Falsification-Driven Verification

Alex Groce, Iftekhar Ahmed, Carlos Jensen
School of Electrical Engineering and Computer Science
Oregon State University, Corvallis, Oregon
Email: agroce@gmail.com, ahmedi@onid.oregonstate.edu, cjensen@eecs.orst.edu

Paul E. McKenney
IBM Linux Technology Center
Email: paulmck@linux.vnet.ibm.com

***Abstract*—Formal verification has advanced to the point that developers can verify the correctness of small, critical modules. Unfortunately, despite considerable efforts, determining if a "verification" verifies what the author intends is still difficult. Previous approaches are difficult to understand and often limited in applicability. Developers need verification coverage in terms of the software they are verifying, not model checking diagnostics. We propose a methodology to allow developers to determine (and correct) what it is that they have verified, and tools to support that methodology. Our basic approach is based on a novel variation of mutation analysis and the idea of verification driven by falsification. We use the model checker CBMC and show that this approach is applicable not only to simple data structures and sorting routines, and verification of a routine in Mozilla's JavaScript engine, but to understanding an ongoing effort to verify the Linux kernel Read-Copy-Update mechanism.**

I. Introduction

Software model checking [1] has recently, thanks to improvements in model checking tools as well as SAT and SMT solvers, and the large amount of memory available even on commodity workstations, become a potentially valuable tool for developers of critical software modules who want to, at minimum, perform a very aggressive search for bugs and, at best, prove correctness of their code. Tools such as CBMC [2] (the C Bounded Model Checker) allow a software engineer to model check code by writing what is essentially a generalized test harness[1] in the language of the Software Under Test (SUT). Figure 1 shows an example CBMC harness for sorting routines. Only a few aspects differ from normal testing. First, `nondet_int` in CBMC can return *any value*. It is not equivalent to a "random" choice but true nondeterminism: CBMC will explore all values of the type. The `__CPROVER_assume` statement is used to restrict program executions: it has the usual `assume` semantics [3], [4], so CBMC ignores all executions that violate assumptions.

CBMC compiles a harness and the SUT (here a quick sort implementation) into a goto-program, instruments this program with property checks for assertions, array bounds violations, etc., and then unrolls loops based on a user-provided *unwinding bound* to produce a SAT problem or SMT constraint such that satisfying assignments are representations of a trace demonstrating a property violation, known as a *counterexample* [5]. For CBMC, this means that if *any possible execution allowed by the harness* violates any properties checked, a counterexample will be produced. This includes user-specified assertions and automatically generated properties such as array bounds and pointer validity checks. One generated property is that no loop in the program executes more than the *unwinding bound* times. For example, if we run CBMC on the harness shown and set the unwinding bound to 3 and add `-DSIZE=2`, we will check the correctness of the SUT over *all possible arrays* of size 2 or less, including checking that sorting never requires passing through any loop more than 3 times.

[1]By a harness we mean a program that defines an environment and the form of valid tests, and provides correctness properties.

```
#include "sort.h"
int a[SIZE];
int ref[SIZE];
int nondet_int();
int main () {
  int i, v, prev;
  int s = nondet_int();
  __CPROVER_assume((s > 0) && (s <=SIZE));
  for (i = 0; i < s; i++) {
    v = nondet_int();
    printf ("LOG: ref[%d] = %d\n", i, v);
    ref[i] = v; a[i] = v;
  }
  sort(a, s);
  prev = a[0];
  for (i = 0; i < s; i++) {
    printf ("LOG: a[%d] = %d\n", i, a[i]);
    assert (a[i] >= prev);
    prev = a[i];
  }
}
```

Fig. 1. CBMC harness to check a sorting routine.

When a model checker produces a counterexample, a developer's task is straightforward, if sometimes difficult: either the SUT has a fault, or the harness itself is flawed. In both cases, the output of the verification effort is the counterxample trace, which is full of evidence as to the reason for the failure to verify the SUT. Moreover, any solution (fix to SUT or harness) is easily checked: if it is correct, the model checker stops reporting the previous counterexample. This is essentially a normal debugging problem, but with the advantage that solutions are easily checked.

Unfortunately, model checkers do not invariably report counterexamples: eventually the SUT is likely to satisfy the properties encoded in the harness! It is in this case that problems arise: what, precisely, has been verified? Does the harness in fact specify all aspects of correctness required? Is the SUT correct? Formal verification is not only subject to the many issues that make "no faults detected" results dubious in testing [6], [7], but also to more subtle problems. For example, an incorrect *assume* statement may constrain a system so that not only are there no counterexamples, there are no (interesting) executions of the system at all.

This problem has concerned the model checking community for some time [8], [9], and resulted in efforts to define

coverage metrics for model checking. While such metrics are interesting and useful, they have typically been aimed at hardware verification, and most useful to experts in formal verification. In this paper, we adapt traditional mutation testing [10], [11] to the problem of software verification. A *mutant* of a program is a version of the program that introduces a small syntactic change. The idea behind mutation testing is that a good test suite will be able to detect when (as is usually the case) such a change introduces a bug in the SUT. In the case of bounded model checking, since we aim at *verification* rather than merely good testing, it seems clear that surviving mutants are likely to indicate a weakness of the verification.

The use of mutation testing most often seen in the software engineering literature is simply noting a mutation kill rate. This is not enough for verification. The typically small scope of the code to be verified, and the presumed importance of code targeted for verification suggests an approach in which *individual mutants* are examined by the developer. Without additional assistance, such an approach cannot scale. We propose that the capabilities of the model checking tool, the nature of formal verification, and the adoption of certain best practices can make this seemingly too-demanding approach in fact practical for real verification tasks.

The contribution of this paper is a *falsification-driven* verification methodology that uses mutants to aid the user of a model checker in understanding "successful" verification results, determining when a harness is flawed, and correcting harness. To support this approach, we show how to use mutation testing to choose a problem size in bounded model checking, how to mutate a harness to determine if any similar harnesses have an equal (or better) mutation kill rate, and most importantly, how to modify CBMC, a harness, and mutants to automatically produce *successful high-coverage executions covering mutated code* in order to understand mutant behavior and find subtle harness flaws. This approach, unlike a simpler method of searching for cases where the mutated code and original code behave differently for the same inputs, applies even to verification of reactive and concurrent systems, where there is no easily obtained notion of "for the same inputs." At a more general level, we discuss the fundamental nature of "verification" in a real-world context where specifications are never known to be complete. We propose that falsification, as in Popper's philosophy of science [12], is a useful conceptual framework for verification efforts: rather than focusing on what can be proven about a program, it may be best to focus on how a verification distinguishes the "real" program from similar alternative programs that do *not* match the theory of program behavior. This approach aims at verification, but continually evaluates and refines a verification effort by its ability to *falsify* rather than to verify.

