2014 IEEE International Conference on Software Testing, Verification, and Validation

Cause Reduction for Quick Testing

Alex Groce,* Amin Alipour,* Chaoqiang Zhang,* Yang Chen,† and John Regehr,†
*School of Electrical Engineering and Computer Science
Oregon State University {grocea,zhangch,alipourm}@onid.orst.edu
†School of Computing
University of Utah
{chenyang,regehr}@cs.utah.edu

***Abstract*—In random testing, it is often desirable to produce a "quick test" — an extremely inexpensive test suite that can serve as a frequently applied regression and allow the benefits of random testing to be obtained even in very slow or oversubscribed test environments. Delta debugging is an algorithm that, given a failing test case, produces a smaller test case that also fails, and typically executes much more quickly. Delta debugging of random tests can produce effective regression suites for previously detected faults, but such suites often have little power for detecting new faults, and in some cases provide poor code coverage. This paper proposes extending delta debugging by simplifying tests with respect to code coverage, an instance of a generalization of delta debugging we call *cause reduction*. We show that test suites reduced in this fashion can provide very effective quick tests for real-world programs. For Mozilla's SpiderMonkey JavaScript engine, the reduced suite is *more* effective for finding software faults, even if its reduced runtime is not considered. The effectiveness of a reduction-based quick test persists through major changes to the software under test.**

I. Introduction

In testing a flash file system implementation that eventually evolved into the file system for the Mars Science Laboratory (MSL) project's Curiosity rover [1], [2], one of the authors of this paper discovered that, while an overnight sequence of random tests was effective for shaking out even subtle faults, random testing was not very effective if only a short time was available for testing. Each individual random test was a highly redundant, ineffective use of testing budget. As a basic sanity check/smoke test before checking a new version of the file system in, it was much more effective to run a regression suite built by applying delta debugging [3] to a representative test case for each fault previously found.

Delta debugging (or delta-minimization) is an algorithm (called *ddmin*) for reducing the size of failing test cases. Delta debugging algorithms have retained a common core since the original proposal of Hildebrandt and Zeller [4]: use a variation on binary search to remove individual components of a failing test case t to produce a *new* test case t_{1min} satisfying two properties: (1) t_{1min} fails and (2) removing any component from t_{1min} results in a test case that does not fail. Such a test case is called *1-minimal*. Because 1-minimal test cases are potentially much larger than the smallest possible set of failing components, we say that *ddmin reduces* the size of a test case, rather than truly minimizing it. While the precise details of *ddmin* and its variants can be complex, the family of delta debugging algorithms can generally be simply described. Ignoring caching and the details of an effective divide-and-conquer strategy for constructing candidate test cases, *ddmin* for a base failing test case t_b proceeds by iterating the following two steps until termination:

1) Construct the next candidate simplification of t_b, which we call t_c. Terminate if no t_c remain (t_b is 1-minimal).
2) Execute t_c by calling $rtest(t_c)$. If $rtest$ returns ✗ (the test fails) then it is a simplification of t_b. Set $t_b = t_c$.

In addition to detecting actual regressions of the NASA code, *ddmin*-minimized test cases obtained close to 85% statement coverage in less than a minute, which running new random tests often required hours to match. Unfortunately, the delta debugging-based regression was often *ineffective* for detecting *new* faults unrelated to previous bugs. Inspecting minimized test cases revealed that, while the tests covered most statements, the tests were extremely focused on corner cases that had triggered failures, and sometimes missed very shallow bugs easily detected by a short amount of more new random testing. While the bug-based regression suite was effective as a pure *regression* suite, it was ineffective as a quick way to find *new* bugs; on the other hand, running new random tests was sometimes very slow for detecting either regressions or new bugs.

The functional programming community has long recognized the value of very quick, if not extremely thorough, random testing during development, as shown by the wide use of the QuickCheck tool [5]. QuickCheck is most useful, however, for data structures and small modules, and works best in combination with a functional style allowing modular checks of referentially transparent functions. Even using feedback [6], [1], swarm testing [7], or other improvements to standard random testing, it is extremely hard to randomly generate effective tests for complex systems software such as compilers [8] and file systems [1], [2] without a large test budget. For example, even tuned random testers show increasing fault detection with larger tests, which limits the number of tests that can be run in a small budget [9], [8]. The value of the *ddmin* regressions at NASA, however, suggests a more tractable problem: *given* a set of random tests, generate a truly *quick test* for complex systems software. Rather than choose a particular test budget that represents "the" quick test problem, we propose that quick testing is testing with a budget that is at most half as large as a full test budget, and typically more than

978-0-7695-5185-2/14 $31.00 © 2014 IEEE
DOI 10.1109/ICST.2014.37

an order of magnitude smaller. Discussion with developers and the authors' experience suggested two concrete values to use in evaluating quick test methods. First, tests that take only 30 seconds to run can be considered almost without cost, and executed after, e.g., every compilation. Second, a 5 minute budget is too large to invoke with such frequency, but maps well to short breaks from coding (e.g. the time it takes to get coffee), and is suitable to use before relatively frequent code check-ins. The idea of a quick test is inherent in the concept of test *efficiency*, defined as coverage/fault detection *per unit time* [10], [11], as distinguished from absolute effectiveness, where large test suites will always tend to win.

The primary, practical, contribution of this paper is *a proposed method for solving the quick test problem*, based on test case reduction with respect to code coverage (and simple coverage-based test case prioritization). Generalizing the *effect* in *ddmin* from preserving failure to *code coverage* properties makes it possible to apply *ddmin* to improve test suites containing both failing *and successful* test cases, by dramatically reducing runtime while retaining code coverage. This yields test suites with some of the benefits of the ddmin-regression discussed above (short runtime) but with better overall testing effectiveness. We show that retaining statement coverage can approximate retaining other important effects, including *fault detection and branch coverage*. A large case study based on testing Mozilla's SpiderMonkey JavaScript engine uses real faults to show that cause reduction is effective for improving test efficiency, and that the effectiveness of reduced test cases persists even across a long period of development, without re-running the reduction algorithm. Even more surprisingly, for the version of SpiderMonkey used to perform cause reduction and a version of the code from more than two months later, the reduced suite not only runs almost four times faster than the original suite, but detects *more* distinct faults. A mutation-based analysis of the YAFFS2 flash file system shows that the effectiveness of cause reduction is not unique to SpiderMonkey: a statement-coverage reduced suite for YAFFS2 ran in a little over half the time of the original suite, but killed over 99% as many mutants, including 6 not killed by the original suite.

The second contribution of this paper is introducing the idea of cause reduction, which we believe may have applications beyond improving test suite efficiency.

II. The Quick Test Problem

The *quick test* problem is: given a set of randomly generated tests, produce test suites for test budgets that are sufficiently small that they allow tests to be run frequently during code development, and that maximize:

1) Code coverage: the most important coverage criterion is probably statement coverage; Branch and function coverage are also clearly desirable;
2) Failures: automatic fault localization techniques [12] often work best in the presence of multiple failing test cases; more failures also indicate a higher probability of finding a flaw;
3) Distinct faults detected: finally, the most important evaluation metric is the actual number of *distinct* faults that a suite detects; it is generally better to produce 4 failures, each of which exhibits a distinct fault, than to produce 50 failures that exhibit only 2 different faults [13].

