**Automated Software Engineering manuscript No.**
(will be inserted by the editor)

# How Verified (or Tested) is My Code? Falsification-Driven Verification and Testing

**Alex Groce · Iftekhar Ahmed · Carlos Jensen · Paul E. McKenney · Josie Holmes**

the date of receipt and acceptance should be inserted later

**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 CBMC model checker to 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 (RCU) mechanism.

## 1 Introduction

> "Every 'good' scientific theory is a prohibition: it forbids certain things to happen. The more a theory forbids, the better it is."
> -Popper, *Conjectures and Refutations: The Growth of Scientific Knowledge* [57]

Software model checking [15] has recently, thanks to improvements in model checking tools (and advanced SAT/SMT solvers), become a potentially valuable tool for developers of critical software modules who want to either perform a very aggressive search for bugs or, ideally, prove correctness of their code. Tools such as CBMC [39] (the C Bounded Model Checker) allow a software engineer to model check code by writing what is essentially a generalized test harness [25, 28][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,

Alex Groce
Northern Arizona University

[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.

`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 has the usual `assume` semantics [17,27]: CBMC ignores all executions that violate assumptions.

CBMC compiles a harness and the SUT (here a quicksort 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 [14]. 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.

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 counterexample 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.

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? Is the SUT actually correct? Formal verification is not only subject to the problems that make "no faults detected" results dubious in testing [22,24], 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. Moreover, formal verification tools are themselves extremely complex software artifacts, and, like production compilers [65], may themselves have serious bugs that produce wrong results [16]. In the course of this research, we have ourselves encountered several tool-induced incorrect verifications.

The problem of checking verification results has concerned the model checking community for some time, and resulted in efforts to define *coverage metrics* for model checking [37,13]. 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 [9,45] to the problem of software verification. A mutant of a program is a version of the program with 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 (bounded) verification rather than merely good testing, surviving mutants are likely to indicate a real problem.

In software engineering research, mutation is often used only as a way to compare competing test suites by comparing kill rates [19,20]. This is not enough for verification. The typically small scope of the code to be verified, and the presumed importance of verified code suggests an approach in which *individual mutants* are examined by the developer. Without additional assistance, such an approach cannot scale. This paper aims to describe how to make this seemingly too-demanding approach practical for real verification tasks.

Our basic idea is to use mutants *throughout the verification effort*, even in choosing a bound for bounded model checking. At each stage the developer examines the currently surviving mutants, either by inspecting the mutated code or (when this does not make the reason the mutant is not detected clear) looking at *successful executions covering the mutant but satisfying the specification given in the harness*. For critical verification tasks, we suggest that developers not only examine the passing executions of surviving mutants, but the passing executions of *killed mutants*. While examining test cases that do not kill a given mutant could be useful in traditional testing, the model checker makes a much more potent investigation possible, where a developer can constrain the behavior to force the mutant's behavior to matter, if that is possible, and automatically find passing executions that maximize coverage (including the mutated code). We also propose that a developer should use mutants of the test harness itself to ensure that no similar harness has a better mutant kill rate, and that most mutants of the harness reject the SUT itself.

### 1.1 Contributions

This paper is an extension of a paper presented at the 30th IEEE/ACM International Conference on Automated Software Engineering in 2015 [23]. The core contribution of that earlier paper was a *falsification-driven* verification methodology using mutants to aid developers in understanding "successful" verification results, determining when a harness is flawed, and correcting the harness. It showed 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 (and thus harness) behavior. This approach, unlike a simpler method of searching for cases where the mutated and original code behave differently for identical inputs, in principle applied to verification of reactive and concurrent systems, where there is no simple notion of identical inputs. It also proposed the use of mutation analysis to gain limited confidence in program correctness even past model checker scalability limits. At a more general level, the original paper discussed the fundamental nature of "verification" in a real-world context where specifications are never known to be complete. The central concept of that paper, and this extension, is that falsification, as in Karl Popper's famous philosophy of scientific discovery [56], is the critical element of efforts to understand systems, efforts that are always provisional by nature: e.g., most program verification (and even more so, testing) efforts. Popper's approach therefore forms a useful conceptual framework for verification efforts: rather than focusing on what can be proven about a program, we propose that correctness-determination efforts focus on how a verification distinguishes the "real" program from similar alternative programs that do *not* match the theory of program behavior. Such an approach still aims at verification as a final outcome, but continually evaluates and refines that verification effort by its ability to *falsify* rather than to verify.

In this paper, we extend the contributions of the previous work by further elaborating the approach to bounded model checking presented there, but, most significantly, we extend the same ideas to automated software testing, where there are new challenges and it is less clear that the underlying concepts are sound. In our previous work, we dismissed the possibility of using our approach for testing, because when a typical test generation approach fails to find a failing test (that is, testing's version of a counterexample), it does not mean the property is insufficient, or even that the generation is weak. Testing is usually essentially probabilistic. Here we show that this limitation is not fundamental, and the same principles can apply to serious automated test generation efforts as well. The general ideas are analogous, including the modification of a test generation tool to generate high-coverage tests that 1) fail to kill a mutant but 2) cover the mutated code, in order to facilitate understanding of the limits of a harness, but the details require considerable modification in order to adapt to the probabilistic context of test generation. For example, the equivalent of a problem size for bounded model checking is a test budget for automated test generation; however, rather than a fixed outcome, the problem in this case is to present a likelyhood curve to users, and allow them to make tradeoffs based on that curve.