II. A Simple Example Verification

As an example of the proposed verification methodology, consider again the harness shown in Figure 1. If we take the first hit on Google for "quick sort in C" [13], shown in Figure 2[2], we can model check it using the harness, defining `SIZE=2` and setting the unwinding bound to 3 (we need one more unwinding than the largest possible number of items in the array). CBMC reports `VERIFICATION SUCCESSFUL` in less than a second. Does this mean we have verified what we want to verify?

```
#include "sort.h"
void quickSort( int a[], int l, int r)
{
  printf ("LOG: called with l=%d, r=%d\n", l, r);
  int j;
9 if( l < r )
    {
      // divide and conquer
      j = partition( a, l, r);
      quickSort( a, l, j-1);
      quickSort( a, j+1, r);
    }
}
int partition( int a[], int l, int r) {
  int pivot, i, j, t;
  pivot = a[l];
  i = l; j = r+1;
26 while( 1)
    {
28    do ++i; while( i <= r && a[i] <= pivot );
      do --j; while( a[j] > pivot );
30    if( i >= j ) break;
31    t = a[i]; a[i] = a[j]; a[j] = t;
    }
  t = a[l]; a[l] = a[j]; a[j] = t;
  return j;
}
void sort(int a[], unsigned int size) {
  quickSort(a, 0, size-1);
}
```

Fig. 2. Quick sort code.

A. Finding a Good Problem Size

The first question we face is whether 2 is really a good maximum array size to examine. The problem of determining a *completeness threshold* (an execution-length bound sufficient to prove correctness in all cases for a given property) for bounded model checking is fundamentally difficult [14] and is, for real-world C programs, more an art than a science at present[3]. Are there bugs for which 2 is too small an array size? In order to find out, we generate a set of mutants for `quicksort.c`. Using the mutation tool for C code developed by Jamie Andrews [15], we can produce 81 mutants of this code in less than a second. We then run the harness with unwinding bound 2 (and `SIZE=1`) on each of the 81 mutants. The process takes less than a minute and a half (on a Macbook Pro with dual-core 3.1GHz Intel Core i7, using only one core). CBMC reports that 6 mutants do not compile (these remove variable declarations, for the most part), 4 are detected by the harness (a counterexample is produced: we say the mutant is *killed*), and 71 mutants pass without detection (the verification is successful, in which case we say the mutant *survives*). Clearly length 1 arrays are not sufficient to detect even the most glaring bugs in a sort algorithm (no surprise; all size 1 arrays are sorted). What about our choice of size 2? Re-running the mutants (dropping those already killed by the smaller bound) takes slightly over 13 minutes (due to one mutant requiring over 8 minutes to model check) and reduces the number of surviving mutants to 26. We could inspect these mutants by hand, but it seems highly unlikely that a *complete*

[2]In fact, that actual code is incorrect, with an access `a[i]` that does not properly use short circuiting logical operators to protect array bounds; CBMC detected this, and we fixed it for this paper.

[3]In our own practice, the most common way of setting it is to guess a bound and see if the resulting problem is too large for the available resources.

```
9 :   /*(rep_op)*/ if (l <= r)
26 :  /*(rep_const)*/ while(-1)
26 :  /*(rep_const)*/ while( ((1)+1))
28 :  /*(rep_op)*/ do ++i; while(i<r && a[i]<=pivot);
28 :  /*(rep_op)*/ do ++i; while(i!=r && a[i]<=pivot);
28 :  /*(rep_op)*/ do ++i; while(i<=r && a[i]<pivot);
30 :  /*(rep_op)*/ if( i > j ) break;
31 :  /*(del_stmt)*/ t=a[i]; /*  a[i]=a[j]; */  a[j]=t;
```

Fig. 3. Surviving mutants at `SIZE=3`.

verification over all possible arrays with a good specification for sorting would produce such a poor mutation kill rate. If we up the size limit to 3 (the verification taking just over 33 minutes), only 8 mutants survive. Note that each problem, due to the harness' assignment of `s` to any size smaller than the current size, include all smaller problem sizes. This makes the behavior of the verification problem size setting match that of CBMC where an unwinding bound is a maximum, rather than a fixed size. We assume in all cases below[4] that harnesses are constructed so that increasing problem size includes all smaller problems as a best practice.

At this point, we can increase `SIZE` to 4 (which will kill one additional mutant), but the time required to check the remaining mutants is growing rapidly. In fact, completing the check for size 4, even though only the original program and 8 mutants have to be checked, requires *nearly 9 hours*. When the model checking difficulty grows more slowly with problem size, we propose increasing size until the number of mutants killed does not increase with a step up in size (we call such a size *mutant-stable*). However, in many cases, such as this one, the time required to check mutants starts growing unacceptably. We propose a more efficient algorithm for finding a mutant-stable size below (Figure 7), and mutations can be checked in parallel, but the fundamental problem for size 4 (and above) is that some individual mutants require hours to model check. What is a developer to do?

B. Examining Surviving Mutants

The developer should turn to the surviving mutants. The 8 surviving mutants for size 3 are shown in Figure 3. The comment indicates the type of mutant, and the line # in the quick sort file is also given for reference. The relevant lines are marked in Figure 2. Some of these mutants are easily seen to be equivalent to the original code. For example, the two `rep_const` mutations simply change a `while(1)` into an equivalent infinite loop with a different constant non-zero value. These two mutants could in fact have been automatically removed from the set, like uncompilable mutants, by checking their compiled code for equivalence with the original program [16]. We suggest always pruning mutants via Trivial Compiler Equivalence (TCE). The remaining 6 mutants produce different binaries when compiled with an optimizing compiler, so require manual analysis. The 5 `rep_op` mutations all alter comparison operators by changing their value on one corner case, and we may suspect that quick sort is robust to, for example, changing `i <= r` to `i != r` since `i` is initially set to `l`, which we know to be less than `r`.

The `del_stmt` mutant, however, is clearly problematic. How can quick sort be correct if the inner loop's swapping of `a[i]` and `a[j]` is changed to instead copy `a[i]` to `a[j]`? The consequences of this mutant are clearly drastic, but why are they not detected by our harness? We find out by asking CBMC to produce an execution such that 1) the mutated code is covered 2) other coverage is maximized (to avoid degenerate executions, e.g., over size 1 arrays) and 3) the execution is *not a counterexample*. We have modified CBMC, and written instrumentation tools that produce a modified mutant and harness, allowing us to pose such queries (see Section III. Running CBMC in this mode, with the target of maximum branch coverage and statement coverage of the `del_stmt` mutant (actually the statement after it, since it no longer exists), we produce the witness in Figure 4 in less than a minute[5]. Our harness checks that the array `a` is *sorted* after the call to `sort`, but it does not check that it is a permutation of the input!