It is acceptable for a quick test approach to require significant pre-computation and analysis of the testing already performed if the generated suites remain effective across significant changes to the tested code without re-computation. Performing 10 minutes of analysis before each 30 second run is clearly unacceptable; performing 10 hours of analysis once to produce quick test suites that remain useful for a period of months is fine. For quick test purposes, it is also probably more feasible to build a generally good small suite rather than perform change analysis on-the-fly to select test cases that need to be executed [14], [15]; the nature of random tests, where tests are all statistically similar (as opposed to human-produced tests which tend to have a goal) means that in practice selection methods tend to propose running most stored test cases. In addition compilers and interpreters tend to pose a difficult problem for change analysis, since optimization passes rely on deep semantic properties of the test case.

Given the highly parallel nature of random testing, in principle arbitrarily many tests could be performed in 5 minutes. In practice, considerable effort is required to introduce and maintain cloud or cluster-based testing, and developers often work offline or can only use local resources due to security or confidentiality concerns. More critically, a truly small quick test would enable testing on slow, access-limited hardware systems; in MSL development, random tests were not performed on flight hardware due to high demand for access to the limited number of such systems [16], and the slowness of radiation-hardened processors. A test suite that only requires 30 seconds to 5 minutes of time on a workstation, however, would be feasible for use on flight testbeds. We expect that the desire for high quality random tests for slow/limited access hardware may extend to other embedded systems contexts, including embedded compiler development. Such cases are more common than may be obvious: for instance, Android GUI random testing [17] on actual mobile devices can be even slower than on already slow emulators, but is critical for finding device-dependent problems. Quick testing's model of expensive pre-computation to obtain highly efficient execution is a good fit for the challenge of testing on slow and/or oversubscribed hardware.

In some cases, the quick test problem might be solved simply by using test-generation techniques that produce short tests in the first place, e.g. evolutionary/genetic testing approaches where test size is included in fitness [18], [19], [20], or bounded exhaustive testing (BET). BET, unfortunately, performs poorly even for file system testing [21] and is very hard to apply to compiler testing. Recent evolutionary approaches [19] are more likely to succeed, but to our knowledge have not been applied to such complex problems as compiler or interpreter testing, where hand-tuned systems requiring expert knowledge are typical [8], [22].

III. Coverage-Based Test Case Reduction

Delta debugging is an attractive approach to the quick test problem, in that it is a highly effective and easy-to-implement method for reducing redundancy in randomly generated tests. Unfortunately, traditional delta debugging reduces tests *too much*, discarding all behavior not related to the failure. Delta debugging can be applied to the quick test problem, however, using a novel generalization. The assumption has always been that delta debugging is a *debugging* algorithm, only useful for reducing failures. However, the best way to understand *ddmin*-like algorithms is that they reduce the size of a *cause* (e.g. a test case, a thread schedule, etc.) while ensuring that it still causes some fixed *effect* (in *ddmin*, the effect is always test failure): *ddmin* is a special-case of *cause reduction*.[1] The core value of delta debugging can be understood in a simple proposition: given two test cases that achieve the same purpose, the smaller of the two test cases will typically be easier to understand, execute more quickly, and so forth. Delta debugging is valuable because, *given two test cases that both serve the same purpose*, we almost always prefer the smaller of the two. There is no reason why purpose should be limited to failure. For quick testing, *code coverage* is a much more attractive property, in that it also helps detect new faults.

The definitions provided in the core delta debugging paper [3] are *almost* unchanged in cause reduction. The one necessary alteration is to replace the function *rtest*, which "takes a program run and tests whether it produces the failure" in Definition 3 [3] with a function *reffect* such that *reffect* defines "failure" of a run as preserving any effect that holds for the original test case and "success" as not preserving that effect. An actual failure is a particular instance of an effect to be preserved. We call this "new" algorithm *cause reduction* but, of course, it is almost exactly the same as the original *ddmin* algorithm, and most optimizations or variations still apply.

The most interesting consequence of this minor change is that *ddmin* is no longer defined only for failing test cases. If the effect chosen is well-defined for successful test cases, then *ddmin* can be applied to reduce the cause (the test case) in that case also. Are any interesting effects defined for all test cases important enough to inspire reduction efforts?

A. Coverage as an Effect

A large portion of the literature on software testing is devoted to precisely such a class of effects: running a test case always produces the effect of *covering* certain source code elements, which can include statements, branches, dataflow relationships, state predicates, or paths. High coverage is a common goal of testing, as high coverage correlates with effective fault detection [23]. Producing *small* test suites with high code coverage [11] has long been a major goal of software testing efforts, inspiring a lengthy literature on how to minimize a test suite with respect to coverage, how to select tests from a suite based on coverage, and how to prioritize a test suite by coverage [14]. Coverage-based minimization reduces a *suite* by removing test cases; using cause reduction, a suite can also (orthogonally) be reduced by minimizing each test in the suite (retaining all tests) with the effect being *any chosen coverage criteria*. The potential benefit of reduction at the test level is the same as at the suite level: more efficient testing, in terms of fault detection or code coverage per unit of time spent executing tests. Cause reduction with respect to coverage is a promising approach for building quick tests, as random tests are likely to be highly reducible.

As described in the introduction, *ddmin* algorithms proceed by generating "candidate" tests: tests that are smaller than the original test case, but may preserve the property of interest, which in the original algorithm is "this test fails." When evaluating the preservation check on a candidate reduced test case returns ✗ (indicating the test failed) *ddmin* essentially starts over, with the candidate test case as the new starting point for reduction, until no candidates fail. Preservation is formulated as follows for coverage-based reduction:

$$reffect(t_c, t_b) = \begin{cases} \text{iff } \forall s \in c(t_b). s \in c(t_c) & \text{✗} \\ \text{else} & \checkmark \end{cases}$$

where t_c is the currently proposed smaller test, t_b is the original test case, and $c(t)$ is the set of all coverage entities executed by t. While it may be confusing that a valid reduction of the test case returns ✗ we maintain the terminology to show how little difference there is between generalized cause reduction and the *ddmin* algorithm; recall that in *ddmin* the point of preservation is to find tests that fail. Returning ✗ in our context means that the new test has preserved coverage and can therefore be used as the basis for further reduction efforts, while ✓ means that the candidate test does *not* preserve coverage, and should be discarded (the fate of any successful test case in the original *ddmin* algorithm). Note that this definition allows a test case to be minimized to a test with *better* coverage than the original test. In practice, improved coverage seems rare: if a smaller test that does not preserve the added coverage can be found, *ddmin* removes gained coverage.

In principle, *any* coverage criteria could be used as an effect. In practice, it is highly unlikely that reducing by extremely fine-grained coverages such as path or predicate coverages [23] would produce significant reduction. Moreover, *ddmin* is a very expensive algorithm to run when test cases do not reduce well, since every small reduction produces a new attempt to establish 1-minimality: small removals tend to result in a very large computational effort proportional to the reduction. Additionally, for purposes of a quick test, it seems most important to concentrate on coverage of coarse entities, such as statements. Finally, only branch and statement coverage are widely enough implemented for languages that it is safe to assume anyone interested in producing a quick test has tools to support their use. For random testing, which is often carried out by developers or by security experts, this last condition is important: lightweight methods that do not require static or dynamic analysis expertise and are easy to implement from scratch are more likely to be widely applied [24].

[1] The authors would like to thank Andreas Zeller for suggesting the term "cause reduction."

