpected differences in two testers, the results are easy to interpret — i.e., "we are not calling this top-level function anymore" or "the only pathname we're using is root" (examples from our own experience). Unfortunately, as we previously discovered, branch coverage and abstract state-space coverage for radically different search approaches did not differ greatly, even after an hour of testing [13]. It seemed unlikely that most tester faults would induce much larger differences in coverage than those between random testing and model checking. In practice, the coverage metrics we had been concentrating on were simply too coarse-grained to provide quick detection of many tester faults. We therefore turned to a finer grained coverage metric, as an alternative in the many cases where branch or statement coverage does not reveal differences in testers.

1.4 Path Coverage

This paper proposes the use of *path coverage* as a measure for tester effectiveness. We suggest that path coverage is sufficiently fine-grained that reductions in tester effectiveness will likely introduce significant decreases in path coverage of the SUT, even over extremely short testing periods (*10 minutes*, though equivalent to perhaps an hour due to exploitation of parallelism). Path coverage is not a perfect approach to this problem: it generally acts as a test for regressions, checking for a decrease in quality of the tester, rather than a method for exposing faults that have been present in all versions of a tester. We discuss below the possibility of using path coverage results to suggest "improvements" to the tester.

We report our observations of the effectiveness of this method for a complex JPL test framework, applied to a flight mission flash file system. Our results confirm that measuring path coverage would have indicated faults in the test harness or its configuration that otherwise went undetected. Path coverage results also suggest that configurations that we have not used in our larger test runs may sometimes prove effective in increasing path coverage. Path coverage generally makes trade-offs in tester behavior more visible, and may indicate a need for configuration diversity as well as search diversity. Without a *quick and fine-grained* measure of configuration effectiveness, however imprecise it may be, such diversification would probably be too *ad hoc* to be effective.

The primary threats to the validity of these preliminary results are the application to one tester and one SUT, but we expect to follow these preliminary results with an investigation of other industrial-scale automated testers and other SUTs. In a sense, this paper simply repeats the common theme of testing literature, that coverage metrics can be useful in evaluating test suites. However, we believe that by re-focusing this notion to the problem of testing a particular version of a test framework and its configuration, and indicating the utility of full path coverage (generally considered a somewhat expensive metric to record) for this purpose,

we may increase the effectiveness of large-scale automated testing.

1.5 Summary: Proposed Test Methodology

1. At a minimum, compute statement and/or branch coverage results for a short run of a new tester version. Compare this to previous coarse coverage results. If there are unexpected changes, this serves as a quick and easy-to-interpret sign of fault in the new tester.

2. If coarser coverage measures do not reveal a problem, compare total path coverage results. If there is a large decrease, this is a strong indication of a fault in the tester. If the change to the tester was intended to increase *test focus* (that is, to omit some operations to concentrate on certain aspects of behavior) proceed to the next step before assuming a fault is present.

3. Finally, compare path coverage for individual functions (in our case, top-level operations in the test harness). If the change to the tester was intended to increase focus, and the path coverage results show a *tradeoff* in coverage (where less coverage of one aspect of behavior is "paid for" by increased coverage in the areas "in-focus"), then proceed with testing. Otherwise, a fault may well be indicated.

2. MEASURING PATH COVERAGE

We use CIL [20] to instrument the SUT in order to record path coverage information [12]. In some applications of our approach, path coverage would be a simple end-to-end measure, for SUTs in which test cases consist of an input value which produces an execution and output. However, for the stateful systems that are perhaps most suitable for random testing and model-driven verification [21, 15], a test sequence typically consists of a series of function calls. We choose to measure path coverage at the granularity of top-level function calls. That is, we maintain a set of paths covered for each *top-level-entry function* that is called by the test harness. If our test harness only called the `open, close, read` and `write` functions of a file system, we would maintain separate sets of paths for each of those four functions. We record a path as a bit vector, containing every `if-then` decision made by the execution of the function, from entry until return to the test harness. This bit vector includes all decisions made in functions called by the top-level function, recursively, and therefore records all path information. We exploit CIL's reduction of branching/looping constructs to reduce the path-recording problem to only the `if-then` case[1].