We might have discovered this problem by a different method: if we remove the call to `sort` in the harness, and replace it by a loop assigning `nondet_int` to every element in array `a` (a kind of most-general any-order type-correct "mutant" of `sort`), we can run the modified CBMC and see examples of executions our harness allows, which should include *any sorted* array. The problem with this method is that, while it sometimes works, CBMC is also free to set all elements in all arrays to 0, and to generally provide an uninformative example of a successful execution. The requirement to cover a mutant (and as much other code as possible) helps guide CBMC to a successful execution that is *likely to be incorrect*, because a non-equivalent mutant *changes the original program's behavior*. Moreover, while the problem with the harness in this case is simple, understanding arbitrary "passing" but wrong executions can be very difficult without

```
LOG: ref[0] = 2147414872
LOG: ref[1] = 2147480408
LOG: ref[2] = -1073743560
LOG: called with l=0, r=2
LOG: called with l=0, r=-1
LOG: called with l=1, r=2
LOG: called with l=1, r=1
LOG: called with l=3, r=2
LOG: a[0] = 2147414872
LOG: a[1] = 2147480408
LOG: a[2] = 2147480408
```

Fig. 4. Witness to the harness' inability to kill the `del_stmt` mutant.

```
...
   int i, v, count, qcount, prev;
...
  sort(a, s);
  // Pick a value to check
  v = nondet_int();
  count = 0;
  qcount = 0;
...
  for (i = 0; i < s; i++) {
...
     if (ref[i] == v)
       count++;
     if (a[i] == v)
       qcount++;
...
   assert (count == qcount);
}
```

Fig. 5. Modifying the harness to ensure `a` is a permutation of `ref`.

[4]There is one noted exception in Section IV-D.

[5]We show the output of the print statements, not the full CBMC trace, in the interest of space and ease of understanding; examining this output alone, initially, is the most likely course of action for a developer as well.

the ability to think about *a specific bug* the model checker is missing. Moreover, basing the production of witnesses on mutants allows us to compare harnesses even over killed mutants: if one harness reduces the set of passing executions for a mutant, it is arguably a better verification of correctness than one allowing more executions of the mutant, even if both produce a counterexample killing the mutant. Unlike traditional mutation analysis, we can take the question "how killed is this mutant?" seriously because we aim at exhaustive testing. A harness is most effective with respect to a mutant if it allows *no* executions covering the mutant to pass, or, to be precise, only those such that the relevant behavior is still identical to the correct program.

The witness tells us that the sorting harness is too *weak*. We say that a harness is weak if it fails to detect incorrect executions. One harness is stronger than another if it detects more failures; we can indirectly estimate strength by determining how many mutants a harness can kill at a given problem size, and how executions covering killed mutants can still satisfy the harness. Figure 5 shows how to modify the sorting harness to check for permutations [6]. Because CBMC is exhaustive, instead of performing a complete check for permutation, we can detect violation of the property by letting `v` be any value, and ensuring that both `a` and `ref` contain the same *number* of elements equal to `v`. If `a` is *not* a permutation of `ref`, $\exists$`v` such that this is not true, and we can rely on CBMC to report it as part of a counterexample. While a CBMC harness resembles a program to test the SUT, it can make use of unusual specifications relying on exhaustiveness.

If we modify the harness as shown, we can re-check our mutants (including those TCE would remove). With the revised harness, checking mutants at `SIZE=1` takes slightly longer (8 more seconds) and kills the same number of mutants, since the problem is the size, not the harness. At `SIZE=2` mutant kill results are again unchanged, but now completes in about 5 minutes. Finally, at `SIZE=3`, we kill the `del_stmt` mutant that previously survived, after only 14 minutes, not much longer than at `SIZE` 2 with the weaker harness. The `SIZE` 3 verification is stable. Checking stability by running `SIZE` 4 now only requires slightly more than an hour, nearly an order of magnitude faster than before. A stronger harness not only finds more bugs in the SUT, it often *speeds up verification*.

As briefly mentioned in the introduction, it is also possible to understand a mutant by modifying the harness to call both the mutated code and the original code on the same inputs, and search for witnesses where 1) the execution is passing but 2) the return value(s) for the mutated code differ(s) from the original code. However, this increases the complexity of the model checking problem (checking equivalence of two functions is often much harder than simply specifying valid executions) and does not easily apply to any verifications other than simple function calls. For example, forcing the same interleavings in threaded programs, or detecting all differences in state-modification for more reactive code is often infeasible or requires significant human intervention. While we do apply differential checks in some cases below, we do not propose this as a core technique suitable for general-purpose falsification-driven verification.

[6]In fact, if we choose a `val` to check before we assign to `ref` and count that value's appearances, we can completely dispense with storing `ref` at all. In practice, this is more difficult for non-CBMC experts to understand, and makes model checking only slightly faster.

C. Mutating the Harness

Previous efforts to understand model checking results have also considered mutants to the property, usually given as a temporal logic formula [17]. Once a developer is satisfied with a harness, has a mutant-stable bound for verification, and is convinced all surviving mutants are semantically equivalent to the original program (or, if not equivalent, also satisfy the same correct specification), we propose the developer *mutate the test harness itself*. The idea is to check that 1) most mutants of the harness reject the SUT and 2) the remaining mutants have a mutant kill rate no greater than that of the harness. For the fixed sort harness, there are 61 mutants, of which 2 do not compile. Of these, 40 produce an incorrect counterexample for the original, correct, quick sort. An additional 10 have mutant kill rates worse than the original harness (from as low as 5% of mutants killed to only a few percent worse than the fixed harness). The remaining 9 mutants have the same ability to kill mutants as the original harness. Most of these involve modifying a relational operator in a loop or an assumption in a way that does not change the verification problem. The only interesting surviving harness mutant is one that removes the assignment of a fresh non-deterministic value to `v` after the call to `sort`. This means the check for permutation difference will always be performed on the last element of `ref`. On reflection, it seems plausible that this is sufficient to produce a counterexample for all the quick sort mutants, but it is clearly not an improvement to the harness, in terms of either verification strength or clarity.

In addition to showing the current harness is at least a "local minima" with respect to mutants, mutation analysis of the harness also provides some evidence of our technique's ability to detect subtle specification and environment flaws. In particular, it shows the value of *inspecting all surviving mutants*. One mutant modifies the assumption on `s` to be `< SIZE` rather than `<= SIZE`, which is the same as lowering the `SIZE` by one; this is a fairly easy mistake to make in a harness, and likely not much more uncommon than off-by-one bugs in actual code. This reduces the effectiveness of the verification by 19 mutants, so is likely not to escape notice, and would also (in our framework) simply result in a higher size being chosen as mutant-stable. Deleting the assignment `prev = a[i]`, however, only kills 4 fewer mutants than the original harness. Traditional coverage and some model checking coverages cannot detect this problem: because of the assignment to `prev` outside the loop, the variable *is* used in the specification, and in fact used to detect many faults (it eliminates any mutants that can cause `a[0]` to not be the least element). The harness "covers" all behavior of quick sort in general, since the permutation requirement remains in place. However, it cannot detect versions of quick sort that 1) preserve permutation and 2) make the first element correct, but 3) don't always sort the array correctly. In particular, the call to `quickSort` with `j + 1` can be removed or modified to `j + 2`. Examining the deleted/removed recursive calls shows the user of the problem in this case. Our modified CBMC can, as for our first harness bug, produce a witness showing a permutated array with correct `a[0]` but with out-of-order later elements.