TABLE I
SPIDERMONKEY UNLIMITED TEST BUDGET RESULTS

Release	Date	Suite	Size	Time(s)	ST	BR	FN	#Fail	E#F
1.6	12/22/2006	Full	13,323	14,255.068	19,091	**14,567**	966	1,631	22
1.6	12/22/2006	ST-Min	13,323	**3,566.975**	19,091	14,562	966	1,631	**43**
1.6	12/22/2006	DD-Min	1,019	169.594	16,020	10,875	886	1,019	22
1.6	12/22/2006	GE-ST(Full)	168	182.823	19,091	14,135	966	14	5
1.6	12/22/2006	GE-ST(ST-Min)	171	47.738	19,091	14,099	966	14	8
NR	2/24/2007	Full	13,323	9,813.781	**22,392**	**17,725**	**1,072**	**8,319**	20
NR	2/24/2007	ST-Min	13,323	**3,108.798**	22,340	17,635	1,070	4,147	**36**
NR	2/24/2007	DD-Min	1,019	148.402	17,923	12,847	958	166	7
NR	2/24/2007	GE-ST(Full)	168	118.232	21,305	16,234	1,044	116	5
NR	2/24/2007	GE-ST(ST-Min)	171	40.597	21,323	16,257	1,045	64	3
NR	4/24/2007	Full	13,323	16,493.004	**22,556**	**18,047**	**1,074**	189	**10**
NR	4/24/2007	ST-Min	13,323	**3,630.917**	22,427	17,830	1,070	**196**	6
NR	4/24/2007	DD-Min	1,019	150.904	18,032	12,979	961	158	5
NR	4/24/2007	GE-ST(Full)	168	206.033	22,078	17,203	1,064	4	1
NR	4/24/2007	GE-ST(ST-Min)	171	45.278	21,792	16,807	1,058	3	1
1.7	10/19/2007	Full	13,323	14,282.776	**22,426**	**18,130**	**1,071**	**528**	**15**
1.7	10/19/2007	ST-Min	13,323	**3,401.261**	22,315	17,931	1,067	274	10
1.7	10/19/2007	DD-Min	1,019	168.777	18,018	13,151	956	231	12
1.7	10/19/2007	GE-ST(Full)	168	178.313	22,001	17,348	1,061	6	2
1.7	10/19/2007	GE-ST(ST-Min)	171	43.767	21,722	16,924	1,055	5	2
1.8.5	3/31/2011	Full	13,323	4,301.674	**21,030**	**15,854**	**1,383**	**11**	**2**
1.8.5	3/31/2011	ST-Min	13,323	**2,307.498**	20,821	15,582	1,363	3	1
1.8.5	3/31/2011	DD-Min	1,019	152.169	16,710	11,266	1,202	2	1
1.8.5	3/31/2011	GE-ST(Full)	168	51.611	20,233	14,793	1,338	1	1
1.8.5	3/31/2011	GE-ST(ST-Min)	171	28.316	19,839	14,330	1,327	1	1

Legend: ST = Statement Coverage; BR = Branch Coverage; FN = Function Coverage; #Fail = Num. Failing Tests; E#F = Estimated Num. of Distinct Faults
Full = Original Suite; ST-Min = *ddmin*(Full, ST Cov.); DD-Min = *ddmin*(Full, Failure); GE-ST = Greedy Selection for ST. Cov

IV. SPIDERMONKEY JAVASCRIPT ENGINE CASE STUDY

SpiderMonkey is the JavaScript Engine for Mozilla, an extremely widely used, security-critical interpreter/JIT compiler. SpiderMonkey has been the target of aggressive random testing for many years now. A single fuzzing tool, `jsfunfuzz` [22], is responsible for identifying more than 1,700 previously unknown bugs in SpiderMonkey [25]. SpiderMonkey is (and was) very actively developed, with over 6,000 code commits in the period from 1/06 to 9/11 (nearly 4 commits/day). SpiderMonkey is thus ideal for evaluating a quick test approach, using the last public release of the `jsfunfuzz` tool, modified for swarm testing [7]. Figures 1 and 2 show cause reduction by statement coverage in action. The first figure is a short test generated by `jsfunfuzz`; the second is a test case based on it, produced by *ddmin* using statement coverage as effect. These tests both cover the same 9,625 lines code. While some reductions are easily predictable (e.g. `throw StopIteration`), others are highly non-obvious, even to a developer.

```
tryItOut("with((delete __proto__))
         {export __parent__;true;}");
tryItOut("while((false for (constructor in false))){}");
tryItOut("throw __noSuchMethod__;");
tryItOut("throw undefined;");
tryItOut("if(<><x><y/></x></>) {null;}else{/x/;/x/g;}");
tryItOut("{yield;export __count__; }");
tryItOut("throw StopIteration;");
tryItOut("throw StopIteration;");
tryItOut(";yield;");
```

Fig. 1. `jsfunfuzz` test case before statement coverage reduction

```
tryItOut("with((delete __proto__))
         {export __parent__;true;}");
tryItOut("while((false for (constructor in false))){}");
tryItOut("throw undefined;");
tryItOut("if(<><x><y/></x></>) {null;}else{/x/;/x/g;}");
tryItOut("throw StopIteration;");
tryItOut(";yield;");
```

Fig. 2. `jsfunfuzz` test case after statement coverage reduction

The baseline test suite for SpiderMonkey is a set of 13,323 random tests, produced during 4 hours of testing the 1.6 source release of SpiderMonkey. These tests constitute what is referred to below as the **Full** test suite. Running the **Full** suite is essentially equivalent to generating new random tests of SpiderMonkey. A reduced suite with equivalent statement coverage, referred to as **Min**, was produced by performing cause reduction on every test in **Full**. The granularity of minimization was based on the semantic units produced by `jsfunfuzz`, with 1,000 such units in each test in **Full**. A unit is the code inside each `tryItOut` call, approximately 1 line of code. After reduction, the average test case size was just over 122 semantic units, a bit less than an order of magnitude reduction; while increases in coverage were allowed, in 99%

of cases coverage was identical to the original test. The computational cost of cause reduction was, on contemporary hardware, similar to the costs of traditional delta debugging reported in older papers, around 20 minutes per test case [26]. The entire process completed in less than 4 hours on a modestly sized heterogeneous cluster (using fewer than 1,000 nodes). The initial plan to also minimize by branch coverage was abandoned when it became clear that statement-based minimization tended to almost perfectly preserve total suite branch coverage. Branch-based minimization was also much slower and typically reduced test case size by a factor of only 2/3, vs. nearly 10x reduction for statements.

A third suite, referred to as **DD-Min** (Delta Debugging Minimized), was produced by taking all 1,631 failing test cases in **Full** and reducing them using *ddmin* with the requirement that the test case fail and produce the same failure output as the original test case. After removing numerous duplicate tests, **DD-Min** consisted of 1,019 test cases, with an average size of only 1.86 semantic units (the largest test contained only 9 units). Reduction in this case only required about 5 minutes per test case. Results below show why **DD-Min** was not included in experimental evaluation of quick test methods (essentially, it provided extremely poor code coverage, leaving many very shallow bugs potentially uncaught; it also fails to provide enough tests for a 5 minute budget).

Two additional small suites, **GE-ST(Full)** and **GE-ST(Min)** were produced by applying Chen and Lau's GE heuristic [27] for coverage-based suite minimization to the **Full** and **Min** suites. The GE heuristic first selects all test cases that are essential (i.e., they uniquely cover some coverage entity), then repeatedly selects the test case that covers the most additional entities, until the coverage of the minimized suite is equal to the coverage of the full suite (i.e., an additional greedy algorithm, seeded with test cases that must be in any solution). Ties are broken randomly in all cases.

