15-414: Bug Catching: Automated Program Verification

Lecture Notes on Certificates

Matt Fredrikson

Carnegie Mellon University
Lecture 14
Tuesday, October 14, 2025

1 Introduction

Full static verification tries to prove the entire solver correct. This gives a sweeping guarantee, as long as we're willing to trust that the verification toolchain is correct. On the other hand, it can be very challenging and costly to perform full verification as the target functionality gets more complex.

A different approach is to write algorithms that produce evidence that their results are correct. This evidence, a *certificate*, is returned along with the result and can be checked at runtime to ensure correctness. Here we treat the primary computation, which we'll call the *solver*, as untrusted because we won't attempt to verify that it is correct. Instead, we will target verifying the certificate checker, under the presumption that it is less complex and therefore less costly to do so.

If the checker accepts, we learn the claimed property is *true for this particular input*. If it rejects, then the solver's results cannot be trusted and the client code must either fail or have a fallback to obtain the result by other means. The advantage is that we get verified correctness whenever certificates are valid, and we surface the fact that a computation has a bug otherwise. When the solver's task can be decomposed well in this way, we also hopefully have a significantly easier task ahead of us in verifying the checker.

Big picture: certificate checking gives us a practical path to trustworthy automation. We (1) write precise checker contracts, (2) prove small lemmas that connect a successful check to the desired property, and (3) let Why3 discharge the VCs for these simple components. We avoid verifying complex heuristics while still getting strong, instance-wise guarantees.

In this lecture we develop this approach on two examples in Why3:

- $\gcd(a, b)$ via a Bézout certificate.
- Whether a graph is bipartite, with certificates given by either a 2-coloring or an odd cycle.

Learning goals.

- Understand how Bezout's identity and the extended Euclidean algorithm yield easily verifiable certificates.
- Formalize and verify the correctness of certificates for graph bipartiteness, and see how these certificates can be generated with little additional overhead.
- Reduce verification for solvers from functional correctness to safety.
- Learn how fuel simplifies termination arguments, and how it relates to certificate checking.
- Understand the tradeoffs between static verification and certificate checking.

2 GCD via a Bézout Certificate

We use the classical observation: if $a, b \geq 0$ are not both 0 and integers x, y and $g > 0$ satisfy

$$g = ax + by, \qquad g \mid a, \qquad g \mid b,$$

then any common divisor d of a and b divides g (since d divides any linear combination of a and b). Hence $d \leq g$ and g is a greatest common divisor. We capture exactly this reasoning:

```
lemma lincomb_of_common_divisors:
  forall a b x y d:int.
    divides d a -> divides d b -> divides d (a*x + b*y)

lemma bezout_implies_gcd:
  forall a b g x y:int.
    0 < g -> g = a*x + b*y -> divides g a -> divides g b -> g = gcd a
        b
```

The checker for a purported (g, x, y) simply tests the identity and the two divisibility conditions and, if it returns `true`, the lemmas yield $g = \gcd(a, b)$.

```
let check_gcd (a b g x y : int) : bool
  requires { 0 <= a /\ 0 <= b }
  ensures  { result -> g = gcd a b }
= if g = 0 && a = 0 && b = 0 then true
  else 0 < g && g = a*x + b*y && mod a g = 0 && mod b g = 0
```

Extended Euclid (how the certificate is produced). Extended Euclid maintains, alongside the usual remainders, coefficients that witness each remainder as a linear combination of the inputs:

$$\texttt{aa} = ax_0 + by_0, \qquad \texttt{bb} = ax_1 + by_1.$$

Each step computes $q = \texttt{aa} \div \texttt{bb}$, $r = \texttt{aa} \bmod \texttt{bb}$ (so $\texttt{aa} = q\,\texttt{bb} + r$, $0 \le r < \texttt{bb}$), and updates

$$\texttt{aa} \leftarrow \texttt{bb},\ \texttt{bb} \leftarrow r,\ (x_0, y_0) \leftarrow (x_1, y_1),\ (x_1, y_1) \leftarrow (x_0 - qx_1,\ y_0 - qy_1).$$

This preserves the invariants because

$$r \ = \ \texttt{aa} - q\,\texttt{bb} \ = \ a(x_0 - qx_1) + b(y_0 - qy_1).$$

When $\texttt{bb} = 0$, we have $\texttt{aa} = \gcd(a, b)$ and $\texttt{aa} = ax_0 + by_0$, so (x_0, y_0) are Bézout coefficients.

```
(* Extended Euclid: returns (g, x, y) with g = gcd(a,b) = a*x + b*y *)
let euclid (a b : int) : (int, int, int)
  requires { 0 <= a /\ 0 <= b }
  ensures  { let (g,x,y) = result in 0 <= g /\ g = gcd a b /\ g = a*x
      + b*y }
=
  let ref aa = a in let ref bb = b in
  let ref x0 = 1 in let ref y0 = 0 in   (* aa = a*x0 + b*y0 *)
  let ref x1 = 0 in let ref y1 = 1 in   (* bb = a*x1 + b*y1 *)
  while bb <> 0 do
    invariant { 0 <= aa /\ 0 <= bb }
    invariant { aa = a*x0 + b*y0 /\ bb = a*x1 + b*y1 }
    variant   { bb }
    let q  = div aa bb in
    let r  = mod aa bb in
    let nx = x0 - q*x1 in
    let ny = y0 - q*y1 in
    aa <- bb; bb <- r;
    x0 <- x1; y0 <- y1;
    x1 <- nx; y1 <- ny
  done;
  (aa, x0, y0)
```

