Tools for software verification

an industrial perspective

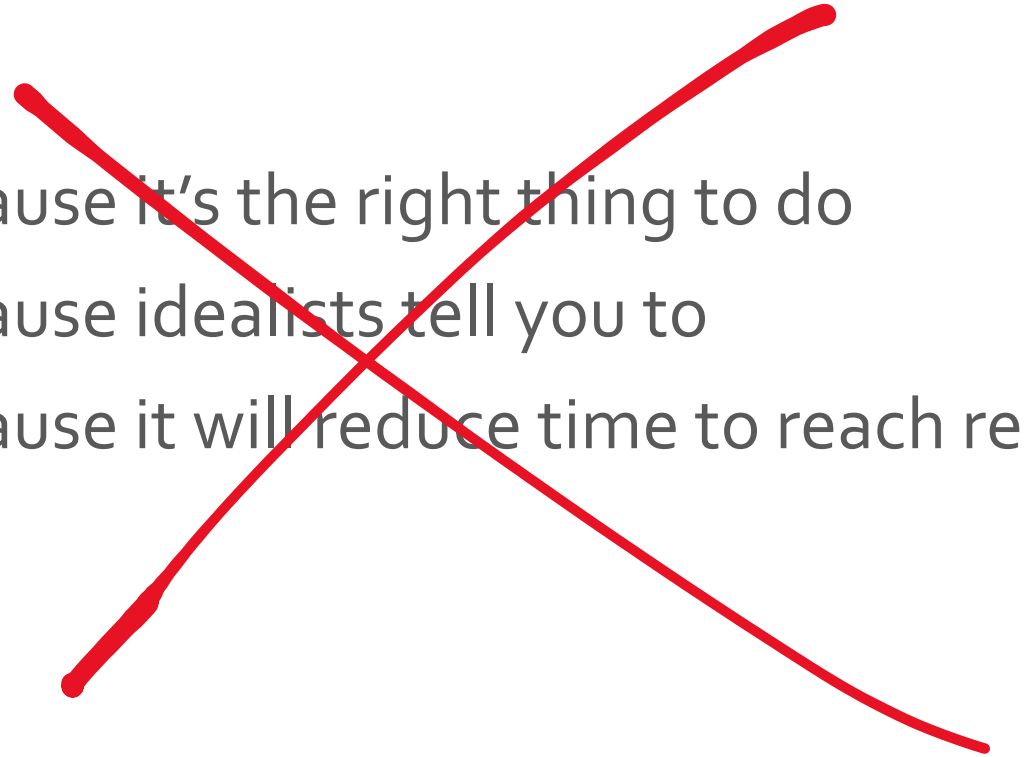
K. Rustan M. Leino

Amazon Web Services

24 June 2025
LICS 2025
Singapore

Why verify software

- Because it's the right thing to do
- Because idealists tell you to
- Because it will reduce time to reach release quality



When to verify software

- When testing is too incomplete
- When failure would be too expensive
- When it's a competitive advantage
- When code has been statistically generated

Other benefits of the process of proving programs

- Writing specifications
- Thinking of design before coding
- Spending more time with the design and code
- Isolating places of errors

Brief survey of some techniques

- Testing
- Static analysis
- Model checking
- Deductive verification

Technique:

Testing

- Idea is easy to understand
- Unsound (misses bugs)
- Requires downward-closed program
- May be difficult to set up environment

Technique:

Static analysis

type systems, abstract interpretation, data-flow analysis, …

- Sound (finds all bugs)
- Easy to apply
- Modular
- Type-like properties, coarse-grained
- User-extensible

Some successes at AWS:

- Minimum encryption-key width (Java CheckerFramework)
- Builder idiom parameter initialization (Java CheckerFramework)
- TypeScript type/property recovery (Jitterbug)

Technique:

Model checking

bounded modeling checking, systematic scheduling, run-time monitors, …

- Unsound (bounded model checking)
- Systematic and detailed
- Modular
- Can use specifications
- Hands-off
- Easy to understand error traces

Some successes at AWS:

- Memory safety (CBMC)
- Protocol modeling (P)
- Concurrency bugs (Shuttle)

Technique:

Deductive verification

interactive theorem proving,
interactive program proving

- Sound
- Supports specifications
- Requires interaction when automation runs out
- Spec/program/proof co-development
- Requires expertise and patience

Some successes at AWS:

- AWS Libcrypto (HOL Light)
- s2n-bignum (HOL Light)
- AWS authentication engine (Dafny)
- AWS Encryption SDK, AWS Datatype Encryption SDK (Dafny)