The evaluation measures for suites are: size (in # tests), statement coverage (ST), branch coverage (BR), function coverage (FN), number of failing tests (#Fail), and estimated number of faults (E#F). All coverage measures were determined by running gcov (which was also used to compute coverage for *reffect*). Failures were detected by the various oracles in `jsfunfuzz` and, of course, detecting crashes and timeouts.

Distinct faults detected by each suite were estimated using a binary search over all source code commits made to the SpiderMonkey code repository, identifying, for each test case, a commit such that: (1) the test fails before the commit and (2) the test succeeds after the commit. With the provision that we have not performed extensive hand-confirmation of the results, this is similar to the procedure used to identify bugs in previous work investigating the problem of ranking test cases such that tests failing due to different underlying faults appear early in the ranking [13]. This method is not always precise. It is, however, uniform and has no obvious problematic biases. Its greatest weakness is that if two bugs are fixed in the same check-in, they will be considered to be "one fault"; the estimates of distinct faults are therefore best viewed as *lower* bounds on actual distinct faults. In practice, hand examination of tests in previous work suggested that the results of this method are fairly good approximations of the real number of distinct faults detected by a suite. Some bugs reported may be faults that developers knew about but gave low priority; however, more than 80 failures result in memory corruption, indicating a potential security flaw, and all faults identified were fixed at some point during SpiderMonkey development.

In order to produce 30 second and 5 minute test suites (the extremes of the likely quick test budget), it was necessary to choose subsets of **Full** and **Min**. The baseline approach is to randomly sample a suite, an approach to test case prioritization used as a baseline in numerous previous test case prioritization and selection papers [14]. While a large number of plausible prioritization strategies exist, we restricted our study to ones that do not require analysis of faults, expensive mutation testing, deep static analysis, or in fact any tools other than standard code coverage. As discussed above, we would like to make our methods as lightweight and generally applicable as possible. We therefore chose four coverage-based prioritizations from the literature [14], [28], which we refer to as ΔST, $|$ST$|$, ΔBR, and $|$BR$|$. ΔST indicates a suite ordered by the incremental improvement (Δ) in statement coverage offered by each test over all previous tests (an additional greedy algorithm), while $|$ST$|$ indicates a suite ordered by the absolute statement coverage of each test case (a pure greedy algorithm). The first test executed for both ΔST and $|$ST$|$ will be the test with the highest total statement coverage. ΔBR and $|$BR$|$ are similar, except ordered by different coverage.

Finally, a key question for a quick test method is how long quick tests remain effective. As code changes, a cause reduction and prioritization based on tests from an earlier version of the code will (it seems likely) become obsolete. Bug fixes and new features (especially optimizations in a compiler) will cause the same test case to change its coverage, and over time the basic structure of the code may change; SpiderMonkey itself offers a particularly striking case of code change: between release version 1.6 and release version 1.8.5, the vast majority of the C code-base was re-written in C++. All experiments were therefore performed not only on SpiderMonkey 1.6, the baseline for cause reduction, but applied to "future" (from the point of view of quick test generation) versions of the code. The first two versions are internal source commits, not release versions (NR for non-release), dating from approximately two months (2/24/2007) and approximately four months (4/24/2007) after the SpiderMonkey 1.6 release (12/22/2006). When these versions showed that quick tests retained considerable power, it indicated that a longer lifetime than we had hoped for might be possible. The final two versions of SpiderMonkey chosen were therefore the 1.7 release version (10/19/2007) and the 1.8.5 release version (3/31/2011). Note that all suites were reduced and prioritized based on the 1.6 release code; no re-reduction or re-prioritization was ever applied.

TABLE II
SPIDERMONKEY 30S TEST BUDGET MEAN RESULTS

Ver.	Suite	Size	ST	BR	FN	#Fail	E#F
1.6	Full(F)	27.1	16,882.1	11,895.4	897.0	2.8	2.8
1.6	F+ΔST	27.6	18,270.5	13,050.1	949.9	4.2	4.0
1.6	F+ΔBR	25.7	18,098.2	13,144.7	936.4	2.8	2.6
1.6	Min(M)	102.2	17,539.4	12,658.6	916.6	**12.6**	7.1
1.6	M+ΔST	**106.9**	**18,984.7**	**13,873.6**	**963.1**	12.0	**9.0**
1.6	M+ΔBR	77.3	18,711.6	13,860.9	958.8	7.3	5.4
2/24	Full(F)	37.8	19,718.0	14,644.9	991.6	23.9	3.1
2/24	F+ΔST	45.1	19,958.0	14,813.9	1,006.0	35.1	3.0
2/24	F+ΔBR	39.4	20,502.2	15,511.6	1,021.8	23.5	4.4
2/24	Min(M)	105.0	20,319.3	15,303.5	1,013.2	32.2	4.0
2/24	M+ΔST	92.9	**21,238.1**	15,984.8	**1,049.1**	35.6	2.7
2/24	M+ΔBR	**117.2**	21,167.2	**16,183.9**	1,042.0	**46.4**	**5.0**
4/24	Full(F)	23.8	20,072.8	15,108.5	999.0	0.6	0.6
4/24	F+ΔST	25.3	21,111.7	15,948.7	1,040.3	2.0	**2.0**
4/24	F+ΔBR	25.8	21,101.4	16,122.2	1,037.8	2.0	**2.0**
4/24	Min(M)	100.8	20,485.7	15,564.3	1,016.6	1.6	1.6
4/24	M+ΔST	**113.5**	**21,731.8**	16,631.1	**1,056.9**	2.0	**2.0**
4/24	M+ΔBR	105.4	21,583.7	**16,763.8**	1,056.4	**3.0**	**2.0**
1.7	Full(F)	27.5	20,061.6	15,288.4	1,002.0	1.4	1.4
1.7	F+ΔST	30.0	21,112.9	16,140.3	1,042.0	**4.0**	**3.0**
1.7	F+ΔBR	29.2	21,047.3	16,280.5	1,036.3	2.0	2.0
1.7	Min(M)	103.5	20,416.8	15,675.1	1,015.7	1.8	1.8
1.7	M+ΔST	**116.4**	**21,668.4**	16,762.6	**1,054.0**	**4.0**	**3.0**
1.7	M+ΔBR	109.7	21,535.6	**16,908.7**	1,053.8	**4.0**	**3.0**
1.8.5	Full(F)	83.4	19,300.8	13,907.5	1,291.4	0.0	0.0
1.8.5	F+ΔST	98.8	19,876.9	14,430.8	1,320.4	**1.0**	**1.0**
1.8.5	F+ΔBR	98.0	19,963.1	**14,494.2**	1,326.0	**1.0**	**1.0**
1.8.5	Min(M)	140.8	19,043.3	13,621.1	1,286.0	0.0	0.0
1.8.5	M+ΔST	**179.4**	19,848.2	14,338.0	1,325.0	**1.0**	**1.0**
1.8.5	M+ΔBR	178.3	**19,975.8**	14,453.0	**1,329.0**	**1.0**	**1.0**

Legend: ST=Statement Cov.; BR=Branch Cov.; FN=Func. Cov.; #Fail=Num. Failing Tests; E#F=Est. Num. Distinct Faults; Full/F=Original Suite; Min/M=*ddmin*(Full, ST Cov.); ΔST=Inc. ST Cov. Prioritization, ΔBR=Inc. BR Prior.

