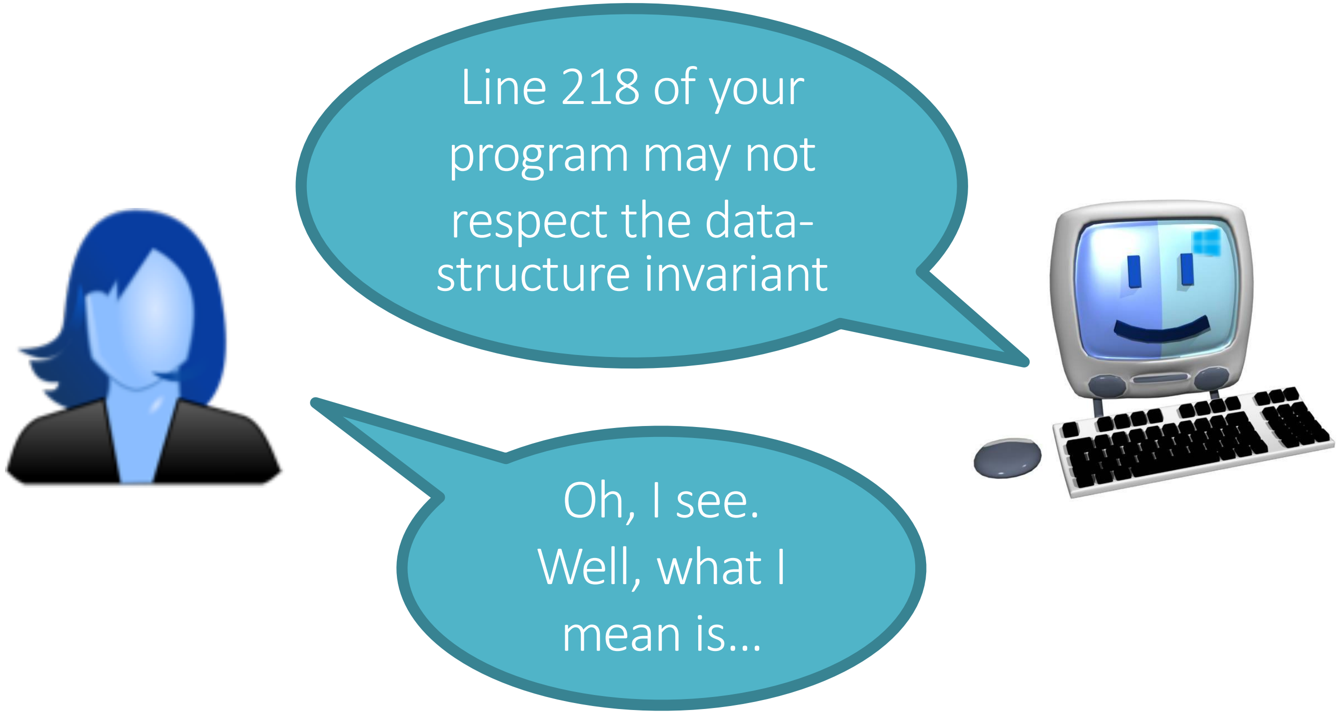
Trends in Automated Verification

K. Rustan M. Leino

Senior Principal Engineer
Automated Reasoning Group (ARG), Amazon Web Services

Keynote, SSAS 2018, 27 June 2018, NIST, Gaithersburg, MD

Automated verification



Reasoning about software programs

Old topic, considered for example by Alan Turing [1949]

Vision, along with idealism and criticism, was developed during the 1970s

Photo credits:

https://commons.wikimedia.org/wiki/File:Alan_Turing.jpg, Jon Callas from San Jose, USA.
https://www.amazon.com/Abba-11-14-Photo-Print/dp/B076CS43M6/ref=sr_1_1?ie=UTF8&qid=1530114575&sr=8-1&keywords=abba+posters
Leino,
https://commons.wikimedia.org/wiki/File:Stevie_Wonder_1973.JPG