We still initially present our approach in the context of formal verification, where it is most obviously useful, and where there is a higher chance of killing enough mutants to make manual analysis of un-killed mutants feasible.

## 2 A Simple Example Verification

As an example of the proposed verification methodology, consider again the harness shown in Figure 1. If we take an early Google result for "quicksort in C" [43],

```
#include "sort.h"
void quickSort( int a[], int l, int r)
{
  printf ("LOG: called with l=%d, r=%d\n", l, r);
  int j;
  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;
  while( 1)
    {
      do ++i; while( i <= r && a[i] <= pivot );
      do --j; while( a[j] > pivot );
      if( i >= j ) break;
      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** Quicksort code.

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 maximum number of items in the array). CBMC reports `VERIFICATION SUCCESSFUL` in less than a second. Have we verified what we want to verify?

## 2.1 Finding a Good Problem Size

The first question we face is whether 2 is 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 [40] 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 [3], 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 16GB RAM and 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

[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`.

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-checking the mutants with this bound (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 increase the size limit to 3 and unwinding 4 (now the analysis takes 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, includes 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 inclusiveness in this paper[4].

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 the simple (if highly imprecise) heuristic of 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?

### 2.2 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 number in the quicksort 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.

[4] There is one noted exception in Section 4.4.

```
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.

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 [54]. 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 quicksort 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 quicksort 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 3). 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 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