In general the number we will use to measure tester effectiveness is simply the total number of unique paths through functions executed during a test run. E.g., for the four function example above, the path coverage would be the sum of the number of unique paths through `open`, unique paths through `close`, unique paths through `read`, and unique paths through `write`.

2.1 Overhead: Costs vs. Benefits

[1]In practice, additional information is required for switch statements, but our SUT does not use this feature of C.

In previous work, we measured the overhead of adding path coverage instrumentation to a model checking run at only around a 12% [12] slowdown. Model checking based testing with SPIN, in our experience, spends over 90% of test time hashing and comparing states. Since only the SUT is instrumented, and not the model checker itself, this keeps overheads for even very expensive instrumentations reasonable. If others apply path coverage to test random or systematic testers with lower computational overhead for the test method itself, we expect this overhead to increase considerably. Using path coverage to test "concolic" testers [22] requires no overhead (such tools already compute explored paths), but is presumably already the primary metric developers use (other than fault detection) in evaluating such tools, though perhaps not from quite the same perspective as in our work. Perhaps most importantly, given that we observed large differences in path coverage in a ten minute test run, we suspect that collecting path coverage is likely to pay off if it prevents overnight runs of faulty testers, even if the overhead for the brief "tester testing" runs is relatively high. For less effective test approaches that generate fewer paths per unit of testing time, the differences may be smaller; however, random testing would appear to be even more amenable to exploitation of parallelism than our state-based testing.

We do note that the memory requirements for storing hundreds of thousands of paths are considerable. Our current implementation naively allocates nearly 2GB for path storage, and makes no attempt to optimize the check for new paths. In practice, we find even this approach efficient enough to use in real test runs, though it would be reasonable to only compute coverage for testing the tester. While there is a possibility of exponential increase in path lengths, we found that allowing for up to 40,000 paths through each top-level function, with up to 13,000 decisions (bits) in each path sufficed even for overnight runs of the test system (and more than sufficed for 10 minute tester evaluation runs). Again, the need to devote perhaps 1 or 2 GB to storing path information may be problematic in other contexts, but compared to memory requirements for model checking (or the need to store all paths already present in most concolic testing), it is not a concern in our case. It is important to remark that we expect keeping "whole test" paths (with the full set of decisions made through the entire test run, annotated with which top-level functions were chosen) would potentially exponentially increase the number of paths to be stored, and impose an unacceptable overhead. In the case of tests that must be considered as whole-program paths (i.e., a single input resulting in a run to termination), we believe that storing paths through individual functions should prevent the exponential explosion from overwhelming test resources. Godefroid has shown this basic approach to work well for concolic testing [8].

3. EXPERIMENTAL RESULTS

Our results are from a tester, based on (non-exhaustive) model checking (model-driven verification of C code [15]) for file systems to be used on the Mars Science Laboratory mission. More details of the tester and the applications tested can be found in our earlier work comparing model checking and random testing [13]. Briefly, the test harness chooses POSIX operations and performs them on two file systems, comparing results. It also introduces hardware faults (bad

Version	Paths	Comments
+ No bad blocks	349,177	No bad blocks allowed
+ No lseek	265,056	Removed the lseek operation from testing
+ Write/read step increase	241,603	Increased granularity of writes/reads to page size (64)
Standard	**240,166**	**Current standard configuration**
+ No close	232,764	Removed the close operation from testing
+ No read	218,961	Removed the read operation from testing
* + No resets	214,666	Without system resets
* Abstraction bug	209,801	Fails to properly distinguish between flash states
* + Broken file descriptor choice	194,289	FD selection fault: results in use of only one file descriptor
+ No unlink	185,946	Removed the unlink operation from testing
+ No initial bad blocks, one bad block fault	190,060	No initial choice, one bad block via fault
- With rename	177,974	Allows rename operations to be performed
- Write/read step decrease	171,496	Decreased granularity of writes/read to 8 bytes
* Alternative random selection	102,424	Randomization method that interacts poorly with swarm search
+ No directories	88,921	Prohibit all directory-creating operations
* Broken path choice	87,693	Path selection code fault, limits pathnames given to operation
* Random testing mode	43,128	Random testing mode
* Only 2 erases allowed	41	Bad commenting in configuration limiting total flash erases
Sequential MC run (12 hours, 4GB memory)	89,976	SPIN without swarm

Table 1: Experimental results

blocks, system resets) that must be properly handled by the SUT. The test harness is approximately 4K lines of code, in PROMELA and C, and the SUTs are approximately 14K lines of C code, in two file systems (NVFS and RAMFS, a flash and RAM file system respectively). SPIN performs a backtracking search of the state-space produced by the test system. The state used is a very coarse abstraction of the flash file system configuration, augmented with path information and some knowledge of the test operation just performed. In general, the model checker is attempting to generate all *states* reachable by file system operations and hardware faults, using unsound abstraction to address the problem of the state-space explosion [5].