III. Algorithms and Techniques

Falsification-driven verification is a semi-automated approach that relies heavily on algorithmic and tool support. While the typically smaller scope of code targeted for verification (vs. testing) makes the work easier, it is not likely to be feasible without automation of many subtasks. Existing tools make producing a set of mutants and checking them using a harness relatively easy, but other tasks require new algorithms and tools. Figure 6 shows the basic flow, which is directed not by a fixed algorithm but by the intelligence and experience [18] of the developer. Novel tools or techniques are on the right side of the diagram (mutation analysis itself is not novel, but our tool for integrating this with the model checker and recording results for use by other parts of the toolchain is non-standard), other than the model checker itself. We show only top-level functionality that a developer uses, not supporting algorithms for instrumentation of SUT and harness (mutants). Also omitted are techniques sometimes useful, easily implemented using these tools, such as pure-nondeterminstic "most-general mutants" and reference checking to find an input where a function and its mutant differ on the same input.

One important need is a version of CBMC capable of applying built-in assertions checks (e.g, bounds checks, pointer dereference, divsion by zero) as *assumptions* rather than assertions. Converting assertions to assumptions is the means by which we produce *passing* executions showing how a harness can fail to detect a mutant. For developer-provided harnesses, this can be performed by automatic source-to-source transformation of the harness, but CBMC's internal constraints have to be handled inside the model checker. We implemented a new CBMC command-line option, `--find-success` that provides this functionality. In all algorithms, `check` means running CBMC as usual, with any needed automatic property checks, while `scheck` indicates running CBMC with `find-success` enabled. In Figure 6 we assume the use of a modified version of CBMC.

Figure 7-11 show core algorithms (implemented as prototype tools in our framework). Two of these are support algorithms for instrumentation, used in the production of witnesses for understanding surviving mutants (Figures 8 and 9), while the other three (Figures 7, 10, and 11) face developers. The uses of these algorithms are described at a high level in the introductory example (in Sections II-A, II-B, and II-C, respectively), and in the case studies below. In the actual implementations, these tools perform considerable record-keeping and logging, communicating through the file system. For example, harness analysis records killing counterexamples and set of survivors (as well as executions times and other statistics) for each run. There are also convenient scripts in the toolchain, for instance a tool to automatically call `maxcover` on all (surviving) mutants. Running `maxcover` on *all* mutants provides a measure of harness strength that is more fine-grained than a simple kill rate: harnesses can be compared by the maximum coverage of *all* mutants, even if they have the same kill rate. If one harness produces executions with lower coverage (or no executions at all) for some killed mutants, it is stronger. For some mutants, *any* passing executions show a harness flaw. While not polished enough for release, these tools (implemented as Python scripts) have proven robust enough for our internal use and are available for examination online.

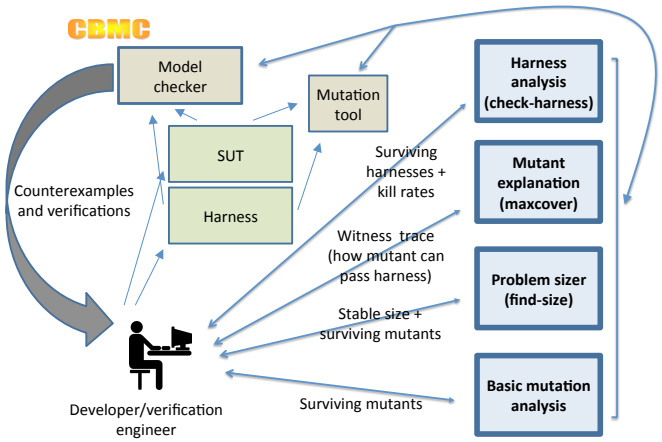

Fig. 6. Basic flow of falsification-driven verification.

```
(int, survivors) find-size(H, M, S0: int,
                           O: int → options,
                           U: int → int)

S = S0-1
r' = {}
changed = False
while changed:
   TOP:
   S = S + 1
   changed = False
   r = r'
   r' = {}
   for m ∈ M:
      if m ∉ r:
         r[m] = check(H,m,U(S),O(S))
         if r[m] == VERIFICATION FAILED:
            //once killed, assume always killed
            M = M \ m
      if r[m] == VERIFICATION SUCCESSFUL:
         r'[m] = check(H,m,U(S+1),O(S+1))
         if r'[m] == VERIFICATION FAILED:
            M = M \ m
            changed = True
            goto TOP
// No result changed, so S is mutant-stable
return (S-1, M)
```

Fig. 7. Algorithm 1: Finding size/unwinding bound and surviving mutants.

```
harness covering(H, TARGET)

H' = H
for stmt ∈ H':
   if stmt == assert(P):
      stmt = assume(P);
cover = [
  assume(total_coverage >= TARGET);
  assert(!mutant_covered);
]
insert cover at end of H'.main()
return H'
```

Fig. 8. Algorithm 2: Convert harness into maximal coverage search.

```
mutant coverinst(m)

n = 0
m' = m
for if_stmt c in m':
   if c has no else:
      add [else {}] to c
for basic_block b in m':
   b = [if !covered[n] {
           covered[n] = 1;
           total_covered += 1;
        }
        b]
   n = n + 1
for stmt s in m':
   if MUTANT(s):
      s = [{mutant_covered = 1;
            s}]
m' = [int total_covered = 0;
      int mutant_covered = 0;
      int covered[n];
      m']
return m'
```

Fig. 9. Algorithm 3: Instrument mutant for coverage.

```
trace maxcover(m, H, S, O, U)

m' = coverinst(m)
T = 0
trace = ∅
failed = False
while (not failed)
   H' = covering(H, T)
   r = scheck(H,m',U(S),O(S))
   if r == VERIFICATION SUCCESSFUL:
     // Can't achieve this level of coverage
     failed = True
   else if r == VERIFICATION FAILED:
     trace = r.trace
     // Instead of incrementing by 1, see
     // what actual coverage was and
     // increase by one from that baseline.
     T = trace.read(total_covered) + 1
return trace
```

Fig. 10. Algorithm 4: Find a maximally covering execution trace that covers a mutant.

```
report check-harness(SUT, M, H, M(H), S, O, U)

K_H = killed(M, H, S)
Hkills = ∅
Hequal = ∅
Hbetter = ∅
for H_i in M(H):
   original = check(H_i, SUT, U(S), O(S))
   // ``kills'' the SUT -- an invalid harness
   if original == VERIFICATION FAILED:
      Hkills += H_i
   else: // check if this kills fewer mutants
      K_Hi = killed(M, H, S)
      if |K_Hi| > |K_H|:
         Hbetter += H_i
      if |K_Hi| == |K_H|:
         Hequal += H_i
      else:
         Hkills += H_i
return (Hkills, Hequal, Hbetter)
```

Fig. 11. Algorithm 5: Analyze a harness.

IV. Case Studies and Experimental Results

A. Algorithm Implementations

Our initial experiments involved relatively small verification problems, based on implementations taken from the web or student code for popular algorithms and data structures.

