(Programs), Proofs and Refutations (and Tests and Mutants)

Alex Groce
Northern Arizona University
United States

Abstract

This essay consists of an imaginary discussion among a group of students after a computer science class, that presents some problems of (and partial solutions to) fundamental issues of program correctness.

CCS Concepts: • **Software and its engineering** → **Dynamic analysis**; **Software testing and debugging**.

Keywords: software testing software verificaton, proof, counterexample, tests

ACM Reference Format:
Alex Groce. 2021. (Programs), Proofs and Refutations (and Tests and Mutants). In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/1122445.1122456

1 The Dialogue

The dialogue takes place in a classroom. The last class of the day has finished, and the professor is packing up laptop, display-adapter, and mouse, and even remembering to turn off the projection screen, as a few students remain sitting near each other. The lingering students have become interested in a **PROBLEM** (or perhaps simply caught up in an argument).

(let's listen in)

PUPIL SIGMA: It just seems trivial to me. I think the code here is simple enough that you can simply inspect it and see that it does what it should do. The idea of binary search is slightly tricky to understand the very first time you see it, but once you understand how it works, writing code to "do it" does not require a PhD.

Woodstock '18, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/10.1145/1122445.1122456

PUPIL BETA: I'm not claiming it takes a PhD, or saying a PhD would help! You saw how Dr. Omega messed it up the first time on the whiteboard.

PUPIL DELTA: *(sotto voce)* That's because Dr. Omega is a bit of an absentminded flake.

PUPIL SIGMA: Okay, that's true. But Alpha noticed it was wrong and once she explained why, we all understood. But I want to say something stronger than just that we all know the code is correct now. Look at the code...

(let's look at that code ourselves (Figure 1))

PUPIL SIGMA: Now that we fixed that one problem, and we all understand the basic idea of binary search, there's no use for anything more formal or complicated. There's so little room for bugs here that any possible problems would be revealed as soon as we used the code. Think about the professor's mistake. As soon as we started using the code, it'd fail to find something present in the array.

PUPIL DELTA: Are you sure? I wonder how often it fails. I'll grant that if the code goes wrong often enough, someone would notice, but you seem to be suggesting that even without tests, we'd notice very quickly. But what if we almost always search for things not in the array? Or even if we search for things present most of the time, isn't it only going to show up when the item is where low and high meet, and might that not be really rare? Especially if the array is very big! I'd want thorough tests, and even for this code I'm not sure how to make them!

PUPIL GAMMA: If you varied the size of the array, in your tests, that should help. At size 1, this bug shows up every time the element is present!

PUPIL ALPHA: Oh that's nice. I bet that's a good idea, to test systems that can vary their size on really small versions. You can probably test really thoroughly at small sizes, too. If an array only has one element, I think you've fully tested it if you just check the case where the element is the one you're searching for and the element is not the one you're searching for!

PUPIL DELTA: Anyway, I think the real problem isn't how to test binary search. I mean, that's fine, and I'm sure there could even be computer scientists who just think about silly trivial thngs like how to write better tests that find more bugs. But don't you think there's something deeper going on here, something that's a *real* problem?

```
int binsearch(int* a, int key, int size) {
  int low = 0;
  int high = size - 1;
  while (low <= high) {  ON THIS LINE THE "=" HAS CLEARLY BEEN ADDED LATER
    int mid = (low + high) / 2;
    int midVal = a[mid];
    if (midVal < key)
      low = mid + 1;
    else if (midVal > key)
      high = mid - 1;
    else {
      return mid; // key found
    }
  }
  return -1;  // key not found.
}
```