TABLE III
SPIDERMONKEY 5M TEST BUDGET MEAN RESULTS

Ver.	Suite	Size	ST	BR	FN	#Fail	E#F
1.6	Full(F)	269.4	17,993.2	13,227.5	933.2	32.6	7.4
1.6	F+ΔST	270.2	19,093.0	14,195.9	966.0	23.0	8.0
1.6	F+ΔBR	272.1	19,064.2	14,504.3	962.0	24.0	9.0
1.6	Min(M)	1,001.2	18,493.2	13,792.4	949.8	121.1	18.8
1.6	M+ΔST	1,088.9	**19,093.0**	14,298.4	**966.0**	138.7	**22.9**
1.6	M+ΔBR	**1,093.1**	19,091.0	**14,563.2**	964.0	**146.3**	20.9
2/24	Full(F)	381.4	21,175.5	16,308.2	1,037.6	237.8	8.3
2/24	F+ΔST	404.5	21,554.0	16,612.1	1,051.0	258.9	7.0
2/24	F+ΔBR	398.7	21,664.2	16,833.1	1,051.0	252.6	8.0
2/24	Min(M)	1,124.9	21,556.8	16,711.3	1,051.1	347.9	10.6
2/24	M+ΔST	**1,255.6**	21,899.8	17,021.9	**1,064.0**	**383.6**	**15.0**
2/24	M+ΔBR	1,227.7	**21,940.0**	**17,180.0**	1,058.1	356.5	12.0
4/24	Full(F)	237.8	21,430.2	16,663.0	1,043.8	7.8	2.7
4/24	F+ΔST	244.7	22,139.0	17,279.3	1,064.0	7.0	2.0
4/24	F+ΔBR	241.2	22,126.8	17,483.3	1,064.0	6.1	3.0
4/24	Min(M)	1,085.6	21,695.8	16,960.1	1,051.4	16.0	2.9
4/24	M+ΔST	1,113.8	22,106.9	17,308.0	**1,065.3**	**18.0**	**5.0**
4/24	M+ΔBR	**1,135.1**	**22,178.0**	**17,550.5**	1,063.0	17.1	3.0
1.7	Full(F)	263.7	21,350.0	16,796.8	1,042.2	10.9	3.6
1.7	F+ΔST	282.1	22,074.0	17,438.1	**1,063.0**	17.8	4.0
1.7	F+ΔBR	278.6	**22,087.5**	**17,670.1**	1,061.0	11.0	5.0
1.7	Min(M)	1,072.9	21,616.9	17,070.0	1,050.4	22.2	4.8
1.7	M+ΔST	**1,186.3**	22,025.0	17,425.7	**1,063.0**	**26.1**	6.0
1.7	M+ΔBR	1,165.8	22,082.3	17,676.6	1,060.0	24.0	**7.0**

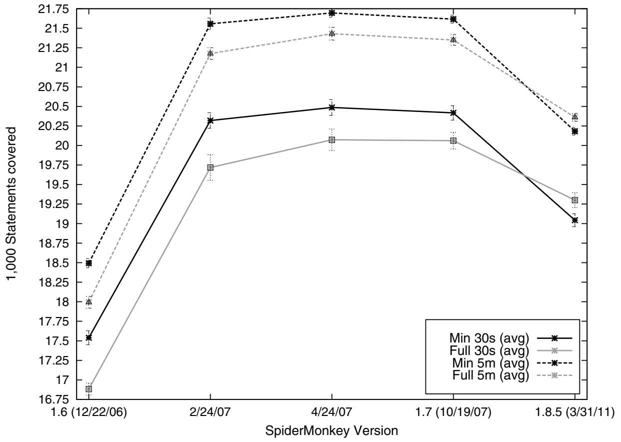

Fig. 3. ST coverage for 30s and 5m quick tests across SpiderMonkey versions

A. Results: An Effective Quick Test?

Table I provides information on the base test suites across the five versions of SpiderMonkey studied. Tables II and III show how each proposed quick test approach performed on each version, for 30 second and 5 minute test budgets, respectively. All nondeterministic or time-limited experiments were repeated 30 times. The differences between minimized (M) suite and full suite (F) for each method and budget are statistically significant at a 95% level, under a two-tailed U-test, with only one exception: the improvement in fault detection for the non-prioritized suites for the 4/24 version is not significant. The best results for each suite attribute, SpiderMonkey version, and test budget combination are shown in bold (ties are only shown in bold if some approaches did not perform as well as the best methods). Results for absolute coverage prioritization are omitted from the table to save space, as Δ prioritization always performed much better, and absolute often performed worse than random selection. Results for version 1.8.5 are also omitted from the 5 minute budget results as the 30 second results suffice to show that minimized tests and prioritizations based on version 1.6 are, as expected, not as useful after 4 additional years of development, though still sometimes improving on the full suite.

The results are fairly striking. First, a purely failure-based quick test such as was used at NASA (**DD-Min**) produces very poor code coverage (e.g., covering almost 100 fewer *functions* than the original suite, and over 3,000 fewer branches). It also loses fault detection power rapidly, only finding ∼7 distinct faults on the next version of the code base, while suites based on all tests can detect ∼20-∼36 faults. Given its extremely short runtime, retaining such a suite as a pure regression may be useful, but it cannot be expected to work as a good quick test. Second, the suites greedily minimized by statement coverage (**GE-ST(Full)** and **GE-ST(Min)**) are very quick, and potentially useful, but lose a large amount of branch coverage and do not provide enough tests to fill a 5 minute quick test. The benefits of suite minimization by statement coverage (or branch coverage) were represented in the 30 second and 5 minute budget experiments by the Δ prioritizations, which produce the same results, with the exception that for short budgets tests included because they uniquely cover some entity are less likely to be included than with random sampling of the minimized suites.

The most important total suite result is that the cause reduced **Min** suite retains (or improves!) many properties of the **Full** suite that are *not* guaranteed to be preserved by our modified *ddmin* algorithm. For version 1.6, only 5 branches are "lost", and (most strikingly) the number of failing test cases is *unchanged*. Most surprisingly, the estimated distinct fault detection is *improved*: it has grown from $\sim$22 faults to $\sim$43 faults. The difference in results is highly statistically significant: dividing the test populations into 30 equal-sized randomly selected test suites for both full and minimized tests we find that the average minimized suite detects 11.83 distinct faults on average, while the average full suite only detects 7.6 faults, with a p-value of $5.2 \cdot 10^{-10}$ under U-test. It is difficult to believe that any bias in the fault estimation method produces this strong an effect. Our best hypothesis as to the cause of the remarkable failure preservation level is that *ddmin* tends to preserve failure because failing test cases have unusually *low* coverage in many cases. Since the *ddmin* algorithm attempts to minimize test size, this naturally forces it to attempt to produce reduced tests that also fail; moreover, some failures execute internal error handling code (many do not, however — the numerous test cases violating `jsfunfuzz` semantic checks, for example). The apparent increased diversity of faults, however, is surprising and unusual, and suggests that the use of *ddmin* as a test mutation-based fuzzing tool might be a promising area for future research. In retrospect, it is obvious that *ddmin* takes as input a test case and generates a large number of related, but distinct, new test cases — it is, itself, a test case generation algorithm. It seems safe to say that the new suite is essentially as good at detecting faults and covering code, with much better runtime (and therefore better test efficiency [10]).