[5] We show the output of the print statements, not the full CBMC trace: this is what a developer will examine first.

```
...
   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`.

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.

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`, there exists a `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 analysis 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.

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 mutant differ(s) from the original. However, this

[6] In fact, if we choose a `val` to check before we assign to `ref`, we could completely dispense with storing `ref` at all.

increases the complexity of the model checking problem (checking equivalence of two functions is often harder than 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 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.

### 2.3 Mutating the Harness

Previous efforts to understand model checking results have also considered mutants to the property, usually given as a temporal logic formula [7]. 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, quicksort. 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 harness 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 preserves semantics. 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 quicksort 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 `s < SIZE` rather than `s <= SIZE`, which is the same as lowering the `SIZE` by one; this is a fairly easy mistake to make in a harness (or any 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 quicksort in general, since the permutation requirement remains in place. However, it cannot detect versions of quicksort 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 developer the problem in this case. Our modified CBMC easily produces a witness showing a permuted array with correct `a[0]` but with out-of-order later elements.

The basic flow of our approach, for bounded model checking, can be summarized as follows:

1. Generate mutant set $M = m_1 \ldots m_n$ for the program $P$.
2. Prune $M$ into $M'$ by equivalence classes based on optimizing compiler output, removing mutants that fail to compile or are equal to the original code.
3. Set unwinding depth/problem size $U$ to 0.
4. Set $r = 0, r' = 1$.
5. While $r! = r'$:
   (a) Set $U = U + 1$.
   (b) Set $r = r'$
   (c) Set $K = \emptyset, S = \emptyset$.
   (d) Check each mutant $m_i \in M$ using $H$ and size $U$: if $m_i$ is killed, $K = K \cup \{m_i\}$, otherwise $S = S \cup \{m_i\}$.
   (e) Set $r' = |K|/|M|$.
6. Examine each mutant in $S$. Remove those that are, by inspection, semantically equivalent to $P$.
7. Modify harness $H$ for mutants in $S$ that indicate a clear violation of the specification, easily understood, until $H$ kills all such mutants. Remove them from $S$ and add them to $K$.
8. For remaining mutants in $S$, generate a successful execution that covers the mutant but satisfied $H$. If the execution is degenerate, add constraints removing that class of execution until a witness to an incorrect, mutant-covering behavior is produced. Use this to modify $H$ and move newly killed mutants from $S$ to $K$.
9. Take mutants in $m_i \in K$, and check whether there exists a successful execution of $m_i$ satisfying $H$. Examine and constrain each such execution to remove degenerate solutions, modifying $H$ as needed.
10. Compute mutants $M_H$ of the harness, and check that all mutants either: produce a counterexample for the original program $P$ or have a kill rate $\leq$ the kill rate for $H$.

## 3 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 (guided by experience) [61] 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 tool-chain is non-standard), other than the model checker itself.

Figures 7-9 show core algorithms (implemented as prototype tools in our framework). In these algorithms $O(S)$is a function mapping an abstract size into particular options, e.g. `-DSIZE`. The uses of these algorithms are described at a high level in the introductory example, and in the case studies below. One additional requirement is a version of CBMC capable of converting built-in assertions checks (e.g, bounds checks, pointer dereference, division by zero) to assumptions. For harness assumptions, this is done by automatic source-to-source transformation (Figure 8, procedure `cover-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.

The `find-size` algorithm (Figure 7) finds a suitable problem size and returns the set of surviving mutants for a harness and program, performing as few model checker calls as possible (once we know a bound is not stable, we move on to the next bound). This algorithm can be easily parallelized by running mutants in the `for` loop at the same time, with any `goto TOP` killing all CBMC runs not terminated. The `maxcover` algorithm (Figure 8) returns for a given mutant and harness, a witness program trace that 1) covers the mutant and 2) covers as much other code as possible (in terms of branch coverage), using the `cover-harness` and `cover-mutant` procedures to instrument harness and mutant; it proceeds by starting with a minimal constraint on coverage (the trace must cover the mutated code) and increases this bound by incrementing it to one more than the actual coverage of the last witness found, until the model checker can prove the coverage is impossible. Other strategies for maximal coverage are possible (trying maximal coverage first, and decreasing the required coverage as attempts fail) but this approach minimizes the number of model checker runs that will fail to produce a witness, which is critical for performance reasons (see Section 4.4). The `check-harness` algorithm (Figure 9) analyzes harness mutants, producing a report of 1) harness mutants that are killed (either they do not verify the SUT or they have worse kill rates than the original harness), that are equal to the original harness in strength, and that are stronger than the original harness. It also returns information on all mutants killed by any harness mutant (except those that reject the SUT) that are not killed by the original harness. The algorithm `killed`, not shown, simply returns the set of mutants killed by a given harness. In our implementations, these tools perform additional record-keeping. For example, harness analysis records killing counterexamples and execution times for each run. We also make use of convenience scripts such as a tool to automatically call `maxcover` on all mutants, which 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 in our experiments and are available, along with our experimental data and CBMC patch, at `https://github.com/agroce/cbmcmutate`.

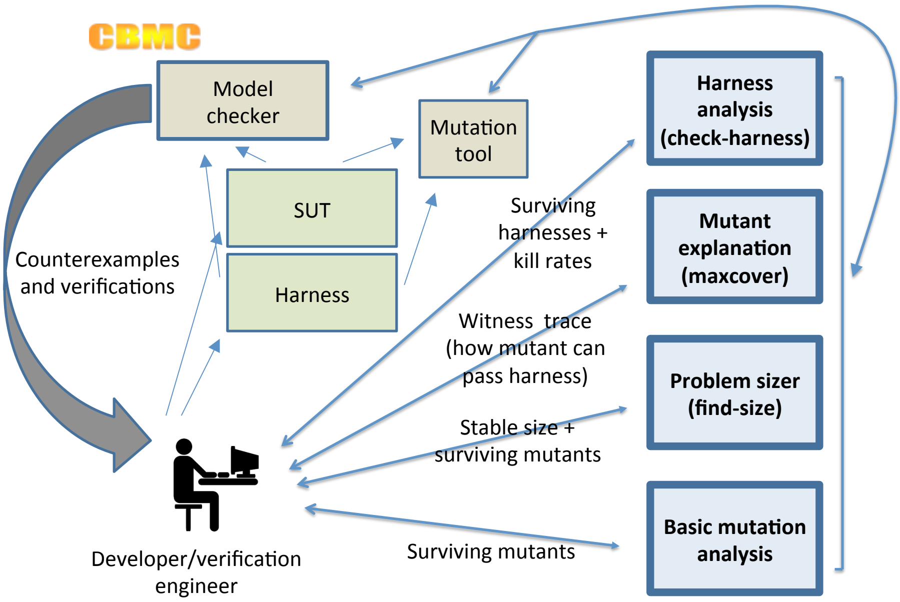


**Fig. 6** Basic flow of falsification-driven verification.

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

  S = S₀-1
  r' = {} : mutant → result
  TOP:
  S = S + 1
  r = r'
  r' = {}
  for m ∈ M:
     if m ∉ domain(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
           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 cover-harness (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'



mutant cover-mutant (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'



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

  m' = cover-mutant(m)
  T = 0
  trace = ∅
  failed = False
  while (not failed)
     H' = cover-harness(H, T)
     r = scheck(H,m',U(S),O(S))
     if r == VERIFICATION SUCCESSFUL:
       failed = True
     else if r == VERIFICATION FAILED:
       trace = r.trace
       T = trace.read(total_covered) + 1
  return trace
```

**Fig. 8** Algorithm 2: Find a maximally covering execution trace that covers a mutant.

### 3.1 Adapting Falsification-Based Approaches to Automated Test Generation

A key insight is that when estimating how hard a mutant is, a single run that either takes a long time to kill the mutant or a run that fails to kill a (known-killable) mutant is sufficient evidence to assume the mutant is difficult, and helps establish a lower bound on test budget needed to reduce risk of missing faults; in contrast, a single run that quickly kills a mutant does not establish that the mutant is in