Figure 1. C Code on the Whiteboard at the End of Class

PUPIL GAMMA: You're just jealous I thought of the one-element array thing.
PUPIL BETA: No, that's not it. All philosophy-CS double majors are like this, all the time. It's annoying. Anyway, let's hear about that "real" problem.
PUPIL DELTA: The real problem is, imagine that we have a set of perfect, absolutely thorough tests for binary search, or whatever we're testing, let's call it program $\mathcal{P}$. Or, better yet, we have a complete proof of correctness of $\mathcal{P}$. Now, those tests or that proof are going to be *with respect to a specification of what $\mathcal{P}$ should do.* Let's call that specification $\mathcal{S}$. Ok, we've demonstrated to everyone's satisfaction that $\mathcal{P}$ satisfies $\mathcal{S}$. I claim we're not much better off than in the case where we just trust Sigma's intuition that $\mathcal{P}$ is "obviously right."
PUPIL BETA: How so?
PUPIL DELTA: Because, we've just shoved back the problem. We were going to trust $\mathcal{P}$, or maybe trust Sigma. Now we're just changing that to trusting $\mathcal{S}$! That seems as bad as trusting Sigma!
PUPIL SIGMA: Hey! What's so bad about trusting me!
PUPIL DELTA: Sorry. But seriously, we've just changed the thing we have to place arbitrary trust in. Even if our proofs or tests are good, how do we know $\mathcal{S}$ is good?
PUPIL ALPHA: $\mathcal{S}$ might be a lot simpler than $\mathcal{P}$.
PUPIL DELTA: Ok, I can imagine that might probabilistically give us some more *confidence*. I don't want to turn this into the general problem of epistemology, but I think there is a practical problem here. In the case of binary search, I think there is a small amount of simplification from $\mathcal{P}$ to $\mathcal{S}$. And a small gain. But both are fairly trivial. It really is probably almost as good to just trust Sigma as to trust $\mathcal{S}$. Sigma made a perfect score on the midterm, after all!

(Delta walks to the whiteboard and points to the code.)

PUPIL DELTA: And maybe trusting Sigma or $\mathcal{S}$ is fine in this case. But if $\mathcal{P}$ is something complicated, just the specification $\mathcal{S}$ is going to be extraordinarily complex, much more complex than the $\mathcal{P}$ for something like binary search. If we don't trust $\mathcal{P}$ for binary search, why on earth would we ever trust something as complicated as $\mathcal{S}$ for a real problem? What's an operating system's $\mathcal{S}$?
PUPIL GAMMA: In the real world, I think this ends up being a social process, really. I mean, you're right that it's about trust, but not trusting just Sigma. How do people come up with the specification for an operating system, and decide if it's any good? A group of experts, I'd guess, work on it. They toss ideas back and forth, they look for counterexamples to proposals about how the system should work. They talk to test engineers, developers, users.
PUPIL DELTA: You know, that sounds like what really happens. Trusting Sigma alone is no good, but a bunch of Sigmas... I don't konw if that's ideal, but it's not nearly as bad. And in practice, it's what we have to rely on. Even mathematical "proofs" are just things enough mathematicians agree to call proofs. It happens that in math, unlike in philosophy, there's pretty frequent agreement on whether a proof *is* a proof, but I've read enough history of the field to know there are also cases where it took a lot of argument and discussion to settle on the definitions and right form of proof. I think Euler's simple theorem about regular polyhedra was one, even.
PUPIL EPSILON: I'm not sure I like this being a matter of trusting a social process at all! In mathematics, there *is* some way for experts to check each other, and to be honest, the stakes aren't as high as with a self-driving car or a mission to Mars. Groups of experts make mistakes all the time, especially if they are on the same "team" and subject to groupthink. One person's dominant personality should not drive a discussion of how to make a safe and reliable computer system!
PUPIL DELTA: Of course. But what else is there? After all, deliberation by the body of people who are in a position to make a decision, by a social process, is how we determine who runs the country, a jury of peers is how we decide if a person should go to jail or not, and so forth. In these cases, the people are far *less* expert. And more is usually at stake!

Science and math also, basically operate that way. That's peer review.

