How Verified is My Code? Understanding "Successful" Verifications

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) 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.

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

```
#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 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], 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 [9], [10] 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 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 (that include the mutant). 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.

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

Fig. 2. Quick sort code.

A. 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" [11], 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?

1) 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 [12] 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 [13], 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;

[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`.

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 5, 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?

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 # 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 [14]. 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]`?

```
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.

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!

B. Contributions

Our primary contribution in this paper is a detailed examination of the extension of traditional mutation testing to understand successful (and "successful") verification results, and determine when a harness is not actually sufficiently powerful to ensure correctness. To support this approach, we show how to use mutation testing to choose an unwinding depth for loops 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 of 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 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. Falsification and Verification

III. Algorithms and Techniques

Given a mutant of program P, M_i, the

[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.

```
(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] == KILLED:
            //once killed, assume always killed
            M = M \ m
      if r[m] == SURVIVED:
         r'[m] = check(H,m,U(S+1)+1,O(S+1))
         if r'[m] == KILLED:
            M = M \ m
            changed = True
            goto TOP
// No result changed, so S is mutant-stable
return (S-1, M)
```

Fig. 5. Algorithm 1: Finding size/unwinding and surviving mutants

```
harness covering(H, TARGET)

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

Fig. 6. Algorithm 2: Convert harness into maximal coverage search

Even a killed mutant (e.g., a mutant the harness detects) can shed critical light on harness vulnerabilities. For example, the code in Figure 7 is a portion of a harness to verify code that merges two sorted arrays, removing all duplicates (the source arrays may contain duplciates 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. However, as is well known, many software faults [15] 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 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 lets us see the problem.

IV. Experimental Results and Case Studies

V. Related Work

The idea that a "successful verification" often simply indicates an inadequate property is long-standing [8] and the use of mutants to provide a coverage measure dates back both to these early explorations and relatively recent work [16]. However, in these efforts the mutation was applied to hardware

```
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. 7. 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. 8. Code to merge two sorted arrays into one sorted array with no duplicate elements

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.

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 [17], 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.

VI. Conclusion

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] 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.

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

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

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

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

[13] 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.

[14] 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.

[15] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2014, pp. 654–665.

[16] 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.

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