Program verification: 1971

```
begin
  comment  This program operates on an array A[1:N], and a
    value of f(1 ≤ f ≤ N). Its effect is to rearrange the elements
    of A in such a way that:
      ∀p,q(1≤p≤f≤q≤N⊃A[p]≤A[f]≤A[q]);
  integer m, n;  comment
      m ≤ f & ∀p,q(1≤p<m≤q≤N⊃A[p]≤A[q]),
      f ≤ n & ∀p,q(1≤p≤n<q≤N⊃A[p]≤A[q]);
    m := 1;  n := N;
  while m < n do
  begin integer r, i, j, w;
    comment
        m ≤ i & ∀p(1≤p<i⊃A[p]≤r),
        j ≤ n & ∀q(j<q≤N⊃r≤A[q]);
      r := A[f];  i := m;  j := n;
    while i ≤ j do
    begin while A[i] < r do i := i + 1;
      while r < A[j] do j := j − 1
      comment  A[j] ≤ r ≤ A[i];
      if i ≤ j then
      begin w := A[i];  A[i] := A[j];  A[j] := w;
        comment  A[i] ≤ r ≤ A[j];
        i := i + 1;  j := j − 1;
      end
    end increase i and decrease j;
    if f ≤ j then n := j
  else if i ≤ f then m := i
    else go to L
  end reduce middle part;
L:
end Find
```

Source: “Proof of a program: FIND” by C.A.R. Hoare, CACM, 1971

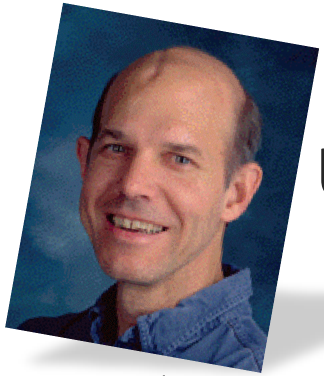
Systems for specification and verification

LANGUAGES		PROOF ASSISTANTS
GYPSY	1970-	Boyer-Moore prover
CLU		Stanford Pascal Verifier
Alphard		PRL, NuPRL
Euclid		
Eiffel	1985-	PVS
Larch		ACL2
		HOL
		Coq

Extended Static Checking (1993-2000)

To enable automation:
- Reduce ambitions from functional correctness to absence of run-time errors