PUPIL EPSILON: Sure, but I think the issue is "what else is there?" It seems to me that computers offer us a chance to finally do better, in certain limited parts of life. Take proofs. Before computers, proofs were inevitably checked by other mathematicians. The idea of reducing the steps in proofs to such simple ones that other people could check the proofs exactly was, I think, tossed around, Leibniz and those people. But it couldn't work, because nobody has the patience or attention to detail. But before computers, nobody also had the patience or attention to detail to produce perfect logarithm tables. Babbage's dream in part was to replace legions of inattentive country clergy computing logarithms poorly with a machine pumping them out perfectly. To an extent he woudln't have though possible, we have that now, and not just for arithmetic problems, but with symbolic math tools. There are people who work on automated theorem provers, and while those aren't replacing mathematicians or checking proofs like Fermat's last theorem yet, I think there are people who have the basics of automatic *proof checking*.

PUPIL DELTA: Fine, but that's *not* relevant to my problem. That's not checking $\mathcal{S}$. It's checking the proof $\mathcal{P}$. Hmm, we called the program $\mathcal{P}$. The proof $\mathcal{Q}$ for "QED".

PUPIL EPSILON: Yes, there's nothing for your problem right now. That's why we have so many buggy specifications, not just buggy programs. But I think we should dream big, like Leibniz. Sure, his methods were fantastical at the time, so were Babbage's. We can dream about how to address the $\mathcal{S}$ problem, too!

PUPIL BETA: Frankly, I don't think this problem of knowing if you've proved or tested the right specification is quite as hard as you all seem to think. I grant that *sometimes* it is, but I think often we have a program like this one, where it's easy. Not because, as Sigma originally suggested, binary search is so trivial, but because binary search is *equivalent to something that really is trivial.*

PUPIL GAMMA: What do you mean?

PUPIL BETA: I mean that the right specification for binary search is very simple: binary search works just like linear search, only faster. Ok, maybe the "faster" part is not so easy, but in either a proof or a test, we just need to compare the "tricky" binary search results to the result for linear search. Linear search is so simple I defy anyone, even the worst student in this class, to get it seriously wrong!

PUPIL EPSILON: That's a really good idea!

PUPIL GAMMA: Except it doesn't work. What if you are using binary search on an array with duplicates? The result won't always be the same as for linear search then.

PUPIL BETA: Oh, fine, we can just say that if both return that the value is found, but they report different positions, we'll check if the right value is present in both positions.

PUPIL GAMMA: But then you are only using the linear search to check "not found" results, and I imagine you could just set up tests to know whether they are using a value in the array or not. And for proofs, I think proving equivalence to linear search for "not found" might be harder than just proving the right answer is provided, since it's *not* actually equivalent to linear search!

PUPIL EPSILON: Ok, maybe Beta hasn't saved us much work here, where it's not quite equivalent and "fixing" the mismatch doesn't seem much easier than just figuring out exactly what binary search should do. But I bet this is a good idea for the kinds of programs we were mentioning above, where a person understanding fully what a program ought to do seems so hard it's almost impossible. Think about either a really abstract file system that doesn't have hardware problems or efficiency issues, or a compiler that doesn't do any optimizations, and maybe produces very slow, but simple, binary code. I don't know about proofs, but for testing at least, comparing complex versions that have to be fast and practical for real-world use to much simpler implementations that you couldn't use in real life... That seems like a very nice way to test some things.

PUPIL DELTA: Yes, yes, that sounds practical. But it doesn't really address the underlying problem at all. It's a cheap hack that's sometimes available. How do we know the "simple" file system or compiler's "idea of what it should do" is right? How do we trust any specification that's too complicated to fit inside a person's head. Or, really, like we said, to fit inside multiple heads at once so a group of reliable people can all agree that they are all thinking of the same thing, and that thing is the right thing. You're just pushing the problem back one step, in a small set of cases, and the same issue really comes up for the simple verison, if the problem at hand is at all complicated, like your examples.

PUPIL EPSILON: It's still a partial solution, like I suggested. Somteimes this will make the $\mathcal{S}$-problem easier. That's a step in the right direction, and I'd rather trust that a simple, non-optimizing, C compiler is right than trust that `gcc -O3` does the right thing!

PUPIL BETA: Alpha, you've been awful quiet, that's not like you. Is something wrong?