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

  K_H = killed(M, H, S)
  Hkills = ∅; Hequal = ∅; Hbetter = ∅; N = ∅
  for H_i in M(H):
     original = check(H_i, SUT, U(S), O(S))
     if original == VERIFICATION FAILED:
        Hkills += H_i
     else: // check if this kills fewer mutants
        K_{H_i} = killed(M, H, S)
        for k ∈ K_{H_i}:
           if k ∉ K_H: N += (H_i, k)
        if |K_{H_i}| > |K_H|:
           Hbetter += (H_i, K_{H_i})
        if |K_{H_i}| == |K_H|:
           Hequal += (H_i, K_{H_i})
        else:
           Hkills += (H_i, K_{H_i})
  return (Hkills, Hequal, Hbetter, N)
```

**Fig. 9** Algorithm 3: Analyze a harness.

fact easy: even hard mutants can sometimes be easy to detect. In part we base this idea on empirical evidence (see below), but it can also be justified by a simple theoretical model of test generation.

The difficulty of detecting a fault (or killing a mutant, which is equivalent) can be simply expressed by a probability of detection, e.g. a trivial fault is detected with almost every generated test (99/100 tests), a typical easy fault is detected frequently (1/100 tests), and a hard fault is detected two orders of magnitude or more less frequently (1/5000 tests). In fact, real faults often seem to fall into such coarse "buckets" of detection ease, though this is not required for our analysis [10].

Figure 10 shows how the minimum number of tests, mean number of tests, and maximum number of tests needed for detection, over 100 runs each consisting of 10,000 tests, vary as the difficulty of a mutant or fault changes. For simplicity, we measure difficulty by increasing $N$, where the probability of detection is $1/N$. As you can see, while the maximum number of tests required increases rapidly, and the mean number of tests also increases steadily with difficulty, the minimum number of tests required increases very slowly. A concrete example shows why: the chance of getting "lucky" and hitting a test with difficulty 1000 on the first test is indeed small, only 1/1000. However, the chance of getting "unlucky" and failing to detect a test with difficulty 1/10 for even as few as 100 tests is less than $1/30,000$.

## 4 Case Studies and Experimental Results

### 4.1 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. Here we highlight the most interesting of these; we also successfully applied the method to bubble sort and a student's harness for verifying a version of Dijkstra's shortest path algorithm that enables path reconstruction [60]. For the Dijkstra harness, the low mutant kill rate of only 58% showed that while

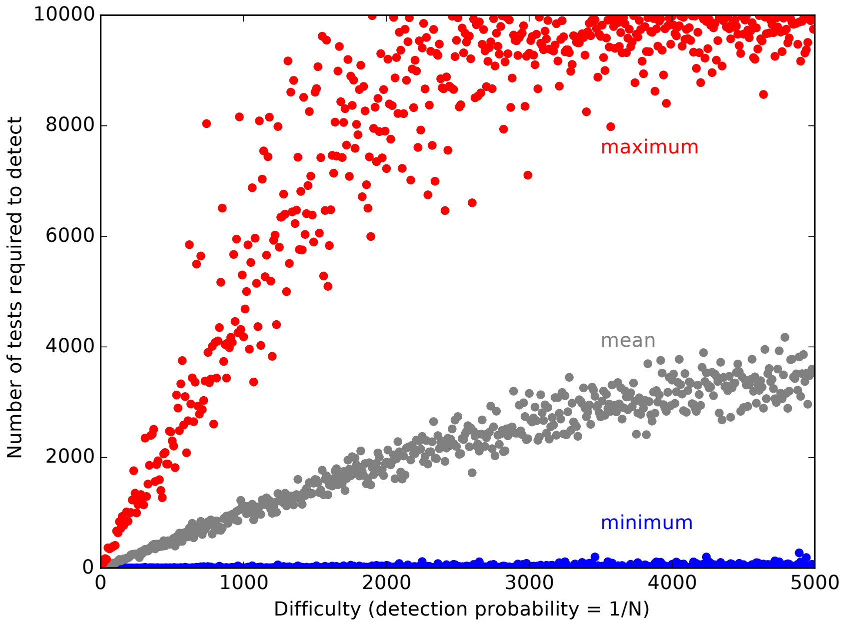


**Fig. 10** Tests required for detection as difficult of fault/mutant increases.

the harness checks incorrect returned paths, it cannot detect when return values indicate there is no path but one exists. Improving the harness is a substantial exercise, but can be guided by the survival witnesses.

#### *4.1.1 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 [6] in the context of modern software verification tools (and Joshua Bloch's discovery of a bug in the assumptions behind Bentley's proof [8]). 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 problem clearly (the sought item at index 0).

#### *4.1.2 Doubly-linked-list Insertion Sort*

Another example, making use of recursive data structures and pointer validity checks, is code for inserting an item (in sorted order) into a doubly-linked list [63]. 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 quicksort 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` permutation) the harness can still detect all mutants. Removing two, however, fails to kill mutants. The two harness mutants with worse kill rates have extremely poor kill rates ($< 50\%$ and $< 25\%$).

#### *4.1.3 AVL Tree*

In the case of the AVL tree, the harness we were working with was unable to reach a mutant-stable unwinding without exhausting memory on the verification of the main program (for AVL trees of up to size 5). We are investigating a more efficient harness encoding, based on the inability to reach mutant-stability.

#### *4.1.4 Merge With Duplicate Removal*

Even a killed mutant—one the harness does detect—can shed critical light on harness vulnerabilities. For example, the code in Figure 11 is a portion of a harness to verify code that merges two sorted arrays, removing all duplicates (the source arrays may contain duplicates or shared items, the output array is guaranteed to be sorted and have all-unique values). This harness detects all non-equivalent mutants of the source code with an unwinding depth of only 2 (the check requires less than a minute).

However, as is well known, many software faults [38] are not represented by a mutant. Because we are model checking, we want our harness to actually rule out *all* bad runs of the program under test. Even a killed mutant's passing executions

```
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. 11** 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. 12** Code to merge two sorted arrays into one sorted array with no duplicate elements

```
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. 13** SpiderMonkey 1.6 Boyer-Moore-Horspool code.

may show such a problem. Here we see that when the output array's size is 1, the way we have written the duplicate check in fact *assume*s away *all executions*! We check no properties of size 1 output arrays, and a fault that only appears with size = 1 will never be detected. No mutant produces such behavior, but noting an incorrect but passing trace of this run using the CBMC extension lets us see the problem.

## 4.2 SpiderMonkey Boyer-Moore-Horspool Implementation

Figures 13 and 14 show, respectively, the source code and an initial harness for verification of the Boyer-Moore-Horspool substring finding algorithm [36, 2] 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 precondition for the code [2]. 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

```
#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. 14** Boyer-Moore-Horspool harness.