1) Binary Search: The ideas in this paper grew out of a side project of the first author: to write a follow-up to Jon Bentley's article on verifying binary search [19] in the context of modern software verification tools (and Joshua Bloch's discovery of a bug in the assumptions behind Bentley's proof [20]). The modeling required is moderately complex (to scale well, an abstract "sorted array" that represents all sorted arrays but only introduces variables equal to the number of probes made by the search is essential). In this case, we did not produce an initial, weaker version of the harness, but checked the existing harness using mutants, and determined that all 3 surviving mutants are equivalent to a true binary search.

Checking harness mutants (which took 37 minutes, including computing the kills for the original harness) produced results confirming the belief this is a good harness. Of the 31 compiling harness mutants, 19 failed to verify the correct binary search, and 7 had worse kill rates than the original harness. The remaining 5 harnesses, all with equal kill rates of 86.7%, all modify an assumption to allow the harness to also check size 0 arrays. This doesn't kill any additional mutants, but is harmless as expected. Of the harness mutants with worse kill rates, three are mutants of the assumptions on the nondeterministic value used to make sure that if binary search returns -1, no index in the array actually contains the searched-for item. Two of these mutants are off-by-one-errors that exclude item 0 from the check, an easy-to-make mistake. Both of these fail to kill *exactly* one mutant killed by the correct harness: the mutant that sets the lower bound initially to 1 instead of 0. Traces of passing runs for these mutants show the underlying problem easily (the sought item is at location 0).

2) Doubly-linked-list Insertion Sort: Another example, making use of recursive data structures and pointer validity checks, is a simple piece of code for inserting an item (in sorted order) into a doubly-linked list [21]. Our initial harness omitted a check for correctness of `prev` pointers. This problem didn't directly prevent mutants from being detected, but pushed the stable size larger, as with the quick sort example above. Looking at a trace of a size 3 run that fails to kill a clearly problematic mutant easily reveals the problem (the results are correct up to `prev` pointers). This example also showed another use of mutants, in that some seemingly problematic surviving mutants actually just showed a pointless redundancy in the implementation, enabling the removal of an entire conditional branch. A harness check (requiring 30 minutes, including computing the mutant kills for the original harness) showed that of the 105 compiling harness mutants, 92 fail to verify the original code. Another 2 have a worse kill rate than the original (which kills 81% of mutants, a low rate due to the code redundancy), and 11 survive. The large number of survivors is due to a redundancy of the final harness, which checks sortedness and the permutation property for both a forward `next` traversal of the list and a `prev` traversal. Omitting any *one* of these (e.g. `prev` sortedness or `next`

```
jsint
js_BoyerMooreHorspool(const jschar *text, jsint textlen,
                      const jschar *pat, jsint patlen,
                      jsint start)
{
  jsint i, j, k, m;
  uint8 skip[BMH_CHARSET_SIZE];
  jschar c;
  JS_ASSERT(0 < patlen && patlen <= BMH_PATLEN_MAX);
  for (i = 0; i < BMH_CHARSET_SIZE; i++)
    skip[i] = (uint8)patlen;
  m = patlen - 1;
  for (i = 0; i < m; i++) {
    c = pat[i];
    if (c >= BMH_CHARSET_SIZE)
      return BMH_BAD_PATTERN;
    skip[c] = (uint8)(m - i);
  }
  for (k = start + m;
       k < textlen;
       k += ((c = text[k]) >= BMH_CHARSET_SIZE) ?
            patlen : skip[c]) {
    for (i = k, j = m; ; i--, j--) {
      if (j < 0)
        return i + 1;
      if (text[i] != pat[j])
        break;
    }
  }
  return -1;
}
```

Fig. 12. SpiderMonkey 1.6 Boyer-Moore-Horspool code.

permutation) the harness can still detect all mutants. Removing two, however, fails to kill mutants. The two harness mutants with worse kill rates have such poor kill rates ($< 50\%$ and $< 25\%$ respectively) that they do not show possible erroneous versions of the harness.

3) Dijkstra's Algorithm: A student in our model checking seminar proposed a verification of an implementation of Dijkstra's shortest path algorithm.

4) AVL Tree:

B. SpiderMonkey Boyer-Moore-Horspool Implementation

Figures 12 and 13 show, respectively, the source code and an initial harness for verification of the Boyer-Moore-Horspool substring finding algorithm [22], [23] in version 1.6 of Mozilla's SpiderMonkey JavaScript engine. Verifying this code presents one immediate issue that is not unusual in verification: how to handle an `assert` in the code being verified. An `assert` at the end of a function or in the main body is typically an additional part of the specification, and is often best left unchanged. An `assert` at the beginning of a function's body, however, is typically a preconditon for the code, indicating that when it is violated the code will not behave correctly. It is natural to consider changing such an assertion into an `assume` and ignoring any problems produced by calling the code with non-conforming inputs. While this can be a useful technique (for instance when it is hard to write a harness that only produces valid inputs, but easy to filter out the invalid inputs and only verify behavior for those) it is also a dangerous technique. Mutation analysis of the harness shows that 4 is a mutant-stable size (where the same size is used for text length, pattern length, and character set size), with a kill rate of 72.3%. On initial examination, the 20 surviving mutants do not seem problematic. A large number involve the `JS_ASSERT` converted to a `__CPROVER_assume`, showing

```
#include "bmh.h"
int main() {
  int i;
  unsigned int v;
  char itext[TSIZE];
  char ipat[PSIZE];
  unsigned int itext_s = nondet_uint();
  __CPROVER_assume(itext_s < TSIZE);
  unsigned int ipat_s = nondet_uint();
  __CPROVER_assume(ipat_s < PSIZE);
  printf ("LOG: size text=%u, pat=%u\n", itext_s, ipat_s);
  for (i = 0; i < itext_s; i++) {
    v = nondet_unit();
    __CPROVER_assume((long)v < (long)BMH_CHARSET_SIZE);
    itext[i] = v;
    __CPROVER_assume(itext[i] < BMH_CHARSET_SIZE);
    printf ("LOG: text[%d] = %u\n", i, itext[i]);
  }
  for (i = 0; i < ipat_s; i++) {
    v = nondet_uint();
    __CPROVER_assume((long)v < (long)BMH_CHARSET_SIZE);
    ipat[i] = v;
    __CPROVER_assume(ipat[i] < BMH_CHARSET_SIZE);
    printf ("LOG: pat[%d] = %u\n", i, ipat[i]);
  }
  jsint r = js_BoyerMooreHorspool(itext, itext_s,
                ipat, ipat_s, 0);
  printf ("LOG: return = %d\n", r);
  int pos, ppos, found;
  v = nondet_uint();
  printf ("LOG: looking at %u\n", v);
  __CPROVER_assume(v >= 0);
  if (r == -1) {
    __CPROVER_assume(v < itext_s);
    pos = v; ppos = 0; found = 1;
    while (ppos < ipat_s) {
      printf ("LOG: itext[%d] = %u, ipat[%d] = %u\n",
                pos, itext[pos], ppos, ipat[ppos]);
      if ((pos>=itext_s)||(itext[pos]!=ipat[ppos])) {
        found = 0; break;
      }
      pos++; ppos++;
    }
    assert (!found);
  } else {
    pos = r; ppos = 0;
    while (ppos < ipat_s) {
      assert (itext[pos] == ipat[ppos]);
      pos++; ppos++;
    }
    v = nondet_uint();
    printf ("LOG: looking at %u\n", v);
    __CPROVER_assume(v < r);
    pos = v; ppos = 0; found = 1;
    while (ppos < ipat_s) {
      printf ("LOG: itext[%d] = %u, ipat[%d] = %u\n",
                pos, itext[pos], ppos, ipat[ppos]);
      if ((pos>=itext_s)||(itext[pos]!=ipat[ppos])) {
        found = 0; break;
      }
      pos++; ppos++;
    }
    assert (!found);
  }
}
```

Fig. 13. Boyer-Moore-Horspool harness.

the harness cannot tell if the assumption is incorrect, which is not surprising (the harness only generates good inputs, and some of the mutants simply discard too many inputs).