PUPIL ALPHA: No, I'm just thinking about the code. Which the rest of you seem to have abandoned.

PUPIL GAMMA: One thing that worries me is that we're talking as if the program $\mathcal{P}$ is just finished, once and for all, and then we test it or prove it and when we're satisfied we call it a day. Maybe that's true for a very small program like binary search in a library, or as a homework assignment (if you are crazy and prove your homework assignments, or even bother testing them). But in the real world, isn't code modified and changed all the time? I feel like a program in reality is either thrown away quickly (in which case who cares if it's right?) or lives to be changed, maybe even eventually having no lines of code in common with the original program.

PUPIL DELTA: Like the ship of Theseus!

PUPIL BETA: Show-off...
PUPIL DELTA: Seriously, it seems to me this is a possible reason tests are even better than proofs! Or at least useful even when we have a proof. Imagine we change the code for some reason. If we have really good tests, it's easy to see if we made a mistake: we just run those tests! But your proof isn't something you can run. You have to look at the proof and the changed code, and think about whether the change breaks the proof. Another chance for human mistakes!
PUPIL EPSILON: There are automated proofs, like I said, where a computer produces the proof, or at least checks that it's correct.
PUPIL DELTA: Sure, but I think the tools for generating proofs without human assistance are not that great right now. That might change, but right now it's true. And the checkers don't seem that helpful here: I bet when you change your program, the proof-checker just always says "your proof no longer works."
PUPIL EPSILON: In general, I think you're right. But there are some limited tools that are almost fully automatic I think. The Turing Award in 2007 was given for a technique called model checking, that is fairly limited in application, but really does, sort of, compute the proof automatically. We talked about it in automata theory class.
PUPIL DELTA: This does suggest a way to think abstractly, though not practically, about how good a test, or proof... or maybe even an $\mathcal{S}$ is? We want to know *how many bugs does it catch?* Of course, that's sort of useless. We don't know all the bugs there might be, or I guess we'd just go down that list and check for each of them. The list of all bugs is just a kind of "super-$\mathcal{S}$". But it can get us a little beyond just $\mathcal{S}$, in that if we have a program, and we release it to users, and they complain about some awful behavior, even if we failed to put "no doing that!" in $\mathcal{S}$, we can easily agree it's bad, and revise $\mathcal{S}$. That bad behavior is in the "set of all bugs" at least in theory.
PUPIL EPSILON: Yes, but I'm looking for practical solutions, ways computers could help us out.
PUPIL DELTA: Sometimes thinking about the problem definition in the most abstract terms can be practical, you know. That's part of the lesson of philosophy.
PUPIL BETA: Oh brother.
PUPIL EPSILON: No, wait. I think you have something there. Imagine a "bugginator" – a computer program that takes another computer program, and changes it to all possible buggy versions. Now, if we have a bugginator, the opposite of a "debugger" if a debugger did what it says on the tin, I guess, then we could get somewhere. We could just run the bugginator, and run all our tests and check all our proofs. Every bug that isn't detected either shows a problem with our tests, or with our proofs, OR with $\mathcal{S}$. There would be some work in figuring out which one, of course, maybe hard work. But it'd be concrete and practical. I mean, this is basically what we do in Delta's scenario of customers reporting problems or a Mars probe mysteriously crashing. We do a post-mortem and if the problem really is a software problem, we end up changing the program, but if we're diligent engineers we obviously fix up our tests, that's what regression testing is, and nobody can argue that's some impractical thing that nobody in industry can use.
PUPIL GAMMA: One little problem, again. We don't have a bugginator. How can you make a bugginator?
PUPIL EPSILON: Ok, you can't. But perhaps you could make a partial bugginator. Inject *some* bugs automatically. I guess the lousy version would be to hire people to inject bugs. Or maybe an LLM can come up with good bugs?
PUPIL DELTA: I don't know how you'd get a good sample, and I don't trust LLMs not to just make up the kinds of bugs engineers already think about. You'd want something principled, that comes up with bug scenarios nobody normal will ever think of, the way the fuzzers we saw in the software security class come up with program inputs nobody considers.
PUPIL EPSILON: That's it! Those fuzzers like AFL are called mutation-based fuzzers because their basic loop is taking some input, and changing it a little bit to see what the program does with the mutated input. So we get bugs for the bugginator by making small random changes to $\mathcal{P}$. You could call them "mutants" to distinguish them from the set of all bugs, these are a sample that's biased to things that are "very close" to the program we're interested in, but it seems reasonable that most programs are fairly close to correct, so the making small changes should make the program "almost" correct but not quite.
PUPIL GAMMA: Not bad, not bad. The bug the Dr. Omega made is very close to the correct version of binary search. So you'd surely include that kind of thing among your "mutants." You don't even need anything smart here, you could just write some dumb thing with regular expressions to change arithmetic operators, or comparisons, or invert `if` conditions.
PUPIL EPSILON: I bet just commenting out random lines of code could get you somewhere.
PUPIL GAMMA: Let's go over to my dorm room and write one of these. I'd like to see how good my tests and my $\mathcal{S}$ are for the operating systems project.