The tester is controlled by 130 configuration options that control fault behavior, system abstraction, allowed operations, and other parameters. It may be useful to note that in our results, we *matched* on path coverage. That is, we considered a state new if it resulted from a previously unexplored path through a function, a heuristic we have previously found useful in combination with other search strategies [12]. This biases every test run in favor of improved path coverage. If a test results in even somewhat lower path coverage here, it is a strong indication that it produces fewer opportunities to explore new paths, rather than simply accidentally exploring fewer paths. Assuming that path coverage is desirable, which seems to be a reasonable expectation (i.e., it is a primary justification for DART and similar approaches [9]), there seems to be little reason to ever run the tester without path matching, when we are committed to the memory and time overhead of gathering path information.

Table 1 shows the total path coverage (sum of number of paths through all functions called by the harness) for a number of versions of the tester. In each case, the results are for *only 10 minutes of testing* by the swarm tool [17], configured to use 16GB of memory and 6 processors. Swarm "diversifies" a model checking search by launching a large number of configurations of the search strategy, ideally on a multicore system. The tester versions include both our current standard version, as we began experimentation, some known-bad versions (the regressions which had failed to find high-probability faults in the SUT), and a number of new variations introduced in order to provide comparison. We mark the "known bad" versions with a ***** in the table: these are actual faulty versions of the tester, applied at least once during our testing efforts. *All known bad versions produced lower path coverage, by at least 5,000 paths.* Consider a concrete example: at one point in testing, we attempted to increase the effectiveness of the feedback mechanism controlling the generation of pathnames used in operations [13]. Our intention was to increase the bias of the testing in favor of pathnames that had been used as arguments to a successful `mkdir` or `creat` operation. However, in coding this change, we accidentally modified the inner loop of the path selection code forcing the pathname selection to return only one of two paths (root and a single path in the root directory). The bug reduces the path coverage for a 10 minute test run by over 100,000 paths (240,000 vs. 90,000 paths).

We provide a secondary justification of our assumption that path coverage will serve as a useful indication of fault-detection capability. In our experience [17] an hour swarm run is considerably more effective than an overnight sequential SPIN search, even taking into account the increased parallelism. That is, if we run a non-diversified search for a certain amount of time, and give a parallelized and diversified search time divided by the number of processors (or even less than that), the swarm search is *much* more effective at detecting faults in the SUT. This correlates well with the path coverage for an overnight sequential model checking run, shown at the bottom of the results table. Sequential model checking for 12 hours produces as few paths as the more broken versions of the tester produce in only 10 minutes (rough equivalent of 60 minutes, single-threaded) of test time.

3.1 The Complications of Test Focus

The results not marked with a * suggest a slightly more

complicated picture. *For three variations, the path coverage improves either slightly or considerably over the default tester.* However, we cannot simply consider these to be "better" testers: in two of these cases, we can show that it is *impossible* to detect certain potential faults of the SUT with the variation — e.g., the most effective (in terms of path coverage) tester does not introduce any hardware bad block faults. We call this kind of modification an increase in *test focus*. It seems plausible that, since we cannot explore the full state-space of the SUT, in some cases limiting the sub-graph we consider may increase our efficiency in finding paths through that sub-graph. A further support for this possibility is found in the results for adding the `rename` operation to the test mix, which surprisingly decreases total path coverage.

Even when an increase in focus does not obviously make certain faults undetectable, we cannot be certain it is a gain: for example, limiting the possible sizes of reads and writes to files increases coverage by over 100,000 paths from the standard configuration, but *may* reduce our ability to find boundary condition errors in page-content splitting. Understanding the implications of focus-changing alterations is a challenge (if it were not, designing good testers would be a science rather than, as at present, something of an art).