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 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" harnesses showed mutants that broke processing of some return values in such a way that, while these harnesses 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 those bugs. 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 size 10). 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. This examples serves as our best evidence of the value of harness mutation.

### 4.3 Linux Kernel RCU Verification Challenges

Read-Copy-Update (RCU) is a synchronization mechanism sometimes used as a replacement for reader-writer locking for linked structures, allowing extremely lightweight readers [47]. In the limiting case, achieved in server-class builds of the Linux kernel, overhead for entering and exiting an *RCU read-side critical section* (using `rcu_read_lock()` and `rcu_read_unlock()`, respectively) is exactly zero [51], making RCU an excellent choice for read-mostly workloads [47,33,50]. However, lightweight readers mean 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 (rendering it inaccessible to new readers), invoke `synchronize_rcu()`, and then reclaim a removed element.

Because both RCU and the Linux kernel are moving targets, any validation and verification must be both automated and repeatable, for inclusion in 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 check-

```
static int rcu_read_nesting_global;

static void rcu_read_lock(void)
{
  (void)__sync_fetch_and_add(&rcu_read_nesting_global, 2);
}

static void rcu_read_unlock(void)
{
  (void)__sync_fetch_and_add(&rcu_read_nesting_global, -2);
}

static void synchronize_rcu(void)
{
 for (;;) {
  if (__sync_fetch_and_xor(&rcu_read_nesting_global, 1)<2)
    return;
  SET_NOASSERT();
  return;
 }
}
```

**Fig. 15** Approximate model of RCU

ing for RCU. We use a pair of RCU-related benchmarks [48,49] 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 [46], 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 model approximating an RCU implementation for multiprocessor systems shown in Figure 15. In this model, the number of RCU read-side critical sections currently in flight is tracked by the global `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. Can falsification aid in these two complex, in-progress, verification efforts?

Our efforts are ongoing, due to the complexity of the targeted code (even with support from the primary developer, a co-author of this paper). At this time the investigation of mutants has already provided valuable information about these verifications benchmarks. First, there are two versions of the Tiny RCU verification. The earliest, very preliminary version, kills only 10 of 169 Tiny RCU mutants. Adding code to the harness to account for interrupts in the dyntick-idle handling kills an additional 12 mutants, confirming that the modification increases the strength of the harness. More importantly, the modeling of concurrency in the harness has two versions, one using CBMC support for pthread mutex locks, the other using disabling of assertions to ignore executions that violate locking semantics. The native mutex version allows much faster verification, and catches the original, hand-constructed checks to ensure the harness can detect faults in

Tiny RCU. However, the native mutex version fails to kill any mutants, a fact we are currently investigating: without mutants, we would not have been aware of this possibly critical problem, which may be a CBMC bug (in the course of this paper's work, we have uncovered several CBMC bugs) or a harness flaw. In support of the verification, we also generated passing maximal-coverage executions for all mutants of the Tiny RCU code. For 97 of the mutants, there is no passing execution; in many cases, these are not killed: the mutant modifies the concurrency semantics so CBMC has no valid executions to analyze (potentially invalid in some cases, which must be investigated). For 79 mutants the maximal-coverage passing runs are currently being examined, to determine the best next steps in improving the Tiny RCU harness. For the second benchmark, we have computed mutant kills and find that the kill rates range between 40% and 46%. While these benchmarks are far from complete, and over-simplify the modeling process, they are already able to catch a substantial number of potential RCU usage errors. Again, we have produced passing runs for the surviving mutants to use in enhancing the process. The good news is that while the RCU verification is much more substantial than the above case studies, the time to analyze mutants is not prohibitive. No single model checking run for the Tiny RCU benchmark takes more than 40 seconds, and total runtime for all mutants in the usage benchmarks ranges from just over 12 seconds for a basic litmus test to less than 5 minutes for the most complex of the benchmarks. Our belief that analyzing all surviving mutants is plausible for code of this size and criticality is supported by our concurrent preliminary work on using mutants to analyze the effectiveness of `rcutorture`, which has improved `rcutorture` itself and (by doing so) exposed a previously undetected RCU bug.

### 4.4 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[18] in almost all cases, but Z3 also aims to be fast at producing satisfying assignments, not proving UNSAT [52], and our few runs with Z3 showed the same pattern.