At this point, we were satisfied with our harness, and ran a check on mutants of the harness itself. To our surprise, three mutants of the harness had a *better* kill rate than the "correct" harness, killing 73.5% of mutants. Investigating these "better" harnessess showed mutants that broke processing of some return values in such a way that, while these harnessess failed to detect certain major bugs in the code, they were able

to detect some JS_ASSERT assumption mutants. Guided by this, we produced a revised harness that raised the kill rate to 79.52%. However, on examining the surviving mutants, we realized that our verification was still unsatisfactory as a good regression for the Boyer-Moore-Horspool code: in particular, if the assertion were ever modified to allow bad inputs to pass through, or otherwise incorrectly changed, we would miss some mutants. We then changed the JS_ASSERT into code that returned a special value to signal assertion failure, and modified the harness once more, allowing some incorrect values to pass through and checking that "assertion failure" happened if, and only if, the inputs were invalid. This harness killed 89.2% of mutants, and the six surviving mutants were easily understood to be equivalent to the BMH code under all valid inputs (in one case we weren't certain about, we had CBMC verify that for all non-assertion violating inputs, this was true up to a large bound). The new harness, informed by the harness mutations, in fact had a better mutant kill rate for size 3 (80.7%) than our first harness had at the mutant-stable point. It was this example that convinced us that harness mutation was an important technique, not only for evaluating our methodology, but in its own right.

C. Linux Kernel RCU Verification Challenges

1) Introduction to RCU: RCU is a synchronization mechanism that is sometimes used as a replacement for reader-writer locking for linked structures, allowing extremely lightweight readers [24]. In the limiting case, which is achieved in server-class builds of the Linux kernel, the overhead of entering and exiting an *RCU read-side critical section* (using rcu_read_lock() and rcu_read_unlock(), respectively) is exactly zero [25], making RCU an excellent choice for read-mostly workloads [24], [26], [27].

However, lightweight readers imply that updaters cannot exclude readers, so must take care to avoid disrupting readers. Updaters typically maintain multiple versions of the portion of the data structure being updated, removing old versions only when readers are no longer accessing them. To this end, RCU provides synchronize_rcu(), which waits for a *grace period*: when all pre-existing RCU readers complete. RCU updaters typically remove a data element (thus rendering it inaccessible to new readers), invoke synchronize_rcu(), and then reclaim the newly removed element.

Because both RCU and the Linux kernel are moving targets, any validation and verification must be both automated and repeatable, for inclusion on a regression-test suite. At present the rcutorture stress-test provides some assurance in the form of automated testing, but ideally would be complemented by some formal verification of the implementation(s) in the kernel. An important question is whether available formal verification tools can provide effective additional regression checking for RCU.

We use a pair of RCU-related benchmarks [28], [29] to provide the beginnings of an answer to this question. The first benchmark applies formal verification to the simplest of the Linux kernel's RCU implementations, Tiny RCU [30], which targets single-CPU systems. This model includes Tiny RCU's handling of idle CPUs as well as its (trivial) grace-period detection scheme. The second benchmark creates the trivial

```
1 static int rcu_read_nesting_global;
2
3 static void rcu_read_lock(void)
4 {
5   (void)__sync_fetch_and_add(&rcu_read_nesting_global, 2);
6 }
7
8 static void rcu_read_unlock(void)
9 {
10  (void)__sync_fetch_and_add(&rcu_read_nesting_global, -2);
11 }
12
13 static void synchronize_rcu(void)
14 {
15   for (;;) {
16     if (__sync_fetch_and_xor(&rcu_read_nesting_global, 1) < 2)
17       return;
18     SET_NOASSERT();
19     return;
20   }
21 }
```

Fig. 14. Approximate Model of RCU

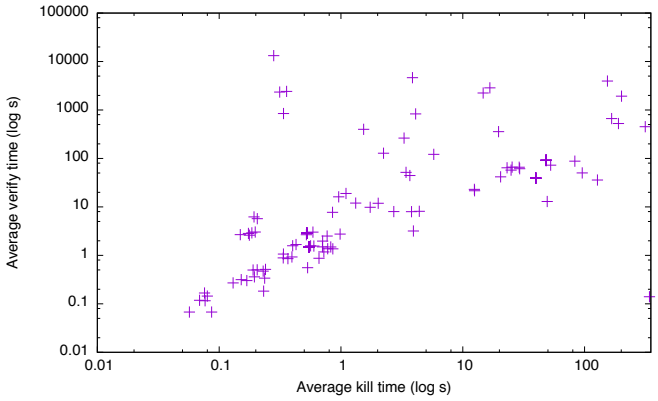

Fig. 15. Average times for killed mutants (SAT) vs. surviving mutants (UNSAT) for all experiments.

model approximating an RCU implementation for multiprocessor systems shown in Figure 14. In this model, the number of RCU read-side critical sections currently in flight is tracked by the global variable rcu_read_nesting_global, which is atomically incremented by rcu_read_lock() and atomically decremented by rcu_read_unlock(). This allows synchronize_rcu() to atomically XOR rcu_read_nesting_global's bottom bit to detect whether the current execution has waited for all pre-existing readers (over-approximated by checking the absence of all readers), with SET_NOASSERT() being invoked to suppress all future assertions. Although this model has a number of shortcomings, perhaps most prominently excessive read-side ordering, it is capable of detecting common RCU-usage bugs, including failure to wait for an RCU grace period and failure to enclose read-side references in an RCU read-side critical section.

D. Experiments: Plausible Verification by Failure to Falsify

A key problem in model checking is the state explosion problem, or, more simply (and more accurately, in that number of states is not always the determining factor in symbolic methods) the problem of scalability. As discussed above, even proving binary search correct over the full domain of integer inputs is not possible within a reasonable time frame. Even when verification is impossible at the desired problem size

falsification can provide limited confidence in the correctness of a program. In particular, we observe from all of our experiments that the average time, for any program and harness pair, to verify the original code and all surviving mutants is *much higher* than the average time to produce a counterexample for killed mutants. Showing that a constraint is satisfiable is, usually, easier than proving it is unsatisfiable. This is not limited to SAT solvers; we used SAT rather than SMT in our experiments because we generally found Z3 to be slower than CBMC's built in version of MiniSAT[31] in almost all cases, but Z3 also aims to be fast at producing satisfying assignments, not proving UNSAT [32], and our few runs with Z3 showed the same pattern.