3.1.1 A Case for Tester Diversification

Rather than focus on finding a single "optimal" tester configuration and design, we suspect that it might be best, in some cases, to exploit a diversity of tester configurations. It is known that search diversity is often critical when we cannot easily make choices about the optimal method for *searching* a state space or performing random walks in random testing [17, 16, 6, 2]. That similar situations might arise in the definition and configuration of the test harness (and thus the state space) itself seems reasonable. Rather than implementing a single monolithic tester and attempting to optimize it, it might be better to define core functionality of the tester but make the *focus* of the search easily changeable. With such a tester, it should be possible to use automated methods very similar to those employed in the swarm tool to diversify testing.

In this case, the existence of path coverage as a relatively inexpensive measure of the effect of a variation might enable more interesting approaches to diversification, based on a meta-search that balances path coverage and the ability to detect a broad class of errors. It might also be useful to use path coverage to guide search diversification in swarm or other similar tools — for example, we suspect that path coverage might be a better basis than error detection for iterative deepening to choose random test run length [2] (due to the very low density of some errors). Further investigation of this topic is clearly in order.

3.1.2 Two Kinds of Tester Modifications

We therefore suggest classing changes to the tester into two groups: changes that alter focus and changes that do not alter focus. For changes that alter focus, a decrease in path coverage does not always indicate a fault in the tester. It may rather indicate that the tester is now more general, and less focused, which may enable it to detect certain errors (but at a cost of decreased efficiency in path exploration). If a change to the tester results in decreased path coverage, but is not expected to decrease tester focus (e.g., a modification in reaction to a change in SUT interface, or a "bug-fix" for the tester's oracle), the path coverage change presumably indicates a fault. Increased path coverage, on the other hand, may not indicate a pure improvement in the tester, if focus removes some faults in the SUT from the scope of testing. As we discuss below, when dealing with changes expected to alter focus, it is best to consider more detailed path information.

3.2 Using More Detailed Path Information

The approach above is all well and good, for the most part, for detecting a regression of our tester. Unfortunately, our baseline is the coverage for some past version of the tester. Raw total path coverage numbers will not help us find flaws in the tester that are present in all past versions and configurations. We may, by accident or design, improve the tester, and discover or verify this improvement using total path coverage. If want to use path coverage to guide improvement, however, we must consider the full vectors of coverage numbers, or even the precise details of which paths have been covered.

3.2.1 An Anecdote: Debugging the Paper Itself

The original version of this paper showed quite different results in Table 1. In part, this was due to basing results on 20 minute test runs, but it was, more significantly, due to an error in the way path coverage was computed. In some cases, a path would persist after the model checker backtracked, causing a path to be assigned to the wrong top-level function, or even to two functions. Fixing this error resulted in a much less ambiguous story: most changes to our standard tester decrease path coverage, as expected.

We detected this error (just before receiving notification that the paper was accepted) by looking at detailed path information. We observed that the results in some cases reported over 1,000 paths through the `unmount` function. Inspection of the code confirmed that at most there were only 64 paths through the function.

3.2.2 Making Focus Visible

Table 2 shows path vectors for a subset of top-level test operations, for the same versions and test runs as Table 1. In this table, we have highlighted the best coverage for each operation in bold. Suddenly, the effects of focus become clear. While removing all directory operations decreases total coverage by over 100,000 paths, it dramatically increases coverage of operations that do not depend on the directory structure of the system: *coverage for* `write`, `read`, `lseek`, *and* `unlink` *is better (by a factor of close to 2 in most cases) than for any other version of the tester.* The "optimal" tester (with best total path coverage), on the other hand, decreases path coverage for all but one of these operations compared to the standard configuration. This further supports our speculations concerning focus diversity. The vector results also show that the truly buggy versions of the tester generally do worse in almost every category: these versions exhibit a fault rather than a trade-off. No faulty version improves on more than one category, over the standard tester. Again, this should work well for fault detection in focus-altering changes: if path coverage decreases for operations that are the intended beneficiaries of focus, we expect that the change is likely faulty.