Figure 16 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

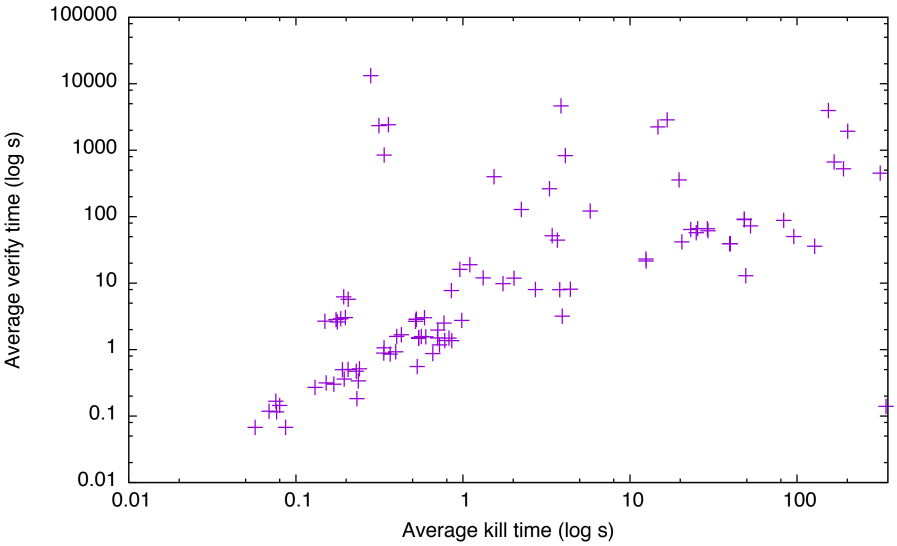


**Fig. 16** Average times for killing/verifying mutants, in seconds.

(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.

## 4.5 Automated Test Generation and Falsification

### *4.5.1* `rcutorture` *Case Study*

Summary of paper [1].

### *4.5.2* `pyfakefs` *Case Study*

Used TSTL [30,31,35,32]

Found bugs:

1. `https://github.com/jmcgeheeiv/pyfakefs/issues/306`
2. `https://github.com/jmcgeheeiv/pyfakefs/issues/307`
3. `https://github.com/jmcgeheeiv/pyfakefs/issues/308`
4. `https://github.com/jmcgeheeiv/pyfakefs/issues/309`
5. `https://github.com/jmcgeheeiv/pyfakefs/issues/310`
6. `https://github.com/jmcgeheeiv/pyfakefs/issues/311`

## 5 Discussion

### 5.1 Falsification vs. Verification

> "Those among us who are unwilling to expose their ideas to the hazard of refutation do not take part in the scientific game."
> -Popper, *The Logic of Scientific Discovery* [56]

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 exactly what has been 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 [56].

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.

Furthermore, many of the theoretical objections to Popper (e.g., such as that we "cannot learn from experience the falsehood of any theory" [42]) do not hold for software correctness problems: we can clearly establish the falsehood of a "theory" in our context by a single counterexample; it is only establishing the truth of a theory, and the value of that truth, that is difficult for us.

### 5.2 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" nor 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 [34, 26] 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.

## 6 Related Work

The idea that a "successful verification" in model checking (or even theorem proving) often simply indicates an inadequate property is long-standing [13, 37]. Use of mutants [7, 44] to provide a coverage measure dates back both to these early explorations and relatively recent work [41, 4, 12]. However, in these efforts the mutation was usually applied to hardware models, and (critically) the surviving mutants were used to, e.g., 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. [11]. 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 [5]. Previous work on the topic [5] 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" [22] is fundamental to all efforts to improve software quality. Recent efforts of most interest have focused on measuring *checked coverage* [58, 59, 53], where a metric tries to make sure the code under test potentially changes the value of an assert, using dynamic slicing [66, 62]. This is weaker than requiring the oracle kill a mutant, our goal, but more manageable for testing,

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

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).

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 [21], 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.

## 7 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. A verification is considered effective when it finds no faults in the SUT and detects every faulty variation of the SUT. An obvious source of faulty SUT variations 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 additional developer effort. Given these faulty 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). Developers can also check that a verification harness does not have any mutants that 1) verify the SUT while 2) killing more mutants than the original 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 but realistic verification efforts, and can contribute to serious systems verification and modeling efforts for complex code such as the Linux kernel RCU implementations. The bigger picture is that our approach attempts to apply the ideas of Karl Popper's falsification-centered approach to the philosophy of science to the understanding of software systems. In this view, verification is almost always provisional, but we can gain considerable confidence in a verification by making serious attempts to prove its inadequacy.

In future work we plan to continue to apply this falsification-driven approach to the RCU verification, and to other critical systems-software targets, which we expect will lead to discovery of new ways a model checker's ability to ask "what if" questions about program behavior [21, 29] can improve developer understanding of verification efforts. We would also like to integrate falsification-driven verification support into the CBMC Eclipse tools, and use speculative model checking calls and incremental SAT to make mutant analysis available to developers continuously as part of their development/debugging process. Finally, these techniques should also be applicable to verification using, e.g., Java Pathfinder [64] (at least in symbolic mode [55]; in pure explicit-state exploration the problems of non-exhaustive exploration may dominate).

We would also like to integrate falsification-driven verification support into the CBMC Eclipse tools, and use speculative model checking calls and incremental SAT to make mutant analysis available to developers continuously as part of their development/debugging process. Finally, these techniques should also be applicable to verification using, e.g., Java Pathfinder [64] (at least in symbolic mode; in pure explicit-state exploration the problems of non-exhaustive exploration may dominate).

A portion of this work was funded by NSF grants CCF-1217824 and CCF-1054786.

