How Verified is My Code? 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 finally advanced to a state where non-experts, including systems software developers, may want to verify the correctness of small but critical modules. Unfortunately, despite considerable efforts in the area, determining if a "verification" actually verifies what the author intends it to is still difficult, even for model checking experts. Previous approaches from the model checking community are valuable, but difficult to understand and limited in applicability. Developers using a tool like a bounded model checker need verification coverage in terms of the software they are verifying, rather than in model checking terms. In this paper we propose a tool framework and methodology to allow both developers and expert users to determine, more precisely, just what it is that they have verified for software systems. Our basic approach is based on a novel variation of mutation analysis, a conceptual model of verification based on Popper's notion of falsification, and even empirical examination of the ease of SAT/SMT solving in different cases. We use the popular C/C++ bounded model checker CBMC, modified to allow a user to determine the "strength" of a mutant, and show that our approach is applicable not only to simple (but complete, within bounds) verification of data structures and sorting routines, but to understanding efforts to verify the Linux kernel Read-Copy-Update mechanism, code from Mozilla's JavaScript engine, and other real-world examples.**

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 a CBMC harness for sorting routines. This is a simple program, but typical of the structure of a small verification problem. Only a few aspects of this code differ from normal testing. First, `nondet_int` in CBMC can return any value, nondeterministically. It is not equivalent to a "random" choice but true nondeterminism: CBMC will explore all possible values. The `__CPROVER_assume` statement is used to restrict the program executions considered: it has the usual `assume` semantics [3], [4], so CBMC ignores all executions that violate assumptions.

```
#include <stdio.h>
#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.

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 such 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 (counting the iteration where the bound is exceeded).

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 status of the verification effort is clear and the resulting output (a detailed trace, including the output of any

[1] By a harness we mean a program that defines the environment in which the software is verified, defines the form of valid tests, and provides correctness properties; in CBMC such a harness looks very similar to a harness for more traditional software testing.

print statements) 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 allowable 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, however, they have typically been aimed at the hardware verification community, and often useful primarily to experts in formal verification. In this paper, we adapt more 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 will not suffice in this case: simply noting a mutation kill rate is not enough. The typical 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 show 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.

Our primary contribution in this paper is a falsification-driven approach to verification: an extension of traditional mutation testing that aids the user of a model checker in understanding successful (and "successful") verification results, determining when a harness is not actually strong enough to ensure correctness, and correcting the verification 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 to automatically produce *successful high-coverage executions covering mutated code* in order to understand mutant behavior and find subtle harness flaws. We also propose the use of mutation analysis to gain limited confidence in program correctness even past model checker scalability limits. At a more general level, we discuss the fundamental nature of "verification" in a real-world context

```
#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.

where specifications are never known to be complete. We propose that falsification, as in certain theories of natural science, is a more useful conceptual framework for most software verification efforts: rather than focusing on what can be proven about a program, it may be best to focus on how a verification effort distinguishes the "real" program from similar alternative programs that can be shown to *not* match the theory of program behavior.

II. A Simple Example Verification

As an example of the proposed verification methdology, consider again the harness shown in Figure 1. If we take the first hit on Google for "quick sort in C" [12], 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 `VERIFICATON SUCCESSFUL` in less than a second. Does this mean we have verified what we want to verify? How do we understand this "successful" verification result better?

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 [13] 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

[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 computational 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`.

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 [14], 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, and 71 mutants pass without detection. 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 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.