To keep human effort low:
- Give up on trying to find certain kinds of errors



Greg Nelson

Underlying logical engine:
- Satisfiability Modulo Theories (SMT) solver

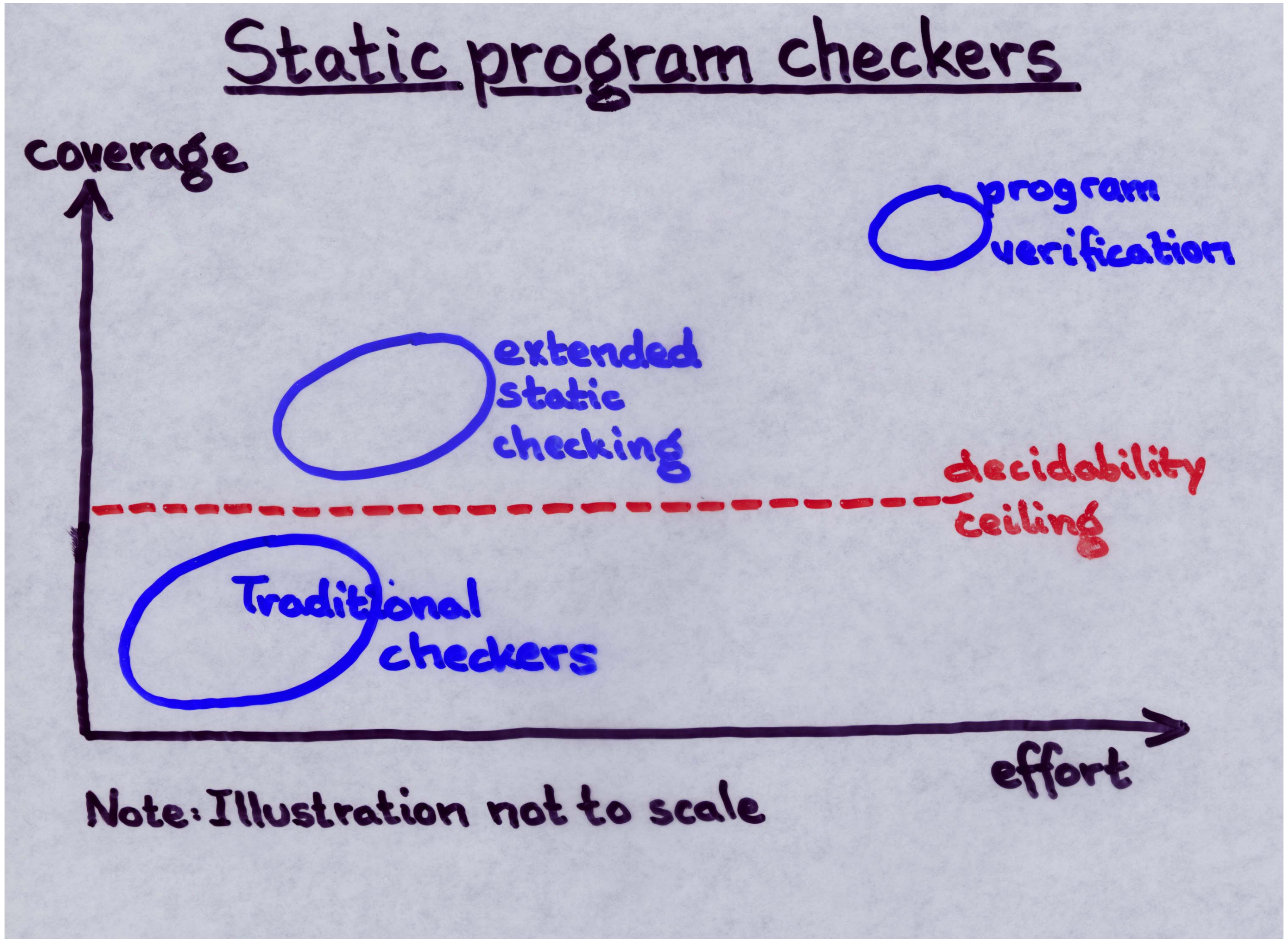
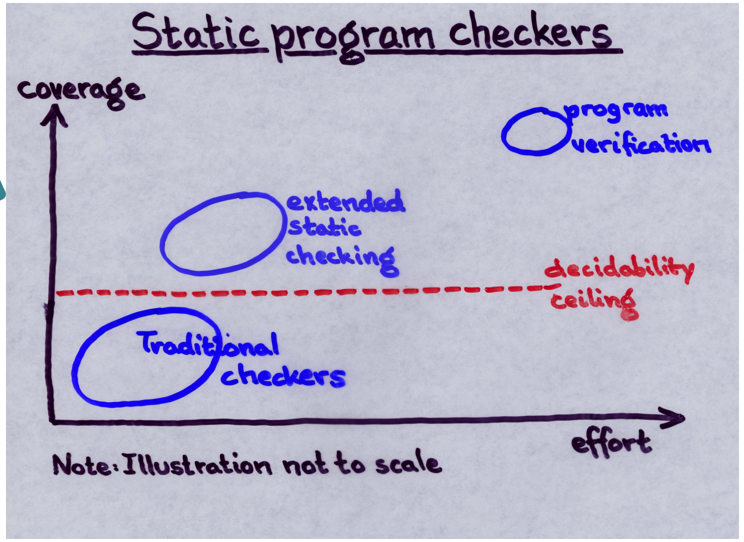
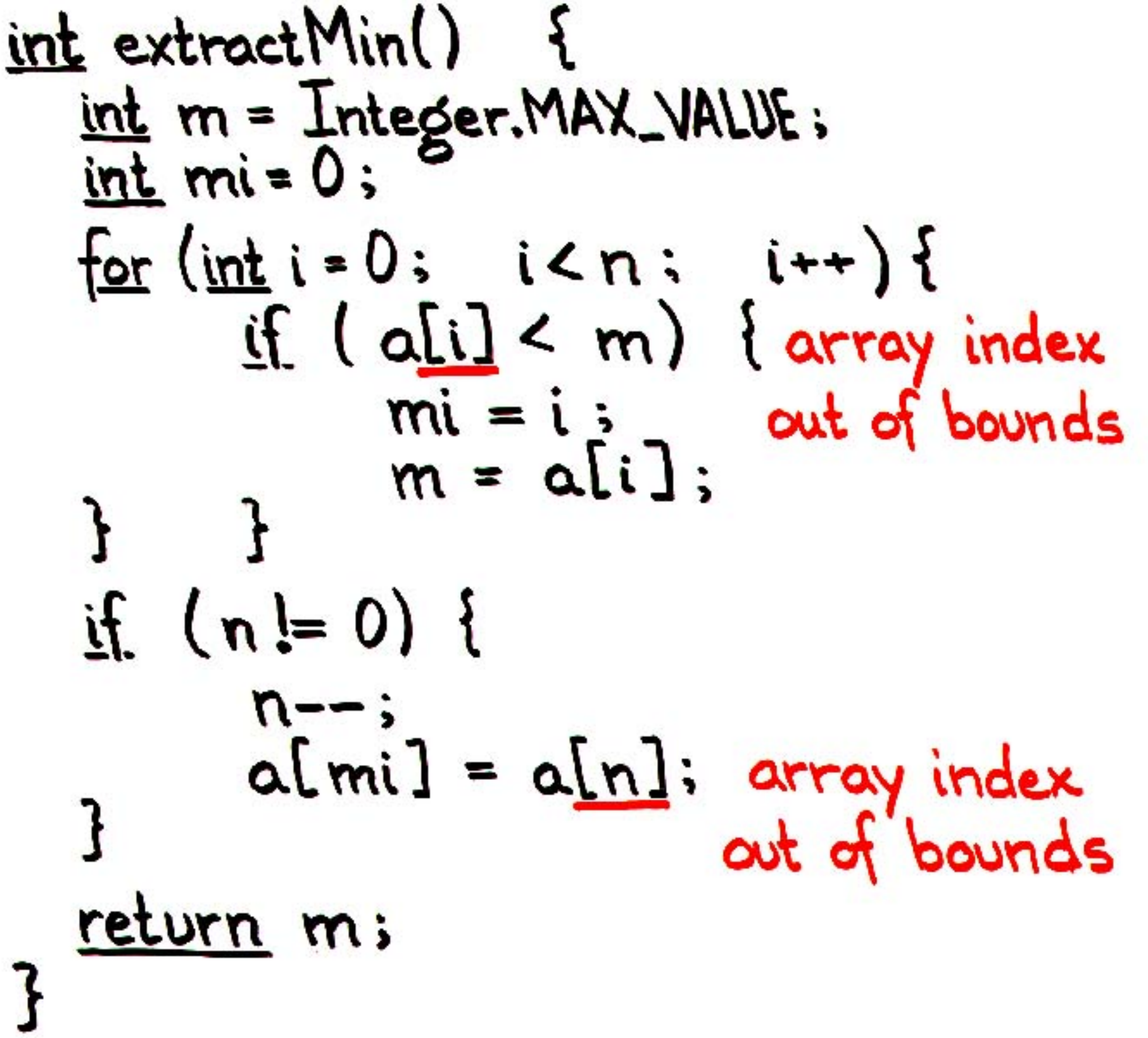