## References


1. Ahmed, I., Jensen, C., Groce, A., McKenney, P.E.: Applying mutation analysis on kernel test suites: an experience report. In: International Workshop on Mutation Analysis, pp. 110–115 (2017)
2. Alipour, M.A., Groce, A., Zhang, C., Sanadaji, A., Caushik, G.: 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)
3. Andrews, J.H., Briand, L.C., Labiche, Y.: Is mutation an appropriate tool for testing experiments? In: International Conference on Software Engineering, pp. 402–411 (2005)
4. Auerbach, G., Copty, F., Paruthi, V.: Formal verification of arbiters using property strengthening and underapproximations. In: Formal Methods in Computer-Aided Design, pp. 21–24 (2010)
5. Ball, T., Kupferman, O., Yorsh, G.: Abstraction for falsification. In: Computer Aided Verification, pp. 67–81 (2005)
6. Bentley, J.: Programming pearls: Writing correct programs. Communications of the ACM **26**(12), 1040–1045 (1983)
7. Black, P.E., Okun, V., Yesha, Y.: Mutation of model checker specifications for test generation and evaluation. In: Mutation 2000, pp. 14–20 (2000)
8. Bloch, J.: 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` (2006)
9. Budd, T.A., Lipton, R.J., DeMillo, R.A., Sayward, F.G.: Mutation analysis. Yale University, Department of Computer Science (1979)
10. Chen, Y., Groce, A., Zhang, C., Wong, W.K., Fern, X., Eide, E., Regehr, J.: Taming compiler fuzzers. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 197–208 (2013)
11. Chockler, H., Gurfinkel, A., Strichman, O.: Beyond vacuity: Towards the strongest passing formula. In: Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design, pp. 24:1–24:8 (2008)
12. Chockler, H., Kroening, D., Purandare, M.: Computing mutation coverage in interpolation-based model checking. IEEE Trans. on CAD of Integrated Circuits and Systems **31**(5), 765–778 (2012)
13. Chockler, H., Kupferman, O., Kurshan, R.P., Vardi, M.Y.: A practical approach to coverage in model checking. In: Computer Aided Verification, pp. 66–78 (2001)
14. Clarke, E., Grumberg, O., McMillan, K., Zhao, X.: Efficient generation of counterexamples and witnesses in symbolic model checking. In: Design Automation Conference, pp. 427–432 (1995)
15. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press (2000)
16. Cuoq, P., Monate, B., Pacalet, A., Prevosto, V., Regehr, J., Yakobowski, B., Yang, X.: Testing static analyzers with randomly generated programs. In: NASA Formal Methods Symposium, pp. 120–125 (2012)

17. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Englewood Cliffs, New Jersey (1976)
18. Een, N., Sorensson, N.: An extensible SAT-solver. In: Symposium on the Theory and Applications of Satisfiability Testing (SAT), pp. 502–518 (2003)
19. Gligoric, M., Groce, A., Zhang, C., Sharma, R., Alipour, A., Marinov, D.: Comparing non-adequate test suites using coverage criteria. In: International Symposium on Software Testing and Analysis, pp. 302–313 (2013)
20. Gligoric, M., Groce, A., Zhang, C., Sharma, R., Alipour, A., Marinov, D.: Guidelines for coverage-based comparisons of non-adequate test suites. ACM Transactions on Software Engineering and Methodology (accepted for publication)
21. Groce, A.: Error explanation with distance metrics. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 108–122 (2004)
22. Groce, A.: (Quickly) testing the tester via path coverage. In: Workshop on Dynamic Analysis (2009)
23. Groce, A., Ahmed, I., Jensen, C., McKenney, P.E.: How verified is my code? falsification-driven verification. In: 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015, pp. 737–748 (2015). DOI 10.1109/ASE.2015.40
24. Groce, A., Alipour, M.A., Gopinath, R.: Coverage and its discontents. In: Onward! Essays, pp. 255–268 (2014)
25. Groce, A., Erwig, M.: Finding common ground: choose, assert, and assume. In: Workshop on Dynamic Analysis, pp. 12–17 (2012)
26. Groce, A., Holzmann, G., Joshi, R.: Randomized differential testing as a prelude to formal verification. In: International Conference on Software Engineering, pp. 621–631 (2007)
27. Groce, A., Joshi, R.: Exploiting traces in program analysis. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 379–393 (2006)
28. Groce, A., Joshi, R.: Random testing and model checking: Building a common framework for nondeterministic exploration. In: Workshop on Dynamic Analysis, pp. 22–28 (2008)
29. Groce, A., Kroening, D.: Making the most of BMC counterexamples. Electron. Notes Theor. Comput. Sci. **119**(2), 67–81 (2005)
30. Groce, A., Pinto, J.: A little language for testing. In: NASA Formal Methods Symposium, pp. 204–218 (2015)
31. Groce, A., Pinto, J., Azimi, P., Mittal, P.: TSTL: a language and tool for testing (demo). In: ACM International Symposium on Software Testing and Analysis, pp. 414–417 (2015)
32. Groce, A., Pinto, J., Azimi, P., Mittal, P., Holmes, J., Kellar, K.: TSTL: the template scripting testing language. `https://github.com/agroce/tstl` (2015)
33. Guniguntala, D., McKenney, P.E., Triplett, J., Walpole, J.: The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with Linux. IBM Systems Journal **47**(2), 221–236 (2008)
34. Hamlet, R.: When only random testing will do. In: International Workshop on Random Testing, pp. 1–9 (2006)
35. Holmes, J., Groce, A., Pinto, J., Mittal, P., Azimi, P., Kellar, K., O'Brien, J.: TSTL: the template scripting testing language. International Journal on Software Tools for Technology Transfer (2016). DOI \url{https://doi.org/10.1007/s10009-016-0445-y}. Online first
36. Horspool, R.N.: Practical fast searching in strings. Software - Practice & Experience **10**(6), 501–506 (1980)
37. Hoskote, Y., Kam, T., Ho, P.H., Zhao, X.: Coverage estimation for symbolic model checking. In: ACM/IEEE Design Automation Conference, pp. 300–305 (1999)
38. Just, R., Jalali, D., Inozemtseva, L., Ernst, M.D., Holmes, R., Fraser, G.: Are mutants a valid substitute for real faults in software testing? In: ACM SIGSOFT Symposium on Foundations of Software Engineering, pp. 654–665 (2014)
39. Kroening, D., Clarke, E.M., Lerda, F.: A tool for checking ANSI-C programs. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 168–176 (2004)
40. Kroening, D., Strichman, O.: Efficient computation of recurrence diameters. In: Verification, Model Checking, and Abstract Interpretation, pp. 298–309 (2003)
41. Kupferman, O., Li, W., Seshia, S.: A theory of mutations with applications to vacuity, coverage, and fault tolerance. In: Formal Methods in Computer-Aided Design, pp. 1–9 (2008)
42. Lakatos, I.: The role of crucial experiments in science. Studies in History and Philosophy of Science Part A **4**(4), 309–325 (1974)