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 6, and mutations can be checked in parallel, but the fundamental problem for size 4 (and above) is that some individual mutants require hours to produce a `VERIFICATION SUCCESSFUL` result. 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

```
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`.

[15]. We suggest always pruning mutants via Trivial Compiler Equivalence (TCE) before working with mutants. 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 truth value on one corner case (when two values are equal), 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. Running CBMC in this new mode, with the target of maximum branch coverage and statement coverage of the `del_stmt` mutant (or, rather, the statement after it in the CFG, since it no longer exists), we produce the "counterexample" in Figure 4 in less than a minute[4]. 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 original array!

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 `a` (a kind of most-general any-order type-correct

[4]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.

"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 choose 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 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 needs to be modified. It is too *weak*. We say that 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 this additional critical property[5] . 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. This is a useful reminder that while a CBMC harness resembles a C program to test the SUT, it can make use of approaches to testing that are not viable in a non-exhaustive context.

If we modify the harness as above, we can re-check our mutants. In practice, we would remove the two we determined are equivalent based on compiler output, but for the sake of comparison we retain all mutants. 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 the run now only requires slightly over 5 minutes. Finally, at `SIZE=3`, we kill the `del_stmt` mutant that previously survived, and the process requires only 14 minutes, only slightly more than at `SIZE` 2 with the weaker harness. The `SIZE` 3 verification is stable. Checking this by running all surviving mutants at `SIZE` 4

[5]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.

```
(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. 6. Algorithm 1: Finding size/unwinding bound and surviving mutants.

now only requires slightly more than an hour, nearly an order of magnitude faster than with the weaker harness. A stronger harness not only finds more bugs in the SUT, it often *speeds up verification* by making the set of satisfying instances larger for the SAT solver. Even UNSAT is easier to prove when the possible executions to consider are reduced, in many cases.

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 [16]. 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 our fixed 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 much 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.

```
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. 7. 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. 8. Algorithm 3: Instrument mutant for coverage.

III. Algorithms and Techniques

IV. Experimental Results and Case Studies

A. Algorithm Implementations

1) Binary Search:

2) Doubly-linked List Insertion Sort:

3) AVL Tree:

4) Merge With Duplicate Removal:

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

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

```
int main () {
  int a[SIZE], b[SIZE], c[SIZE*2];
  int i, v, i1, i2, csize;
  int asize = nondet_int();
  int bsize = nondet_int();
  __CPROVER_assume ((asize >= 0) && (bsize >=0));
  __CPROVER_assume ((asize <= SIZE) && (bsize <= SIZE));
  for (i = 0; i < asize; i++) {
    a[i] = nondet_int();
    __CPROVER_assume((i == 0) || (a[i] >= a[i-1]));
  }
  for (i = 0; i < bsize; i++) {
    b[i] = nondet_int();
    __CPROVER_assume((i == 0) || (b[i] >= b[i-1]));
  }
  csize = merge_sorted_nodups(a, asize, b, bsize, c);
  assert (csize <= (asize + bsize));
  i1 = nondet_int();
  i2 = nondet_int();
  __CPROVER_assume((i1 >= 0) && (i2 >= 0));
  __CPROVER_assume((i1 < csize) & & (i2 < csize));
  __CPROVER_assume(i1 != i2);
  assert(c[i1] != c[i2]);
  v = nondet_int();
  __CPROVER_assume ((v >= 0) && (v < asize));
  v = a[v];
  int found = 0;
  for (i = 0; i < csize; i++) {
    if (c[i] == v)
      found = 1;
  }
  assert (found == 1);
  v = nondet_int();
  __CPROVER_assume ((v >= 0) && (v < bsize));
  v = b[v];
  int found = 0;
  for (i = 0; i < csize; i++) {
    if (c[i] == v)
      found = 1;
  }
  assert (found == 1);
}
```

Fig. 10. Harness for merge_sorted_nodups