Interactive program proving

- Dafny (`dafny.org`)
- Programming language
  - Verification-aware (proof-oriented)
- Constructs for
  - Imperative programming
  - Functional programming
  - Specifications
  - Proofs
- VS Code integration

Demo

Partition, InsertionSort

```
method Partition(a: array<int>, pivot: int) returns (k: nat)
  modifies a
  ensures k <= a.Length
  ensures forall i :: 0 <= i < k ==> a[i] < pivot
  ensures forall i :: k <= i < a.Length ==> pivot <= a[i]
  ensures multiset(a[..]) == multiset(old(a[..]))
{
  var j := 0;
  k := a.Length;
  while j < k
    invariant 0 <= j <= k <= a.Length
    invariant forall i :: 0 <= i < j ==> a[i] < pivot
    invariant forall i :: k <= i < a.Length ==> pivot <= a[i]
    invariant multiset(a[..]) == multiset(old(a[..]))
  {
    if a[j] < pivot {
      j := j + 1;
    } else if pivot <= a[k-1] {
      k := k - 1;
    } else {
      a[j], a[k-1] := a[k-1], a[j];
      j, k := j + 1, k - 1;
    }
  }
}
```

```
datatype List<X> =
  Nil | Cons(head: X, tail: List<X>)

function Elements<X>(xs: List<X>): multiset<X> {
  match xs
  case Nil => multiset{}
  case Cons(x, tail) =>
    multiset{x} + Elements(tail)
}

function Insert(xs: List<int>, y: int): List<int> {
  match xs
  case Nil => Cons(y, Nil)
  case Cons(x, tail) =>
    if y <= x then Cons(y, xs)
    else Cons(x, Insert(tail, y))
}

function InsertionSort(xs: List<int>): List<int> {
  match xs
  case Nil => Nil
  case Cons(x, tail) =>
    Insert(InsertionSort(tail), x)
}
```

```
predicate Sorted(xs: List<int>) {
  match xs
  case Cons(x, Cons(y, _)) => x <= y && Sorted(xs.tail)
  case _ => true
}

lemma InsertionSortCorrect(xs: List<int>)
  ensures Sorted(InsertionSort(xs))
  ensures Elements(InsertionSort(xs)) == Elements(xs)
{
  match xs
  case Nil =>
  case Cons(x, tail) =>
    InsertCorrect(InsertionSort(tail), x);
}

lemma InsertCorrect(xs: List<int>, y: int)
  requires Sorted(xs)
  ensures Sorted(Insert(xs, y))
  ensures Elements(Insert(xs, y))
       == multiset{y} + Elements(xs)
{
}
```

Some AWS uses of Dafny

- AWS's authentication engine
  - Talk at AWS re:Inforce 2024: https://youtu.be/oshxAJGrwMU?si=2HRf2XvNR-8RMIXZ
  - Paper at ICSE 2025

Some AWS uses of Dafny

- AWS Encryption SDK
  - https://github.com/aws/aws-encryption-sdk
- AWS Database Encryption SDK
  - https://docs.aws.amazon.com/database-encryption-sdk/latest/devguide/what-is-database-encryption-sdk.html

Demo

Wildcard match

```
predicate Match(pattern: string, text: string) {
  if pattern == "" && text == "" then
  f  true
  else if pattern == "" then
    false
  else if text == "" then
    && pattern[0] == ‘*’
    && Match(pattern[1..], text)
  else if pattern[0] != '*' then
    && (pattern[0] == '?' || pattern[0] == text[0])
    && Match(pattern[1..], text[1..])
  else
    || Match(pattern[1..], text)
    || Match(pattern, text[1..])
}
```

```
predicate ValidExpansion1(patternChar: char, s: string) {
  match patternChar
  case '*' => true
  case '?' => |s| == 1
  case _ => |s| == 1 && patternChar == s[0]
}

predicate ValidExpansion(pattern: string, segments: seq<string>) {
  && |pattern| == |segments|
  && forall i :: 0 <= i < |pattern| ==> ValidExpansion1(pattern[i], segments[i])
}

ghost predicate MatchSpec(pattern: string, text: string) {
  exists segments ::
    ValidExpansion(pattern, segments) && Concat(segments) == text
}
```

```
lemma SameDefinition(pattern: string, text: string)
  ensures MatchSpec(pattern, text) == Match(pattern, text)
{
  if Match(pattern, text) {
    var segments := Ping(pattern, text);
  }
  if MatchSpec(pattern, text) {
    var segments := SegmentsFromMatch(pattern, text);
    Pong(pattern, segments, text);
  }
}

…
```

How it's done

- Interaction at the level of programs
  - "auto-active verification"
  - Dafny, Java verifier, Verus, …
- In CI or proof didn't happen
- Syntax/constructs matter
- Dafny→Java or Java→Dafny ?