3.2.3 Going to the Source

Version	mount	write	read	lseek	mkdir	rmdir	creat	unlink
+ No bad blocks	**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
+ Write/read step increase	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
* + No resets	19,200	19,687	74	89	12,132	9,542	10,254	1,655
* Abstraction bug	18,685	16,120	86	142	11,905	10,016	9,427	2,358
* + Broken FD choice	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
+ No initial bad blocks, 1 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
- Write/read step decrease	15,277	13,771	93	134	9,432	6,729	7,345	1,775
* Alternative random selection	7,738	7,693	91	126	4,577	4,275	4,730	2,743
+ No directories	18,116	**59,850**	**160**	**206**	0	0	3,803	**4,707**
* Broken path choice	12,903	12,641	23	104	1,243	1	1,266	1
* Random testing mode	11,026	1,741	61	66	2,197	1,599	2,542	1,246
* Only 2 erases allowed	10	2	3	2	2	1	2	1
Sequential model checking run	7,324	823	9	10	9,062	8,749	6,131	91

Table 2: More detailed coverage results

We speculate that it should be possible to use the detailed path information to guide improvement of the tester, at the cost of considerable effort. For functions with a small number of paths, it is reasonable to simply examine the full set of paths and consider whether any interesting paths are not included. This seems plausible, in our case, for the `read` and `lseek` operations. In cases where the sheer number of paths makes this impossible, we speculate that compression of subpaths through functions called by the top-level function and abstraction of loop paths might reduce the number of paths to a manageable number. This idea of abstraction of paths introduces a final issue, beyond the scope of this paper: all paths are not created equal. Ideally, we would know which paths matter most. However, at present, other than some intuitions in the direction of compositional coverage (function paths rather than whole-program paths) and the idea that loops should be treated differently, there is little empirical or theoretical justification for preferring some paths to others, in general. There is no substitute, at present, for knowledge of the problem domain and SUT.

4. CONCLUSIONS AND FUTURE WORK

As noted in the introduction, one core element of this work may be obvious, or at least widely agreed-upon: path coverage is a plausible measure of the effectiveness of a test suite. A tester's execution produces a test suite, and the effectiveness of that suite is the best "correctness" specification for the tester. Therefore, measuring path coverage is a good method for testing complex automated testers, when simpler methods such as branch or statement coverage do not suffice. We go beyond this (perhaps obvious) conclusion to show that even a *10 minute tester run* can expose faults in the tester, effectively making the question of coverage overhead irrelevant. We also propose a simple scheme for classifying changes to the tester according to whether they are expected to alter test focus, and note that even when focus is altered, total path coverage is somewhat useful, and a vector of coverage for various functions is highly informative.

One obvious area for further research is the idea of test focus and tester diversification. If path coverage is a suitable low-cost measure of tester effectiveness, we may be able to write intelligent diversification systems that exploit this measure, given some ability to trade coverage off with test generality.

Further experimental results to confirm the utility of path coverage for our purposes (and the low overhead of measuring it, at least in model checking contexts) are also in order.

Acknowledgements: The author would like to especially thank Rajeev Joshi for discussions relating to the topic of this paper, and contributions to the development and maintenance of the tester and instrumentation systems. Thanks are also due to Gerard Holzmann and Klaus Havelund for helpful thoughts.

5. REFERENCES

[1] http://mars.jpl.nasa.gov/msl/.

[2] 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.

[3] 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.

[4] 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.

[5] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 2000.

[6] 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.

[7] Matthew B. Dwyer, Suzette Person, and Sebastian Elbaum. Controlling factors in evaluating path-sensitive error detection techniques. In *Foundations of Software Engineering*, pages 92–104, 2006.

[8] Patrice Godefroid. Compositional dynamic test generation. In *Principles of Programming Languages*, pages 47–54, 2007.

[9] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Programming Language Design and Implementation*, pages 213–223, 2005.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] Richard Hamlet. Random testing. In *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.

[15] Gerard Holzmann and Rajeev Joshi. Model-driven software verification. In *SPIN Workshop on Model Checking of Software*, pages 76–91, 2004.

[16] Gerard Holzmann, Rajeev Joshi, and Alex Groce. Swarm verification. In *Automated Software Engineering*, pages 1–6, 2008.

[17] 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.

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

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

[20] George Necula, Scott McPeak, Shree P. 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.

[21] 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.

[22] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *Foundations of Software Engineering*, pages 262–272, 2005.

[23] 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.

[24] 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.