In some cases we may not need to expose the certificate to the client code, and it is instead more convenient to have a public function that interfaces the way that a non-certificate producing solver would. We can recover this interface, but we need some way of notifying clients when certificate checking failed. We can do this by raising an exception.

```
exception InvalidCertificate

let gcd_cert (a b : int) : int
  requires { 0 <= a /\ 0 <= b }
  ensures  { result = gcd a b }
```

```
  raises   { InvalidCertificate }
=
  let (g,x,y) = euclid a b in
  if not (check_gcd a b g x y) then raise InvalidCertificate else g
```

What have we achieved? We didn't verify the solver beyond showing that it terminates, and it never attempts to divide by zero. Nonetheless our public wrapper `gcd_cert` does have a postconndition which establishes exactly what we would have proved of the solver. This is a weaker property because it allows an exception that doesn't return a result. We can't even know if `gcd_cert` will *ever* return a result without testing it. But, we can rest assured that when it does return, the result will be correct.

3 Bipartite Graphs

A graph $G = (V, E)$ is *bipartite* if we can split V into disjoint parts L and R that cover V and such that every edge crosses sides: for $(u, v) \in E$, we have $u \in L \Leftrightarrow v \in R$. Equivalently, G is 2-colorable. This shows up in practice in scheduling (tasks vs. machines), constraint systems that must alternate types, and core algorithms (e.g. matchings and flows are simplest on bipartite graphs).

```
predicate proper_bipartition (g: graph) (l r: Set.set) =
  forall v . Set.mem v g.dom -> (Set.mem v l <-> not (Set.mem v r)) /\
  forall u v . G.edge g u v -> (Set.mem u l <-> Set.mem v r)
```

Algorithmically, we can check this property via breadth-first search: pick a root, color it L, and whenever we traverse an edge (u, v), color v on the opposite side from u. If we ever see an edge whose endpoints already have the same color, then no bipartition exists. Depending on the outcome, certificates take different forms:

- **Yes** (*bipartite*): a 2-coloring (L, R) that covers V, is disjoint, and separates every edge.
- **No** (*not bipartite*): a closed walk with an odd number of edges.

```
type cert =
  | Bipart (Set vertex) (Set vertex)
  | OddCycle path
```

Why does an odd cycle suffice as a "no" certificate? Intuition: each edge flip-flops sides. After k steps along a path, you are on the starting side iff k is even. Coming back to the start after an *odd* number of steps would put the start on both sides at once—impossible—so no proper bipartition can exist. Somewhat remarkably, Why3 is able to prove this fact with a straightforward induction!

```
predicate consecutive_edges (g:graph) (p:path) =
  length p >= 1 /\
  (forall i:int. 0 <= i < length p - 1 -> G.edge g p[i] p[i+1])

let rec lemma path_parity (g:graph) (l r:Set.set) (p:path) (i:int)
  requires { proper_bipartition g l r }
```

```
  requires { consecutive_edges g p }
  requires { length p >= 1 }
  requires { 0 <= i < length p }
  ensures  { (mod i 2 = 0) -> (Set.mem p[i] l <-> Set.mem p[0] l) }
  ensures  { (mod i 2 = 1) -> (Set.mem p[i] l <-> Set.mem p[0] r) }
  variant  { i }
=
  if i = 0 then ()
  else parity_along_path g l r p (i - 1);

lemma odd_cycle_implies_not_bipartite_def:
  forall g:graph, p:path.
    odd_cycle g p -> not (bipartite g)
```

The checker itself is built from straightforward set predicates (coverage, disjointness, domain membership), two one-sided neighbor checks ("every neighbor of L lies in R, and vice versa"), and a check that all of the edges in a path are in the graph. We show the computational checker to validate the edge crossings in the bipartition are valid, only to see how iteration over applicative sets works in Why3, as this is a new feature that we haven't covered before.

```
let check_side_ok (g:graph) (side other:Set.set) : bool
  requires { Set.subset side g.dom /\ Set.subset other g.dom }
  ensures { result <-> side_ok g side other }
=
  let ref ok = true in
  for u in side with Set.Iter do
    invariant { ok <->
                  forall v . Set.mem v it.Set.Iter.visited ->
                    Set.subset (G.neighbors g v) other
              }
    variant { Set.cardinal side - Set.cardinal it.Set.Iter.visited }
    ok <- ok && Set.subset (G.neighbors g u) other
  done;
  ok
```

Putting all this together, the checker dispatches on the certificate:

- for `Bipart (l,r)` it runs the set checks and the two one-sided neighbor scans; together these imply `bipartite g`;
- for `OddCycle p` it checks that the path is closed, has an odd number of edges, and really follows edges; a short parity lemma concludes `not (bipartite g)`.