Figure 3 graphically exhibits the raw differences in statement coverage for the suites sampled as quick tests, ignoring the effects of prioritization, with 1 standard-deviation error bars on points. The power of coverage-based cause reduction can be seen in Tables II and III by comparing "equivalent" rows for any version and budget: results for each version are split so that **Full** results are the first three rows and the corresponding prioritization the for **Min** tests are the next three rows. For the first three versions tested, it is almost always the case that for every measure, the reduced suite value is better than the corresponding full suite value. For 30s budgets this comparison even holds true for the 1.7 version, nearly a year later. Moving from 1.6 to 1.7 involves over 1,000 developer commits and the addition of 10,000+ new lines of code (a 12.5% increase). In reality, it is highly unlikely that developers would not have a chance to produce a better baseline on more similar code in a four year period (or, for that matter, in any one month period). The absolute effect size, as measured by the lower bound of a 95% confidence interval, is often large – typically 500+ lines and branches and 10 or more functions, and in a few cases more than 10 faults.

TABLE IV
YAFFS2 RESULTS

Suite	Size	Time(s)	ST	BR	FN	MUT
Full	4,240	729.032	4,049	1,925	332	616
Min	4,240	402.497	4,049	1,924	332	611
Full(F)	174.4	30.0	4,007.367	1,844.0	332.0	568.3
F+ΔST	372.5	30.0	**4,049.0**	1,918.0	332.0	594.0
F+ΔBR	356	30.0	**4,049.0**	**1,925.0**	332.0	**596.0**
F+|ST|	112.5	30.0	4,028.0	1,889.0	332.0	589.0
Min(M)	315.8	30.0	4,019.7	1,860.5	332.0	559.0
M+ΔST	**514.7**	30.0	**4,049.0**	1,912.0	332.0	571.0
M+ΔBR	500.0	30.0	**4,049.0**	1,924.0	332.0	575.0
M+|ST|	255.0	30.0	4,028.0	1,879.0	332.0	552.0
Full(F)	1,746.8	300.0	4,044.7	1,916.0	332.0	608.7
F+ΔST	2,027.0	300.0	**4,049.0**	1,921.0	332.0	601.0
F+ΔBR	2,046.0	300.0	**4,049.0**	**1,925.0**	332.0	604.0
F+|ST|	1,416.0	300.0	4,042.0	1,916.0	332.0	**611.0**
Min(M)	3,156.6	300.0	4,048.1	1,920.0	332.0	607.1
M+ΔST	**3,346.0**	300.0	**4,049.0**	1,924.0	332.0	601.0
M+ΔBR	3,330.0	300.0	**4,049.0**	1,924.0	332.0	605.0
M+|ST|	2,881.7	300.0	**4,049.0**	1,924.0	332.0	**611.0**

It is difficult to generalize from one subject, but based on the SpiderMonkey results, we believe that a good initial quick test strategy to try for other projects would be to combine cause reduction by statement coverage with test case prioritization by either Δ statement or branch coverage. In fact, limitation of quick tests to very small budgets may not be critical. Running only 7 minutes of minimized tests on version 1.6 detects an average of twice as many faults as running 30 minutes of full tests and has (of course) indistinguishable average statement and branch coverage. The difference is significant with p-value of $2.8 \cdot 10^{-7}$ under a U-test. In general, for SpiderMonkey versions close to the baseline, running N minutes of minimized tests, however selected, seems likely to be much better than running N minutes of full tests. The real limitation is probably how many minimized tests are available to run, due to the computational cost of minimizing tests.

V. YAFFS 2.0 Flash File System Case Study

YAFFS2 [29] is a popular open-source NAND flash file system for embedded use; it was the default image format for early versions of the Android operating system. Lacking a large set of real faults in YAFFS2, we applied mutation testing to check our claim that cause reduction not only preserves source code coverage, but tends to preserve fault detection and other useful properties of randomly generated test cases. The evaluation used 1,992 mutants, randomly sampled from the space of all 15,246 valid YAFFS2 mutants, using the C mutation software shown to provide a good proxy for fault detection [30], with a sampling rate (13.1%) above the 10% threshold suggested in the literature [31]. Sampled mutants were not guaranteed to be killable by the API calls and emulation mode tested. Table IV shows how full and quick test suites for YAFFS2 compared. MUT indicates the number of mutants killed by a suite. Results for $|\text{BR}|$ are omitted, as absolute prioritization by branch coverage produced an equivalent suite to absolute prioritization by statement coverage. Runtime reduction for YAFFS2 was not as high as with SpiderMonkey tests (1/2 reduction vs. 3/4), due to a smaller change in test size and higher relative cost of test startup. The average length of original test cases was 1,004 API calls, while reduced tests averaged 213.2 calls. The most likely cause of the smaller reduction is that the YAFFS2 tester uses a feedback [1] model to reduce irrelevant test operations. Basic retention of desirable aspects of **Full** was, however, excellent: only one branch was "lost", function coverage was perfectly retained, and 99.1% as many mutants were killed. The reduced suite killed 6 mutants not killed by the original suite. We do not know if mutant scores are good indicators of the ability of a suite to find, e.g., subtle optimization bugs in compilers. Mutant kills *do* seem to be a reliable method for estimating the ability of a suite to detect many of the shallow bugs a quick test aims to expose before code is committed or subjected to more testing. Even with lesser efficiency gains, cause reduction plus *absolute* coverage prioritization is by far the best way to produce a 5 minute quick test, maximizing 5-minute mutant kills without losing code coverage. *All* differences in methods were significant, using a two-tailed U-test (in fact, the highest p-value was 0.0026).

VI. GCC: The Potentially High Cost of Reduction

Finally, we attempted to apply cause reduction to test cases produced by Csmith [8] using the GCC 4.3.0 compiler (released 3/5/2008), using C-Reduce [32] modified to attempt only line-level reduction, since we hypothesized that reducing C programs would be more expensive than reducing SpiderMonkey or YAFFS2 test cases, which have a simpler structure. Our hypothesis proved more true than we had anticipated: after 6 days of execution (on a single machine rather than a cluster), our reduction produced only 12 reduced test cases! The primary problem is twofold: first, each run of GCC takes longer than the corresponding query for SpiderMonkey or YAFFS2 tests, due to the size and complexity of GCC (tests are covering 161K+ lines, rather than only about 20K as in SpiderMonkey) and the inherent start up cost of compiling even a very small C program. Second, the test cases themselves are larger — an average of 2,222 reduction units (lines) vs. about 1,000 for SpiderMonkey and YAFFS — and reduction fails more often than with the other subjects.

While 12 reduced test cases do not make for a particularly useful data set, the results for these instances did support the belief that reduction with respect to statement coverage preserves interesting properties. First, the 12 test cases selected all crashed GCC 4.30 (with 5 distinct faults, in this case confirmed and examined by hand); after reduction, the test cases were reduced in size by an average of 37.34%, and all tests still crashed GCC 4.3.0 with the same faults. For GCC 4.4.0 (released 4/21/2009), no test cases in either suite caused the compiler to fail, and the reduced tests actually covered 419 *more* lines of code when compiled. Turning to branch coverage, an even more surprising result appears: the minimized tests cover an additional 1,034 branches on GCC 4.3.0 and an additional 297 on 4.4.0. Function coverage is also *improved* in the minimized suite for 4.4.0: 7,692 functions covered in the 12 minimized tests vs. only 7,664 for the original suite. Unfortunately the most critical measure, the gain in test efficiency, was marginal: for GCC 4.3.0, the total compilation time was 3.23 seconds for the reduced suite vs. 3.53 seconds for the original suite, though this improved to 6.35s vs 8.78s when compiling with GCC 4.4.0. Even a 37.34% size reduction does not produce large runtime improvement, due to the high cost of starting GCC. However, the added value of the reduced tests is high enough that we are (1) rewriting portions of C-Reduce to execute much faster and (2) planning to devote a large computing budget to minimizing a high-coverage Csmith-produced suite for the latest versions of GCC and LLVM. It is unclear if 5 minutes of testing, even after coverage prioritization, will be a strong regression, but a stable, more efficient "snapshot" of good random tests for critical infrastructure compilers will be a valuable contribution to GCC and LLVM's already high-quality test suites.