(All but one of the students walk out of the classroom, Beta and Delta holding hands (they are dating), Epsilon and Gamma discussing whether to write their bugginator in Python or Rust, Sigma talking on the phone, asking another student about hitting the gym.)

PUPIL ALPHA: Guys? Guys? I think the code is still wrong. What happens if the size of `a` is so big that `high` plus `low` overflows? I don't think your bugginator would catch that problem with the proof or the tests or the specification. The

```
#define MAX_SIZE 10
int main () {
  int a[MAX_SIZE];
  int SIZE = nondet_int();
  __CPROVER_assume(SIZE > 0);
  __CPROVER_assume(SIZE <= MAX_SIZE);
  int k = nondet_int();
  int present = 0;
  for (int i = 0; i < SIZE; i++) {
    a[i] = nondet_int();
    if (i > 0) {
      __CPROVER_assume(a[i] >= a[i-1]);
    }
    if (a[i] == k) {
      present = 1;
    }
  }
  int r = binsearch(a, k, SIZE);
  if (r != -1) {
    assert(a[r] == k);
  } else {
    assert(!present);
  }
  }
```

Figure 2. CBMC Proof Harness for Binary Search

```
#include <algorithm>
#include <deepstate/DeepState.hpp>
using namespace deepstate;
#define MAX_SIZE 32
TEST(Run, Bentley) {
  int a[MAX_SIZE];
  unsigned int SIZE = DeepState_UIntInRange(1, MAX_SIZE);
  int k = DeepState_Int();
  int present = 0;
  for (int i = 0; i < SIZE; i++) {
    a[i] = DeepState_Int();
    if (a[i] == k) {
      present = 1;
    }
  }
  std::sort(std::begin(a), &a[SIZE]);
  if (!present && DeepState_Bool()) {
    k = a[DeepState_UIntInRange(0, SIZE-1)];
    present = 1;
  }
  int r = binsearch(a, k, SIZE);
  if (r != -1) {
    assert(a[r] == k);
  } else {
    assert(!present);
  }
}
```

Figure 3. DeepState Test Harness for Binary Search

real problem is the specification and proof and tests all assume something about how big an array we want, and I think a proof might assume an `int` is an actual integer, not a computer int. This is a problem of the imagination. How do we get an imaginator?

2 Postscript

This dialogue is a tribute to, and reflection on, Imre Lakatos' classic work, *Proof and Refutation* [9]. It also owes a debt to MacKenzie's more recent classic, *Mechanizing Proof: Computing, Risk, and Trust* [10]. To some extent the approach to thinking about proofs, tests, and their meaning that (some of) the students arrive at is based on a Popperian [12] falsification methodology proposed in work by my colleagues and myself [6, 7]. Finally, the initial discussion owes a substantial debt to De Millo, Lipton, and Perlis' (in)famous "Social processes and proofs of theorems and programs" [3].