Photo credit: Compaq SRC

Source: Invited talk "ESC/Java" by K.R.M. Leino at Larch User's Group Meeting, FM'99, Toulouse, France, Sep 1999

Extended Static Checking for Java (ESC/Java)

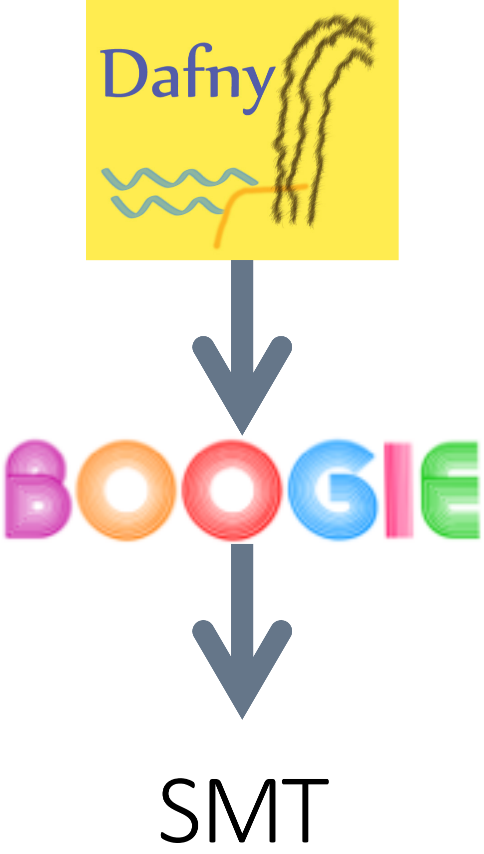




Source: Talk and first (pre-tool) ESC/Java demo by K.R.M. Leino to Hopper et al. at DEC WRL, “Extended Static Checking for Java”, March 1998