A. Threats to Validity

First, we caution that cause reduction by coverage is intended to be used on the highly redundant, inefficient tests produced by aggressive random testing. While random testing is sometimes highly effective for finding subtle flaws in software systems, and essential to security-testing, by its nature it produces test cases open to extreme reduction. It is likely that human-produced test cases (or test cases from directed testing that aims to produce short tests) would be not reduce well enough to make the effort worthwhile. The quick test problem is formulated specifically for random testing, though we suspect that the same arguments also hold for model checking traces produced by SAT or depth-first-search, which also tend to be long and redundant. The primary threat to validity is that experimental results are based on one large case study on a large code base over time, one mutation analysis of a smaller but also important and widely used program, and a few indicative tests on a very large system, the GCC compiler.

VII. Related Work

This paper follows previous work on delta debugging [3], [4], [33] and other methods for reducing failing test cases. While previous work has attempted to generalize the circumstances to which delta debugging can be applied [34], [35], [36], this paper replaces preserving failure with any chosen effect. Surveying the full scope of work on failure reduction in both testing [32], [37] and model checking [38], [39] is beyond the scope of this paper. The most relevant work considers delta debugging in random testing [26], [1], [2], [40], which tends to produce complex, essentially unreadable, failing test cases [26]. Random test cases are also highly redundant, and the typical reduction for random test cases in the literature ranges from 75% to well over an order of magnitude [26], [40], [1], [32], [13]. Reducing highly-redundant test cases to enable debugging is an essential enough component of random testing that some form of automated reduction seems to have been applied even before the publication of the *ddmin* algorithm, e.g. in McKeeman's early work [41], and reduction for compiler testing is an active research area [32]. Recent work has shown that reduction has other uses: Chen et. al showed that reduction was required for using machine learning to rank failing test cases to help users sort out different underlying faults in a large set of failures [13].

Second, we propose an orthogonal approach to test suite minimization, selection and prioritization from that taken in previous work, which is covered at length in a survey by Yoo and Harman [14]. Namely, while other approaches have focused on minimization [42], [43], [44], [27], selection [15] and prioritization [28], [45], [46] at the granularity of entire test suites, this paper proposes reducing the size of the test cases composing the suite, a "finer-grained" approach that can be combined with previous approaches. Previous work on suite minimization has shown a tendency of minimization techniques to lose fault detection effectiveness [47]. While our experiments are not intended to directly compare cause reduction and suite-level techniques, we note that for SpiderMonkey, at the 30 second and 5 minute levels, fault detection was much better preserved by our approach than by prioritizations based on suite minimization techniques.

The idea of a quick test proposed here also follows on work considering not just the effectiveness of a test suite, but its *efficiency*: coverage/fault detection per unit time [10], [11]. Finally, as an alternative to minimizing or prioritizing a test suite, tests can be constructed with brevity as a criteria, as in evolutionary testing and bounded exhaustive testing [18], [19], [20], [21]. However, the applications where random testing is most used tend to be precisely those where "small by construction" methods have not been shown to be as successful, possibly for combinatorial reasons.

VIII. Conclusions and Future Work

This paper shows that generalizing the idea of delta debugging from an algorithm to reduce the size of failing test cases to an algorithm to reduce the size of test cases with respect to *any* interesting effect, which we call *cause reduction*, allows us to produce *quick tests*: highly efficient test suites based on inefficient randomly generated tests. Reducing a test case with respect to statement coverage not only (obviously) preserves statement and function coverage; it also approximately preserves branch coverage, test failure, fault detection, and mutation killing ability, for two realistic case studies (and a small number of test cases for a third subject, the GCC compiler). Combining cause reduction by statement coverage with test case prioritization by additional statement coverage produced, across 30 second and 5 minute test budgets and multiple versions of the SpiderMonkey JavaScript engine, an effective quick test, with better fault detection and coverage than performing new random tests or prioritizing a previously produced random test suite. The efficiency and effectiveness of reduced tests persists across versions of SpiderMonkey and GCC that are up to a year later in development time, a long period for such actively developed projects.

In future work we first propose to further investigate the best strategies for quick tests, across more subjects, to determine if the results in this paper generalize well. Second, it is clear from GCC that cause reduction by coverage is too expensive for some subjects, and the gain in efficiency is relatively small compared to the extraordinary computational demands of reduction. Two alternative mitigations come to mind: first, it is likely that reduction by even coarser coverages, such as function coverage, will result in much faster reduction (as more passes will reduce the test case) and better efficiency gains. Whether cause reduction based on coarse coverage will preserve other properties of interest is doubtful, but worth investigating, as statement coverage preserved other properties much more effectively than we would have guessed. Initial experiments with function coverage based reduction of SpiderMonkey tests showed good preservation of failure and fault detection, but we did not investigate how well preservation carried over to future versions of the software yet. A second mitigation for slow reduction (but not for limited efficiency gains) is to investigate changing *ddmin* to fit the case where expected degree of minimization is much smaller than for failures, and where the probabilities of being removable for contiguous portions of a test case are essentially independent, rather than typically related, which motivates the use of a binary search in *ddmin*.

We also propose other uses of cause reduction. While some applications are relatively similar to coverage-based minimization, e.g., reducing tests with respect to peak memory usage, security privileges, or other testing-based predicates, other possibilities arise. For example, reduction could be applied to a program itself, rather than a test. A set of tests (or even model checking runs) could be used as an effect, reducing the program with respect to its ability to satisfy all tests or specifications. If the program can be significantly reduced, it may suggest a weak test suite, abstraction, or specification. This approach goes beyond simply examining code coverage because examining code that is removed despite being covered by the tests/model checking runs can identify code that is truly under-specified, rather than just not executed (or dead code). Cause reduction can also be used to produce "more erroneous" code when considering "faults" with a quantitative nature. For example, C++ compilers often produce unreasonably lengthy error messages for invalid programs using templates, a problem well known enough and irritating enough to inspire a contest (http://tgceec.tumblr.com/) for the shortest program producing the longest error message. Using C-Reduce, we started with code from LLVM, with an effect designed to maximize error message length proportional to code length. C-Reduce eventually produced a small program:

```
struct x0 struct A<x0(x0(x0(x0(x0(x0(x0(x0(x0(x0(_T1,x0(_T1>
  <_T1*, x0(_T1*_T2>      binary_function<_T1*, _T2, x0{ }
```

This produces a very large error message on the latest `g++`, and the message doubles in size for each additional `(x0`.

References

[1] A. Groce, G. Holzmann, and R. Joshi, "Randomized differential testing as a prelude to formal verification," in *International Conference on Software Engineering*, 2007, pp. 621–631.

[2] A. Groce, K. Havelund, G. Holzmann, R. Joshi, and R.-G. Xu, "Establishing flight software reliability: Testing, model checking, constraint-solving, monitoring and learning," *Annals of Mathematics and Artificial Intelligence*, accepted for publication.