Either way we derive the postcondition `result <-> bipartite g`.

Producing certificates. To produce certificates with little overhead, we extend BFS with a parent map. While exploring a component we maintain (L, R) and a queue. Discovering a fresh neighbor v of u colors v opposite to u, enqueues it, and records `parent[v] = u`. A same-side edge (u, v) triggers reconstruction of an odd cycle by joining the root$\to u$ path, the edge (u, v), and the $v \to$root path. Otherwise, when the

queue empties for all components, (L, R) is a 2-coloring. The result is passed to the verified checker above.

Termination via Fuel. Recalling that our goal as far as proving things goes with the solver is to keep a minimal footprint. We want to verify safety, in this case that all of our accesses into the adjacency map are valid, and termination to the extent that we can.

For termination, a natural variant is "number of undiscovered vertices", which varies inversely with the size of the queue. But this variant decreases only when we enqueue a *fresh* neighbor; the search also dequeues vertices whose neighbors are already discovered. Making the variant *decrease on every iteration* forces global invariants that relate (i) discovered sets, (ii) the queue, and (iii) a "no duplicate enqueues" fact. This is more than we would ideally like to prove if we are only going for safety, and indeed these facts are good progress towards proving correctness.

We can sidestep this by adding an explicit variant "fuel" argument that makes termination certain, and trivial to prove. At external call sites, we initialize `fuel` by an upper bound of how many iterations the search should ever continue for. Each expansion consumes one unit of fuel, and fuel never increases. Choosing

$$\texttt{fuel}_0 \;=\; |V| + 1$$

dominates the total number of possible expansions. The recursive call is guarded by a test on `fuel`; the `variant` is just `fuel`.

```
let rec bfs_loop (g:graph) (s:bfs_state) (fuel:int)
  : (bfs_state, option (vertex, vertex))
  requires { fuel >= 0 }
  requires { state_ok g s }
  ensures  { let st, _ = result in state_ok g st }
  variant  { fuel }
= match s with
  | State l r q pm ->
    match Q.deq q with
    | None -> (s, None)
    | Some (u, q1) ->
        let (l', r') =
          if Set.mem u l || Set.mem u r then (l, r)
          else (Set.add u l, r) in
        let s' = State l' r' q1 pm in
        let (s'', co) = scan_neighbors g u s' in
        match co with
        | Some (a,b) -> (s'', Some (a,b))
        | None ->
            if fuel = 0 then (s'', None)
            else bfs_loop g s'' (fuel - 1)
        end
    end
```

There is a sort of parallel between how we are using fuel to establish termination, and how we use certificates for correctness.

- With a *certificate* for correctness, the untrusted solver might have a bug; we will not know until run time, but when the checker *accepts* we know the result is correct for this input (and if it *rejects*, we fail fast).
- With *fuel* for termination, consider what happens if the solver if buggy. Again, we will not know until run time. The difference is that fuel ensures we *always terminate*: when fuel hits 0, we stop regardless of what the solver did.

Putting these together: a correct implementation should *never* exhaust fuel when started with $|V| + 1$, so termination arrives by emptying the queue or by discovering a conflict. If we do exhaust fuel, we still return a value—but it may be wrong because the search stopped early. This is acceptable because we then run the certificate checker. If we prematurely returned a bipartition that does not cover the domain or does not separate all edges, the checker rejects and we surface the bug. Both shift trust away from the unverified solver code. Fuel prevents non-termination even when the solver is wrong; the checker prevents us from silently accepting a wrong answer.

4 Summary

Certificate checking concentrates proof effort on tiny, easily verified components while letting an untrusted solver do the heavy lifting. Importantly, we avoid proving as much as we can of the solver, and focus our efforts on ensuring that certificate checking is efficient and correct.

- For greatest common divisor, extended Euclid produces Bézout coefficients; a small checker validates the identity and divisibility; a short lemma lifts success to $g = \gcd(a, b)$.
- For graph bipartition, a solver returns either a 2-coloring or an odd cycle. The checker uses simple set predicates, a one-sided neighbor scan, and a parity lemma to certify either outcome.

Whenever certificates are the basis for trusting that the results of a computation are correct, it is essential to incorporate testing. Without observing the solver's behavior on a sufficient span of inputs, we otherwise have no way of determining whether it will produce certificates that pass validation.