```
int merge_sorted_nodups(int a[], int asize,
                        int b[], int bsize, int c[]) {
  int apos = 0, bpos = 0, cpos = -1, csize = 0;
  while ((apos < asize) || (bpos < bsize)) {
    if ((apos < asize) &&
        ((bpos >= bsize) || (a[apos] < b[bpos]))) {
      if ((cpos == -1) || (c[cpos] != a[apos])) {
        c[++cpos] = a[apos];
        csize++;
      }
      apos++;
    } else {
      if ((cpos == -1) || (c[cpos] != b[bpos])) {
        c[++cpos] = b[bpos];
        csize++;
      }
      bpos++;
    }
  }
  return csize;
}
```

Fig. 11. Code to merge two sorted arrays into one sorted array with no duplicate elements

B. SpiderMonkey Boyer-Moore-Horspool Implementation

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, but with extremely lightweight readers [**?**]. 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 [**?**]. Thus, RCU can achieve excellent performance, scalability, real-time response, and energy efficiency for read-mostly workloads [**?**], [**?**], [**?**].

However, lightweight readers imply that updaters cannot exclude readers, which means that updaters 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*, that is, for all pre-existing RCU readers to complete. RCU updaters typically removes a data element (thus rendering it inaccessible to new readers), invokes synchronize_rcu(), and then reclaims the newly removed element. Production-quality implementations of synchronize_rcu() use batching techniques to achieve extremely scalability, so that a single underlying grace period can satisfy more than 1,000 concurrent updates in the Linux kernel [**?**].

The Linux kernel contains more than 10,000 uses of the RCU API [**?**], and a userspace RCU library [**?**], [**?**] is seeing significant use in userspace applications. That said, it is clear that validation and verification is a first-class concern for RCU implementations.

Should we cite this? http://conf.researchr.org/event/pldi2015/pldi2015-papers-verifying-read-copy-update-in-a-logic-for-weak-memory

Because both RCU and the Linux kernel are moving targets, any validation and verification must be both automated and repeatable, that is, must be something that could be included in a regression-test suite. The rcutorture stress-test suite qualifies, but it would be good to evaluate its effectiveness on the one hand (via permutation analysis) and to include formal verification on the other. However, regression-test use of formal verification required that the following properties: (1) Direct use of source code, (2) Automatic discarding of irrelevant source statements, (3) Reasonable memory and CPU overhead, (4) Automatic mapping of error reports to source lines, and (5) Modest additional input beyond the source code. Failing to provide any of these requirements provides unacceptable vulnerability to human error. This raise the question of whether readily available formal-verification tools meet these requirements.

We use a pair of RCU-related benchmarks 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 [**?**], which targets single-CPU systems. This model includes Tiny RCU's handling of idle CPUs as well as the its (trivial) grace-period detection scheme. The second benchmark creates the trivial model

```
 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. 12. Approximate Model of RCU

approximating an RCU implementation for multiprocessor systems shown in Figure 12. 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, perhsps most prominently excessive read-side ordering, it is capable of detecting common RCU-usage bugs, including failure to wait for an RCU grace period, failure to enclose read-side references in an RCU read-side critical section.

Nevertheless, it is important to note that these two benchmarks are not examples of useful verifications, but rather the simplest examples of a class that might some day expand to include useful verifications.

(And of course, CBMC proved capable of handling simple variable-pair litmus tests, as well as single-threaded examples involving linked lists, but proved incapable of handling multi-threaded tests involving linked lists. The maintainers of CBMC are working to improve its handling of pointers.)

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 [17].

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 "science" of verifying a property of a computer program.

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" 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 [18], [19] 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 neccessarily in need of correction.

VI. Related Work

The idea that a "successful verification" often simply indicates an inadequate property is long-standing [9], [8]. Use of mutants [16], [20] to provide a coverage measure dates back both to these early explorations and relatively recent work [21]. 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. [22]. 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 [23]. Previous work on the topic [23] focused on abstractions based on underapproximation, to ensure counterexamples were not spurious. We instead preserve the goal of verification[6], but drive the verification process, from the human point of view, by repeated falsification of incorrect systems.

Our idea of examining successful executions to better understand surviving (and even killed) mutants is a peculiar variation of the fault localization and error explanation problem in model checking [24], with the twist being that we are "explaining" an artificial fault that 1) typically does not cause a test failure (for surviving mutants) and 2) has an obviously known location.

VII. Conclusions and Future Work

Acknowledgment

The authors would like to thank...

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

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

[14] 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.

[15] 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.

[16] 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.

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

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

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

[19] 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.

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

[21] 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.

[22] 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.

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

[24] A. Groce, "Error explanation with distance metrics," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2004, pp. 108–122.