[3] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *Software Engineering, IEEE Transactions on*, vol. 28, no. 2, pp. 183–200, 2002.

[4] R. Hildebrandt and A. Zeller, "Simplifying failure-inducing input," in *International Symposium on Software Testing and Analysis*, 2000, pp. 135–145.

[5] K. Claessen and J. Hughes, "QuickCheck: a lightweight tool for random testing of haskell programs," in *ICFP*, 2000, pp. 268–279.

[6] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *International Conference on Software Engineering*, 2007, pp. 75–84.

[7] A. Groce, C. Zhang, E. Eide, Y. Chen, and J. Regehr, "Swarm testing," in *International Symposium on Software Testing and Analysis*, 2012, pp. 78–88.

[8] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011, pp. 283–294.

[9] J. H. Andrews, A. Groce, M. Weston, and R.-G. Xu, "Random test run length and effectiveness," in *Automated Software Engineering*, 2008, pp. 19–28.

[10] A. Gupta and P. Jalote, "An approach for experimentally evaluating effectiveness and efficiency of coverage criteria for software testing," *Journal of Software Tools for Technology Transfer*, vol. 10, no. 2, pp. 145–160, 2008.

[11] M. Harder, J. Mellen, and M. D. Ernst, "Improving test suites via operational abstraction," in *Software Engineering, 2003. Proceedings. 25th International Conference on*. IEEE, 2003, pp. 60–71.

[12] J. A. Jones and M. J. Harrold, "Empirical evaluation of the Tarantula automatic fault-localization technique," in *Automated Software Engineering*, 2005, pp. 273–282.

[13] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr, "Taming compiler fuzzers," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2013, pp. 197–208.

[14] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Softw. Test. Verif. Reliab.*, vol. 22, no. 2, pp. 67–120, Mar. 2012.

[15] J. Bible, G. Rothermel, and D. S. Rosenblum, "A comparative study of coarse- and fine-grained safe regression test-selection techniques," *ACM Trans. Softw. Eng. Methodol.*, vol. 10, no. 2, pp. 149–183, 2001.

[16] A. Groce, K. Havelund, and M. Smith, "From scripts to specifications: The evolution of a flight software testing effort," in *International Conference on Software Engineering*, 2010, pp. 129–138.

[17] Android Developers Blog, "UI/application exerciser monkey," http://developer.android.com/tools/help/monkey.html.

[18] L. Baresi, P. L. Lanzi, and M. Miraz, "Testful: an evolutionary test approach for Java," in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2010, pp. 185–194.

[19] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. ACM, 2011, pp. 416–419.

[20] J. H. Andrews, T. Menzies, and F. C. Li, "Genetic algorithms for randomized unit testing," *IEEE Transactions on Software Engineering (TSE)*, vol. 37, no. 1, pp. 80–94, 2011.

[21] J. Andrews, Y. R. Zhang, and A. Groce, "Comparing automated unit testing strategies," Department of Computer Science, University of Western Ontario, Tech. Rep. 736, December 2010.

[22] J. Ruderman, "Introducing jsfunfuzz," 2007, http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/.

[23] M. Gligoric, A. Groce, C. Zhang, R. Sharma, A. Alipour, and D. Marinov, "Comparing non-adequate test suites using coverage criteria," in *International Symposium on Software Testing and Analysis*, 2013, pp. 302–313.

[24] A. Groce, A. Fern, J. Pinto, T. Bauer, A. Alipour, M. Erwig, and C. Lopez, "Lightweight automated testing with adaptation-based programming," in *IEEE International Symposium on Software Reliability Engineering*, 2012, pp. 161–170.

[25] J. Ruderman, "Mozilla bug 349611," https://bugzilla.mozilla.org/show_bug.cgi?id=349611 (A meta-bug containing all bugs found using jsfunfuzz.).

[26] Y. Lei and J. H. Andrews, "Minimization of randomized unit test cases," in *International Symposium on Software Reliability Engineering*, 2005, pp. 267–276.

[27] T. Y. Chen and M. F. Lau, "Dividing strategies for the optimization of a test suite," *Inf. Process. Lett.*, vol. 60, no. 3, pp. 135–141, 1996.

[28] G. Rothermel, R. Untch, C. Chu, and M. Harrold, "Prioritizing test cases for regression testing," *Software Engineering, IEEE Transactions on*, vol. 27, no. 10, pp. 929–948, 2001.

[29] "Yaffs: A flash file system for embedded use," http://www.yaffs.net/.

[30] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *International Conference on Software Engineering*, 2005, pp. 402–411.

[31] L. Zhang, S.-S. Hou, J.-J. Hu, T. Xie, and H. Mei, "Is operator-based mutant selection superior to random mutant selection?" in *International Conference on Software Engineering*, 2010, pp. 435–444.

[32] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for C compiler bugs," in *Conference on Programming Language Design and Implementation*, 2012, pp. 335–346.

[33] A. Zeller, "Yesterday, my program worked. today, it does not. why?" in *ESEC / SIGSOFT Foundations of Software Engineering*, 1999, pp. 253–267.

[34] J. Choi and A. Zeller, "Isolating failure-inducing thread schedules," in *International Symposium on Software Testing and Analysis*, 2002, pp. 210–220.

[35] H. Cleve and A. Zeller, "Locating causes of program failures," in *ICSE*, G.-C. Roman, W. G. Griswold, and B. Nuseibeh, Eds. ACM, 2005, pp. 342–351.

[36] A. Zeller, "Isolating cause-effect chains from computer programs," in *Foundations of Software Engineering*, 2002, pp. 1–10.

[37] G. Misherghi and Z. Su, "Hdd: hierarchical delta debugging," in *International Conference on Software engineering*, 2006, pp. 142–151.

[38] P. Gastin, P. Moro, and M. Zeitoun, "Minimization of counterexamples in SPIN," in *In SPIN Workshop on Model Checking of Software*. Springer-Verlag, 2004, pp. 92–108.

[39] A. Groce and D. Kroening, "Making the most of BMC counterexamples," *Electron. Notes Theor. Comput. Sci.*, vol. 119, no. 2, pp. 67–81, Mar. 2005.

[40] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer, "Efficient unit test case minimization," in *International Conference on Automated Software Engineering*, 2007, pp. 417–420.

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

[42] S. McMaster and A. M. Memon, "Call-stack coverage for GUI test suite reduction," *Software Engineering, IEEE Transactions on*, vol. 34, no. 1, pp. 99–115, 2008.

[43] A. J. Offutt, J. Pan, and J. M. Voas, "Procedures for reducing the size of coverage-based test sets," in *In Proc. Twelfth Int'l. Conf. Testing Computer Softw*, 1995.

[44] H.-Y. Hsu and A. Orso, "Mints: A general framework and tool for supporting test-suite minimization," in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*. IEEE, 2009, pp. 419–429.

[45] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, "Timeaware test suite prioritization," in *Proceedings of the 2006 international symposium on Software testing and analysis*, ser. ISSTA '06. New York, NY, USA: ACM, 2006, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/1146238.1146240

[46] S. G. Elbaum, G. Rothermel, S. Kanduri, and A. G. Malishevsky, "Selecting a cost-effective test case prioritization technique," *Software Quality Journal*, vol. 12, no. 3, pp. 185–210, 2004.

[47] G. Rothermel, M. J. Harrold, J. von Ronne, and C. Hong, "Empirical studies of test-suite reduction," *Journal of Software Testing, Verification, and Reliability*, vol. 12, pp. 219–249, 2007.