2000s: Intermediate Verification Languages

Boogie, Why, CoreStar, Viper, …



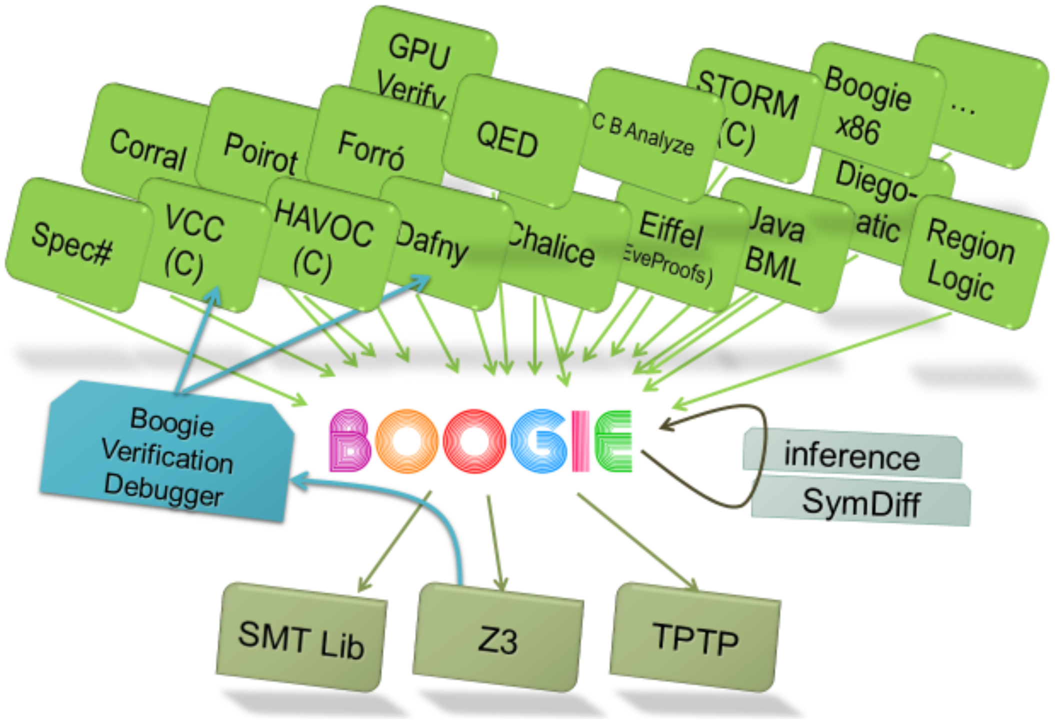
Source language

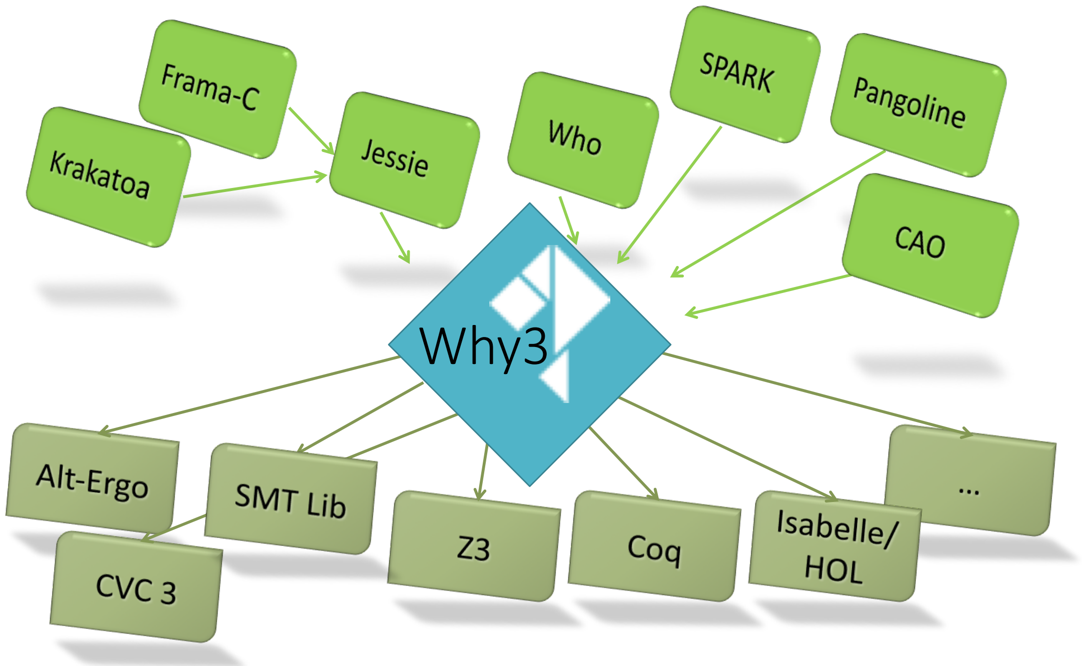
Intermediate verification language

Target logic

2000s: Intermediate Verification Languages

Boogie, Why, CoreStar, Viper, …

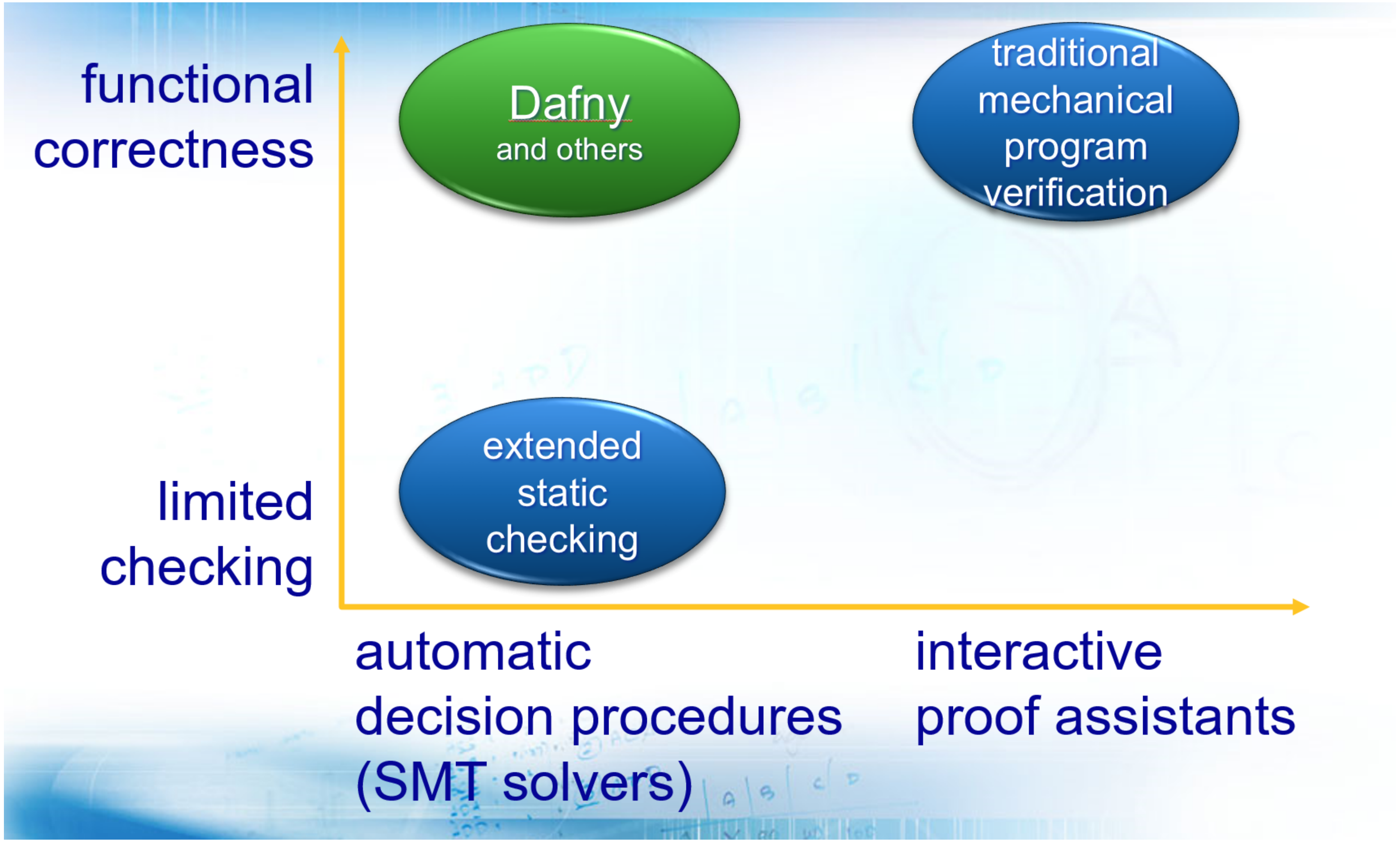




2000s: Modular specification and verification of the heap

- Ownership
  - "Boogie [Spec#] methodology": Spec#, VCC
- Dynamic frames
  - VeriCool, Dafny
- Permissions
  - Boyland's types, Plaid
  - Separation Logic: SmallFoot, jStar, VeriFast
  - Implicit dynamic frames: VeriCool 3, Chalice

Increasing ambitions (2005-)



Source: Paper presentation on Dafny by K.R.M. Leino at LPAR-16, Dakar, Senegal, April 2010

Dafny

Programming language designed for *reasoning*

Language features drawn from:

- Imperative programming
  - `if`, `while`, :=, `class`, …
- Functional programming
  - `function`, `datatype`, `codatatype`, …
- Proof authoring
  - `lemma`, `calc`, `refines`, `inductive predicate`, …

Program verifier

Integrated development environment (IDE)

Demo

Nistonacci

```
function Nist(n: nat): nat {
  if n < 2 then n else Nist(n-2) + 2 * Nist(n-1)
}

method Nistonacci(n: nat) returns (x: nat)
  ensures x == Nist(n)
{
  x := 0;
  var i, y := 0, 1;
  while i < n
    invariant 0 <= i <= n
    invariant x == Nist(i) && y == Nist(i+1)
  {
    x, y := y, x + 2 * y;
    i := i + 1;
  }
}
```

2010s

Annotated program text alone is not enough

Need ability to formalize models, state lemmas, and assist in proof authoring

This has always been possible in interactive proof assistants

- Coq, Isabelle/HOL, Agda, …

Now it has come to automated program verifiers as well

- Dafny, VeriFast, WhyML, F*, Liquid Haskell, …

Photo credits: https://pxhere.com/en/photo/593542, https://pxhere.com/en/photo/170332