Figure 15 shows (with log scales on both axes) the average running times for all experiments (including faulty versions of the harness, incorrect runtime parameters, harness mutation checks, etc.) performed in the course of this work. The general trend is clear: time to verify is usually worse than time to kill, and the worst average time to kill (about 350 seconds) is much better than many average verification times. One use of this relationship is that, in cases where all (non-equivalent) mutants of the SUT are killed, but the SUT verification fails to complete, the SUT might be considered *provisionally* verified. In particular, the larger the ratio between the timeout for failed verification and the longest kill time for any mutant, the "more likely" to be correct we can consider the SUT (the same holds with respect to memory use limits). This belief can be further justified by modifying the harness to force mutant kills to use large problem sizes, violating the usual inclusiveness rule (that way, if the new size allows a counterexample not previously existing, the mutant killing problem for mutants killable at smaller sizes better approximates the counterexample construction problem for the actual fault).

Additionally, the times shown here (with mean mutant kill time of 16.4 seconds and median mutant kill time of 0.54 seconds) show the general feasibility of the falsification-driven approach. Most of the time, killing mutants is cost-effective. The outliers come from a few difficult problems, arising from buggy harnesses (or harness mutants that resemble buggy harnesses). The much worse cost for surviving mutants is due to a few expensive stubborn mutants: the median verification success time is only 1.5 seconds.

V. Discussion

A. Falsification vs. Verification

The core idea of this paper is that, while successful verification is the *result* that a developer seeks when verifying a program, it is most meaningful in a context provided by many failed verifications. The useful model checking harness (e.g., specification) essentially, is one that *prohibits* certain execution sequences. This is not controversial; a good property is defined by its rejection of bad behavior. However, in most verification efforts, there is a focus on arriving at a successful verification, which sheds very little light on what, exactly is verified. By focusing on mutants throughout the verification process, our approach shifts the emphasis to one of "verifying" the verification itself by repeatedly *falsifying* claims that various incorrect programs satisfy the property. This is, at a conceptual level, akin to Karl Popper's philosophy of science [12].

For Popper, all scientific knowledge is provisional, and the key to the scientific approach is a critical effort, based on *prohibitive* theories. In brief, Popper proposes that proper science must be strongly grounded in a search for counterexamples. Using mutants as a basis for verification is akin to this approach, with the harness taken to be the "theory" of the empirical behavior of the world. Mutants, in this view, are counterfactual worlds that are likely to violate any correct theory of the actual world. A "scientific theory" (that is, a harness) is proven effective by its ability to be *shown to be false* in these counterfactual worlds. If we can prove a theory is incorrect for an "incorrect" world and cannot prove it is incorrect for the real world, that gives us greater confidence (always provisional, since our understanding of the world, e.g., any complex software system, is almost always limited and prone to error) that the theory is indeed true of the real world/program. Of course, generating alternative worlds and showing that, for example, special relativity is easily falsified in a world where special relativity does not in fact hold, is not practical in scientific discovery. It is, however, quite easy in the artificial "scientific discovery" sense of verifying properties of computer programs.

B. The Power of Exhaustive Nondeterminism

The ability to improve a harness based on surviving mutants (or on passing runs of killed mutants) essentially relies on the nature of exhaustive bounded model checking based on constraint solving. In non-exhaustive automated testing, the answer to why a mutant is not killed is, often, neither "the oracle is not good enough" or even "the test process is inadequate and needs to be modified" but "you didn't get lucky." That is, killing all mutants is, in many cases, not something we can expect of non-exhaustive test suites. Random testing [33], [34] can perform well in general as a bug-finding method, but its failure to kill any individual mutant is likely to be a matter of probability, rather than a flaw *per se* in the testing itself. In verification, however, there are no accidents: if a harness verifies an incorrect program, either the assumptions, the specification, or the problem size are *necessarily* in need of correction. However, the approach we propose is most suited to the analysis space in which CBMC is situated: on the one hand, within a known bound, its results are exhaustive; on the other hand, the method behaves much like a dynamic analysis, in that there are no false positives.

VI. Related Work

The idea that a "successful verification" in model checking (or even theorem proving) often simply indicates an inadequate property is long-standing [9], [8]. Use of mutants [17], [35] to provide a coverage measure dates back both to these early explorations and relatively recent work [36]. However, in these efforts the mutation was usually applied to hardware models, and (critically) the surviving mutants were used to identify "uncovered" portions of a model, rather than presented to a developer for examination and understanding directly. To our knowledge, no previous work presented passing executions of a source code mutant as a guide to understanding specification weakness. Our modification of the harness is a source-code analogue to attempts to modify logical formulas, e.g., the effort to (in a narrow, vacuity-based sense) produce the *strongest*

passing LTL formula of Chockler et al. [37]. We are not the first to note that model checking, at present, due to the "many obstacles" in proving a system correct, is primarily used for falsification [38]. Previous work on the topic [38] focused on abstractions based on under-approximation, to ensure counterexamples were not spurious. We instead preserve the goal of verification[7], but *drive* the verification process, from the human point of view, by repeated falsification of incorrect systems.

More distantly related is the general effort to determine the quality not only of test suites (which is often focused on missing tests within the "range" of testing, not a problem for CBMC) but of test oracles and entire testing infrastructures. The problem of "testing the tester" [6] is fundamental to all efforts to improve software quality. Recent efforts of most interest have focused on measuring *checked coverage* [39], [40], [41], where a metric tries to make sure the code under test potentially changes the value of an assert, using dynamic slicing [42], [43]. This is weaker than requiring the oracle kill a mutant, our goal, but more manageable for testing, where complete behavioral coverage is less feasible than in model checking (and where source code sizes combined with test inadequacy may make hand mutation analysis infeasible).

VII. Conclusions and Future Work

This paper proposes a *falsification-driven* methodology for formal verification, particularly when verification is performed by the developers of critical software systems. These developers are not experts in formal verification, but in the systems they are verifying. Verification is, we claim, always provisional, in that the potential flaws in our assumptions, specification, and understanding of system behavior tend to leave room for doubt about the correctness of any verification result. Verification of code is not self-explanatory, unlike a counterexample. We propose to take advantage of the use of counterexamples and witnesses and center verification around the *incorrect programs a verification fails to prove incorrect*. An obvious source of such programs is mutants; we also suggest that known-flawed versions of code be included in this set, which all of our tools support, but the key to the method is the generation of a large set of potential buggy versions without developer effort. Given these versions, a developer can examine mutants that a verification effort fails to detect, and (with the algorithms and tools presented in this paper) examine executions showing precisely how a program mutant can "make it through" a verification without being detected, with assurance that these executions will have high coverage (and thus likely be non-trivial), including of the mutant itself. Developers can also check that a verification harness does not have any mutants that 1) verify the SUT but 2) kill *more* mutants than their harness. This can help detect very subtle flaws in harnesses, especially those based on bad reasoning about "equivalent" mutants. We demonstrate, as a proof-of-concept, that our approach can be useful for simple verification efforts, and can contribute to serious systems verification and modeling efforts for the Linux kernel RCU implementations.