Demo

Stream, Semantics

```
codatatype Stream<X> = More(head: X, tail: Stream<X>)

function From(x: int): Stream<int> {
  More(x, From(x + 1))
}

function EveryOther<X>(s: Stream<X>): Stream<X> {
  More(s.head, EveryOther(s.tail.tail))
}

greatest predicate Even(s: Stream<int>) {
  s.head % 2 == 0 && Even(s.tail)
}

greatest lemma FromEvenIsEven(x: int)
  requires x % 2 == 0
  ensures Even(EveryOther(From(x)))
{
  var s := From(x);
  assert s.tail == From(x + 1);
  var t := s.tail.tail;
  assert s == More(x, More(x + 1, t));
  assert t == From(x + 2);
  assert t.head == x + 2;
  FromEvenIsEven(x + 2);
}
```

```
datatype Stmt =
  | VarDecl(v: Variable)
  | ValDecl(v: Variable, rhs: Expr)
  | Assign(lhs: string, rhs: Expr)
  | Block(lbl: Label, stmts: seq<Stmt>)
  | Call(name: string, args: seq<CallArgument>)
  | Check(cond: Expr)
  | Assume(cond: Expr)
  | Assert(cond: Expr)
  | If(cond: Expr, thn: Stmt, els: Stmt)
  | IfCase(cases: seq<Case>)
  | Loop(lbl: Label, invariants: seq<Expr>, body: Stmt)
  | Exit(lbl: Label)
  | Return

datatype State =
  | State(m: Valuation, shadowedVariables: Valuation)
  | AbruptExit(lbl: Label, m: Valuation, shadowedVariables: Valuation)
  | Error
```

```
greatest predicate BigStep(stmt: Stmt, b3: Program, st: State, st': State)
  requires st.State?
{
  match stmt
  case VarDecl(v) =>
    exists val :: HasType(val, v.typ) && st' == st.SaveAsShadow(v.name).Update(v.name, val)
  case ValDecl(v, rhs) =>
    st' == st.SaveAsShadow(v.name).Update(v.name, rhs.Eval(st.m))
  case Assign(lhs, rhs) =>
    st' == st.Update(lhs, rhs.Eval(st.m))
  case Block(lbl, stmts) =>
    exists mid :: BigStepSeq(stmts, b3, st.ClearShadows(), mid) && st' == mid.Lower(lbl).RestoreScope(st)
  case Call(name, args) =>
    BigStepCall(stmt, b3, st, st')
  case Check(cond) =>
    if cond.Eval(st.m) == True then st' == st else st' == Error
  case Assume(cond) =>
    cond.Eval(st.m) == True && st' == st
  case Assert(cond) =>
    if cond.Eval(st.m) == True then st' == st else st' == Error
  case If(cond, thn, els) =>
    BigStep(if cond.Eval(st.m) == True then thn else els, b3, st, st')
  case IfCase(cases) =>
    exists cs <- cases :: cs.cond.Eval(st.m) == True && BigStep(cs.body, b3, st, st')
  case Loop(lbl, invariants, body) =>
    exists st0 ::
      var checks := seq(|invariants|, i requires 0 <= i < |invariants| => Assert(invariants[i]));
      BigStepSeq(checks, b3, st, st0) &&
      if !st0.State? thenst' == st0
      else
        exists st1 ::
          BigStep(body, b3, st0, st1) &&
          if !st1.State? then st' == st1.Lower(lbl)
          else BigStep(stmt, b3, st1, st')
  case Exit(lbl) =>
    st' == st.Raise(lbl)
  case Return =>
    st' == st.Raise(ReturnLabel)
}
…
```

Difficulties

- Introducing another language and/or tool
- Training
- Keeping up IDE expectations
- Libraries need specifications
- Extern specifications are error prone

Logic/type features/issues in Dafny

Many of these are waiting for theorems to be written

- Classic (not intuitionistic) logic
- Partial functions
- Types can be empty
- Axiom of choice
- Predicates defined by least/greatest fixpoint
- ORDINAL type (is it "large enough"?)
- Quantification over function values (cardinality issue in first-order encoding?)
- Abstract supertypes (traits)
- Datatypes and codatatypes
- Subset types (aka predicate subtypes or refinement types)
- Bounded polymorphism
- Type-parameter variance and cardinality restrictions
- Weakest precondition of `var x: T` where type T is empty
- Tail-recursion modulo ghosts
- Termination–general well-founded recursion/induction

Conclusions

- Software verification is in use and useful
- Many hurdles remain, technical and social
- Teach!