43. Lawlor, R.: quicksort.c. http://www.comp.dit.ie/rlawlor/Alg_DS/sorting/quickSort.c. Referenced April 20, 2015
44. Lee, T.C., Hsiung, P.A.: Mutation coverage estimation for model checking. In: Automated Technology for Verification and Analysis, pp. 354–368 (2004)
45. Lipton, R.J.: Fault diagnosis of computer programs. Tech. rep., Carnegie Mellon Univ. (1971)
46. McKenney, P.E.: Re: [PATCH fyi] RCU: the bloatwatch edition (2009). Available: http://lkml.org/lkml/2009/1/14/449 [Viewed January 15, 2009]
47. McKenney, P.E.: Structured deferral: synchronization via procrastination. Commun. ACM **56**(7), 40–49 (2013). DOI 10.1145/2483852.2483867. URL http://doi.acm.org/10.1145/2483852.2483867
48. McKenney, P.E.: Verification challenge 4: Tiny RCU. http://paulmck.livejournal.com/39343.html (2015)
49. McKenney, P.E.: Verification challenge 5: Uses of RCU. http://paulmck.livejournal.com/39793.html (2015)
50. McKenney, P.E., Eggemann, D., Randhawa, R.: Improving energy efficiency on asymmetric multiprocessing systems (2013). https://www.usenix.org/system/files/hotpar13-poster8-mckenney.pdf
51. McKenney, P.E., Slingwine, J.D.: Read-copy update: Using execution history to solve concurrency problems. In: Parallel and Distributed Computing and Systems, pp. 509–518. Las Vegas, NV (1998)
52. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 337–340 (2008)
53. Murugesan, A., Whalen, M.W., Rungta, N., Tkachuk, O., Person, S., Heimdahl, M.P.E., You, D.: Are we there yet? determining the adequacy of formalized requirements and test suites. In: NASA Formal Methods Symposium, pp. 279–294 (2015)
54. Papadakis, M., Jia, Y., Harman, M., Traon, Y.L.: 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)
55. Pasareanu, C.S., Visser, W., Bushnell, D.H., Geldenhuys, J., Mehlitz, P.C., Rungta, N.: Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis. Automated Software Engineering **20**(3), 391–425 (2013)
56. Popper, K.: The Logic of Scientific Discovery. Hutchinson (1959)
57. Popper, K.: Conjectures and Refutations: The Growth of Scientific Knowledge (1963)
58. Schuler, D., Zeller, A.: Assessing oracle quality with checked coverage. In: International Conference on Software Testing, Verification and Validation, pp. 90–99 (2011)
59. Schuler, D., Zeller, A.: Checked coverage: an indicator for oracle quality. Software Testing, Verification, and Reliability **23**(7), 531–551 (2013)
60. scvalex: Finding all paths of minimum length to a node using dijkstras algorithm. https://compprog.wordpress.com/2008/01/17/finding-all-paths-of-minimum-length-to-a-node-using-dijkstras-algorithm/ (2008)
61. Stout, R.: If Death Ever Slept. Viking (1957)
62. Tip, F.: A survey of program slicing techniques. Journal of programming languages **3**, 121–189 (1995)
63. visar: [SOLVED] doubly linked list insertion sort in C. http://www.linuxquestions.org/questions/programming-9/doubly-linked-list-insertion-sort-in-c-4175415860/ (2012)
64. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. Automated Software Engineering **10**(2), 203–232 (2003)
65. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in C compilers. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 283–294 (2011)
66. Zhang, X., Gupta, R., Zhang, Y.: Precise dynamic slicing algorithms. In: International Conference on Software Engineering, pp. 319–329 (2003)