The version of binary search that starts things rolling (and the bug that Alpha notices at the end) comes from Joshua Bloch's blog post [2] reporting the bug, and, thus, fundamentally, from Jon Bentley's original version "proven correct" in *Programming Pearls* [1]. The code has been changed to C code. The programs in question, and instructions for proving and testing (and mutating [11]) them using CBMC [8] and DeepState [5] (and UniversalMutator [4]), respectively,, can be found at https://github.com/agroce/onward24code. Some brief notes on the connections between this code and the imaginary classroom discussion follow.

2.1 CBMC

CBMC serves as an exemplar for *proof*. CBMC translates C programs into goto-programs, and, eventually, into SAT or SMT constraints, such that a satisfying assignment represents a counterexample to the properties to be checked. A proof of unsatisfiability then is a proof of correctness for the program.

Strictly speaking, the proof can be partial: CBMC is a *bounded* model checker, and so requires the use of a bound on loop unrollings. However, in the case of binary search, a limit on unrollings of the search loop is part of the full specification of correctness, so the proof is complete (since CBMC can check that an execution exceeding provided loop bounds does not exist). The "harness" for CBMC is shown in Figure 2. The code for binary search is omitted (see Figure 1).

2.2 DeepState

DeepState exemplifies *tests*. While DeepState can make use of symbolic execution, which can more resemble proof, if is primarily used in conjunction with the more scalable approaches of random testing and coverage-guided fuzzing. The key idea is that while a CBMC output of "correct" indicates that the program input satisfies its specification (though not, as our students discuss, that the specification itself is correct), DeepState may run for days without finding a bug in an incorrect program. On the other hand, for some programs, DeepState will quickly find a bug, and CBMC will simply exhaust the memory of the computer it is running on, and the patience of the user, without accomplishing much of anything. How often this happens, vs. producing the bug quickly, however, is hard to know.

The "harness" for DeepState is shown in Figure 3. Again, the binary search code is omitted. Note the larger size, due to the fact we have to "help out" the testing rather than simply use ASSUME statements to force sorting and rely on exhaustiveness to check cases where the item to be searched for is, in fact, present.

2.3 UniversalMutator

Finally, the idea of mutation is represented by UniversalMutator, which mutates code in C and a number of other languages. UniversalMutator, like other mutation tools, acts as a limited version of Epsilon's hypothetical "bugginator."

References

[1] Jon Bentley. Programming pearls: Writing correct programs. *Communications of the ACM*, 26(12):1040–1045, 1983.

[2] Joshua Bloch. Extra, extra - read all about it: Nearly all binary searches and mergesorts are broken. https://blog.research.google/2006/06/extra-extra-read-all-about-it-nearly.html, 2006.

[3] Richard A De Millo, Richard J Lipton, and Alan J Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5):271–280, 1979.

[4] Sourav Deb, Kush Jain, Rijnard van Tonder, Claire Le Goues, and Alex Groce. Syntax is all you need: A universal-language approach to mutant generation. In *ACM International Conference on the Foundations of Software Engineering*, 2024.

[5] Peter Goodman and Alex Groce. DeepState: Symbolic unit testing for C and C++. In *NDSS Workshop on Binary Analysis Research*, 2018.

[6] Alex Groce, Iftekhar Ahmed, Carlos Jensen, and Paul E McKenney. How verified is my code? falsification-driven verification (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 737–748. IEEE, 2015.

[7] Alex Groce, Iftekhar Ahmed, Carlos Jensen, Paul E McKenney, and Josie Holmes. How verified (or tested) is my code? falsification-driven verification and testing. *Automated Software Engineering*, 25(4):917–960, 2018.

[8] Daniel Kroening, Edmund M. Clarke, and Flavio Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176, 2004.

[9] Imre Lakatos. *Proofs and refutations*. Nelson London, 1963.

[10] Donald MacKenzie. *Mechanizing proof: computing, risk, and trust*. MIT Press, 2004.

[11] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Mutation testing advances: an analysis and survey. In *Advances in Computers*, volume 112, pages 275–378. Elsevier, 2019.

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