Our CBMC patch for `find-success` mode, the various verification harnesses, and our experimental results are all available at `https://github.com/agroce/cbmcmutate`.

[7]Note that we use a model checking approach that already guarantees non-spurious counterexamples, and provides bounded rather than full verification.

References

[1] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, 2000.

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

[3] E. W. Dijkstra, *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[4] A. Groce and R. Joshi, "Exploiting traces in program analysis," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2006, pp. 379–393.

[5] E. Clarke, O. Grumberg, K. McMillan, and X. Zhao, "Efficient generation of counterexamples and witnesses in symbolic model checking," in *Design Automation Conference*, 1995, pp. 427–432.

[6] A. Groce, "(Quickly) testing the tester via path coverage," in *Workshop on Dynamic Analysis*, 2009.

[7] A. Groce, M. A. Alipour, and R. Gopinath, "Coverage and its discontents," in *Onward! Essays*, 2014, pp. 255–268.

[8] Y. Hoskote, T. Kam, P.-H. Ho, and X. Zhao, "Coverage estimation for symbolic model checking," in *ACM/IEEE Design Automation Conference*, 1999, pp. 300–305.

[9] H. Chockler, O. Kupferman, R. P. Kurshan, and M. Y. Vardi, "A practical approach to coverage in model checking," in *Computer Aided Verification*, 2001, pp. 66–78.

[10] T. A. Budd, R. J. Lipton, R. A. DeMillo, and F. G. Sayward, *Mutation analysis*. Yale University, Department of Computer Science, 1979.

[11] R. J. Lipton, "Fault diagnosis of computer programs," Carnegie Mellon Univ., Tech. Rep., 1971.

[12] K. Popper, *The Logic of Scientific Discovery*. Hutchinson, 1959.

[13] R. Lawlor, "quicksort.c," http://www.comp.dit.ie/rlawlor/Alg_DS/sorting/quickSort.c, referenced April 20, 2015.

[14] D. Kroening and O. Strichman, "Efficient computation of recurrence diameters," in *Verification, Model Checking, and Abstract Interpretation*, 2003, pp. 298–309.

[15] 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.

[16] M. Papadakis, Y. Jia, M. Harman, and Y. L. Traon, "Trivial compiler equivalence: A large scale empirical study of a simple fast and effective equivalent mutant detection technique," in *International Conference on Software Engineering*, 2015.

[17] P. E. Black, V. Okun, and Y. Yesha, "Mutation of model checker specifications for test generation and evaluation," in *Mutation 2000*, 2000, pp. 14–20.

[18] R. Stout, *If Death Ever Slept*. Viking, 1957.

[19] J. Bentley, "Programming pearls: Writing correct programs," *Communications of the ACM*, vol. 26, no. 12, pp. 1040–1045, Dec. 1983.

[20] J. Bloch, "Extra, extra - read all about it: Nearly all binary searches and mergesorts are broken," http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html, June 2006.

[21] visar, "[solved] doubly linked list insertion sort in C," http://www.linuxquestions.org/questions/programming-9/doubly-linked-list-insertion-sort-in-c-4175415860/, July 2012.

[22] R. N. Horspool, "Practical fast searching in strings," *Software - Practice & Experience*, vol. 10, no. 6, pp. 501–506, 1980.

[23] M. A. Alipour, A. Groce, C. Zhang, A. Sanadaji, and G. Caushik, "Finding model-checkable needles in large source code haystacks: Modular bug-finding via static analysis and dynamic invariant discovery," in *International Workshop on Constraints in Formal Verification*, 2013.

[24] P. E. McKenney, "Structured deferral: synchronization via procrastination," *Commun. ACM*, vol. 56, no. 7, pp. 40–49, Jul. 2013. [Online]. Available: http://doi.acm.org/10.1145/2483852.2483867

[25] P. E. McKenney and J. D. Slingwine, "Read-copy update: Using execution history to solve concurrency problems," in *Parallel and Distributed Computing and Systems*, Las Vegas, NV, October 1998, pp. 509–518.

[26] D. Guniguntala, P. E. McKenney, J. Triplett, and J. Walpole, "The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with Linux," *IBM Systems Journal*, vol. 47, no. 2, pp. 221–236, May 2008.

[27] P. E. McKenney, D. Eggemann, and R. Randhawa, "Improving energy efficiency on asymmetric multiprocessing systems," June 2013, https://www.usenix.org/system/files/hotpar13-poster8-mckenney.pdf.

[28] P. E. McKenney, "Verification challenge 4: Tiny RCU," http://paulmck.livejournal.com/39343.html, March 2015.

[29] ——, "Verification challenge 5: Uses of RCU," http://paulmck.livejournal.com/39793.html, April 2015.

[30] ——, "Re: [PATCH fyi] RCU: the bloatwatch edition," January 2009, available: http://lkml.org/lkml/2009/1/14/449 [Viewed January 15, 2009].

[31] N. Een and N. Sorensson, "An extensible SAT-solver," in *Symposium on the Theory and Applications of Satisfiability Testing (SAT)*, 2003, pp. 502–518.

[32] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2008, pp. 337–340.

[33] R. Hamlet, "When only random testing will do," in *International Workshop on Random Testing*, 2006, pp. 1–9.

[34] 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.

[35] T.-C. Lee and P.-A. Hsiung, "Mutation coverage estimation for model checking," in *Automated Technology for Verification and Analysis*, 2004, pp. 354–368.

[36] H. Chockler, D. Kroening, and M. Purandare, "Computing mutation coverage in interpolation-based model checking," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 31, no. 5, pp. 765–778, 2012.

[37] H. Chockler, A. Gurfinkel, and O. Strichman, "Beyond vacuity: Towards the strongest passing formula," in *Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design*, 2008, pp. 24:1–24:8.

[38] T. Ball, O. Kupferman, and G. Yorsh, "Abstraction for falsification," in *Computer Aided Verification*, 2005, pp. 67–81.

[39] D. Schuler and A. Zeller, "Assessing oracle quality with checked coverage," in *International Conference on Software Testing, Verification and Validation*, 2011, pp. 90–99.

[40] ——, "Checked coverage: an indicator for oracle quality," *Software Testing, Verification, and Reliability*, vol. 23, no. 7, pp. 531–551, 2013.

[41] A. Murugesan, M. W. Whalen, N. Rungta, O. Tkachuk, S. Person, M. P. E. Heimdahl, and D. You, "Are we there yet? determining the adequacy of formalized requirements and test suites," in *NASA Formal Methods Symposium*, 2015, pp. 279–294.

[42] X. Zhang, R. Gupta, and Y. Zhang, "Precise dynamic slicing algorithms," in *International Conference on Software Engineering*, 2003, pp. 319–329.

[43] F. Tip, "A survey of program slicing techniques," *Journal of programming languages*, vol. 3, pp. 121–189, 1995.