Demo

Lemmas, proofs

```
lemma {:induction false} NistProperty(n: nat)
  ensures Nist(n) >= n
{
  if n < 2 {
  } else {
    calc {
      Nist(n);
    == // def. Nist
      Nist(n-2) + 2 * Nist(n-1);
    >= { NistProperty(n-2); }
      (n-2) + 2 * Nist(n-1);
    >= { NistProperty(n-1); }
      (n-2) + 2 * (n-1);
    ==
      3 * n - 4;
    >=
      n;
    }
  }
}
```

Language illustration: INC

$\text{Cmd} ::= \text{Inc} \mid \text{Cmd} ; \text{Cmd} \mid \text{Repeat}(\text{Cmd})$

Semantics given by the “big step” relation

$(\text{Cmd}, \ \text{State}) \ \rightarrow \text{State}$

where

$(C, \ s) \ \rightarrow \ t$

says that

there is an execution of command C from state s that terminates in state t

Semantics of INC

$$\mathrm{Cmd} ::= \mathrm{Inc} \mid \mathrm{Cmd};\mathrm{Cmd} \mid \mathrm{Repeat}(\mathrm{Cmd})$$

$$\frac{t = s+1}{(\mathrm{Inc}, s) \rightarrow t}$$

$$\frac{(c0, s) \rightarrow s' \qquad (c1, s') \rightarrow t}{(c0;c1,\ s) \rightarrow t}$$

$$\frac{t = s}{(\mathrm{Repeat}(body), s) \rightarrow t} \qquad \frac{(body, s) \rightarrow s' \quad (\mathrm{Repeat}(body), s') \rightarrow t}{(\mathrm{Repeat}(body), s) \rightarrow t}$$

Semantics of INC

$$\mathrm{Cmd} ::= \mathbf{Inc} \mid \mathrm{Cmd} ⨟ \mathrm{Cmd} \mid \mathbf{Repeat}(\mathrm{Cmd})$$

$$\frac{t = s+1}{(\mathbf{Inc}, s) \rightarrow t}$$

$$\frac{\exists\, s'.\ (c0, s) \rightarrow s' \qquad (c1, s') \rightarrow t}{(c0 ⨟ c1,\ s) \rightarrow t}$$

$$\frac{t = s}{(\mathbf{Repeat}(body), s) \rightarrow t} \qquad \frac{\exists\, s'.\ (body, s) \rightarrow s' \quad (\mathbf{Repeat}(body), s') \rightarrow t}{(\mathbf{Repeat}(body), s) \rightarrow t}$$

Demo

INC

```
datatype cmd = Inc | Seq(cmd, cmd) | Repeat(cmd)
type state = int

inductive predicate BigStep(c: cmd, s: state, t: state)
{
  match c
  case Inc =>
    t == s + 1
  case Seq(c0, c1) =>
    exists s' :: BigStep(c0, s, s') && BigStep(c1, s', t)
  case Repeat(body) =>
    s == t ||
    exists s' :: BigStep(body, s, s') && BigStep(c, s', t)
}
```

```
inductive lemma Monotonic(c: cmd, s: state, t: state)
  requires BigStep(c, s, t)
  ensures s <= t
{
  match c
  case Inc =>
  case Seq(c0, c1) =>
    var s' :| BigStep(c0, s, s') && BigStep(c1, s', t);
    Monotonic(c0, s, s’);
    Monotonic(c1, s', t);
  case Repeat(body) =>
    if s == t{
    } else {
      var s' :| BigStep(body, s, s') && BigStep(c, s', t);
      Monotonic(body, s, s’);
      Monotonic(c, s', t);
    }
}
```

Verified systems

	Development
Paris Metro line 14 brake system (B) seL4 Verified (Haskell, Isabelle/HOL, C) CompCert (Coq) Ironclad, IronFleet (Dafny) …	• Tool is part of development process • Specifications, code, proofs developed together • No legacy code

Accessibility

	Verification done by
Paris Metro line 14 brake system (B) seL4 Verified (Haskell, Isabelle/HOL, C) CompCert (Coq)	Formal methods experts
Ironclad, IronFleet (Dafny)	Systems programmers

Conclusions

Program verifiers

  have a high degree of automation and

  support expressive specifications

Program verification is accessible to patient, interested non-experts

Usability is important

Teach!