# What could the next ~~30~~ 10 years of software verification in climate science look like?

**Dominic Orchard**

UNIVERSITY OF CAMBRIDGE

Institute of Computing for Climate Science

University of Kent

Programming Languages and Systems for Science laboratory

**Workshop on Correctness and Reproducibility for Climate and Weather Software**
**NCAR, 9-10th November 2023**

# Terminology I will use

## Validation

*Did we implement the right equations?*

***vs.***

## Verification

*Did we implement the equations right?*

# Terminology: what does "*verified*" mean?

*Verification wrt. a specification*

i.e. check(implementation, specification)

∴ *validation* *<u>is</u>* *verification*

*where specification* $\triangleq$ *≈observation*

*The value of a specification is what we make of it;*
*it depends on our goals and values*

# State-of-the-art verification techniques…
# Testing

- Implementation language = specification language
- Test subset of inputs
  Does not show "absence of bugs" (Dijkstra); may fail to expose unconceived-of bugs
- *Property-based testing*: Automatically consider broad input space

```
def property(x : str) -> bool:
  return (reverse(reverse(x)) == x)
```

"Generator" for `str` systematically specialises / randomises inputs, finding counterexamples

(see QuickCheck, hypothesis)

# State-of-the-art verification techniques…
## Type systems

- Tightly coupled, lightweight specifications
- Static types checked automatically by compiler

  e.g. `integer :: x; logical :: y; x=x+y`    (rejected)

- Dynamic languages may support gradual / optional typing (see Python+mypy)
- Various "fancy" types in research languages capture more program behaviour
  - Dependent types (see Agda); relationships (types depending on values)
  - Refinement types (see Liquid Haskell); representation invariants
  - Graded types (see Granule); data-flow properties
  - Session types (many languages); protocols

# State-of-the-art verification techniques…

## Deductive verification

Software Analyzers

- Annotate with pre- and post- conditions

  $\{\mathsf{pre}\}C\{\mathsf{post}\}$ (Floyd-Hoare logic)

- Automated tool to check conformance (leveraging automatic solvers, e.g., Z3)
- Often needs careful design of invariants at loops
- Requires a formal language semantics
- Can be language-integrated, see Dafny

```
!= static_assert pre("deg >= 0" & "deg <= 360")
!= static_assert post("toRad >= 0" & "toRad <= 6.284")
real function toRad(deg)
  real deg
  real, parameter :: pi = 3.14159265358979323864
  toRad = 2 * pi * (deg/360)
end function toRad
```

# State-of-the-art verification techniques…
## Static analysis tools

- Specification agreed upon; general "bad behaviours" e.g.:
    - "Use after free"
    - Out-of-bounds access
    - Divide-by-zero
    - Overflow
- Not usually domain-specific
- Some nice things for floating-point, see The Herbie Project

# State-of-the-art verification techniques…
# Proof assistants / interactive theorem provers

- Impl. language = spec. language

- **But** forces implementation language choice
  - Unfamiliar
  - Not high-performance
  - Less extensive libraries

- Hugley successively in some area; big efforts

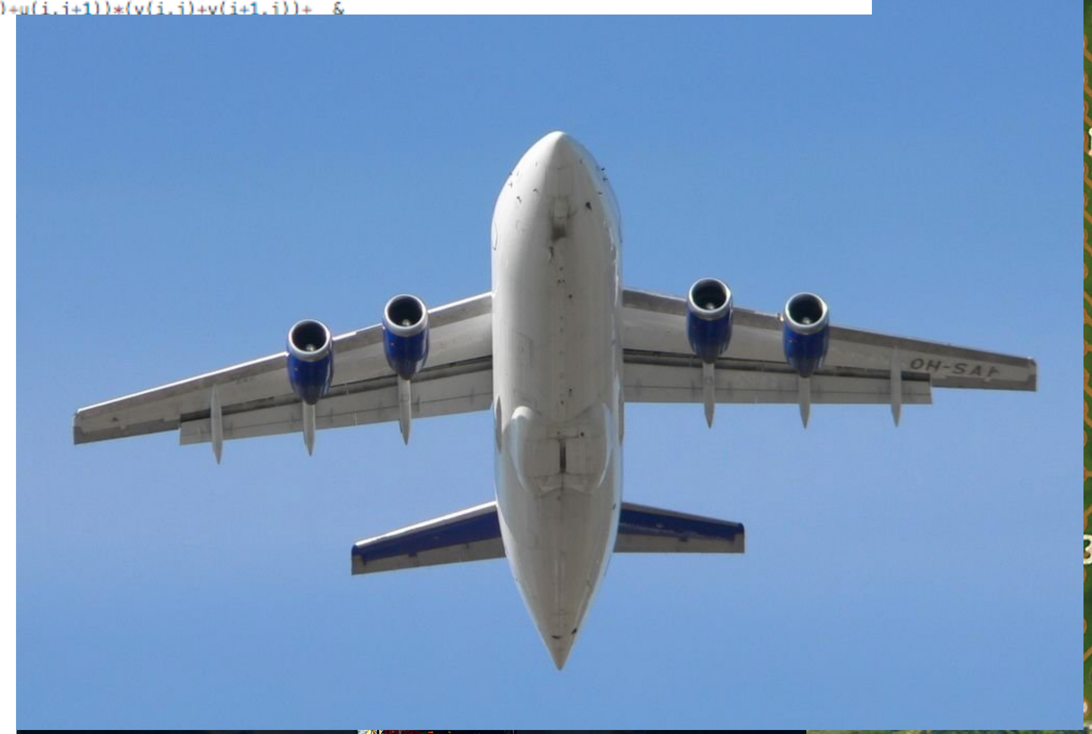
Verified microkernel

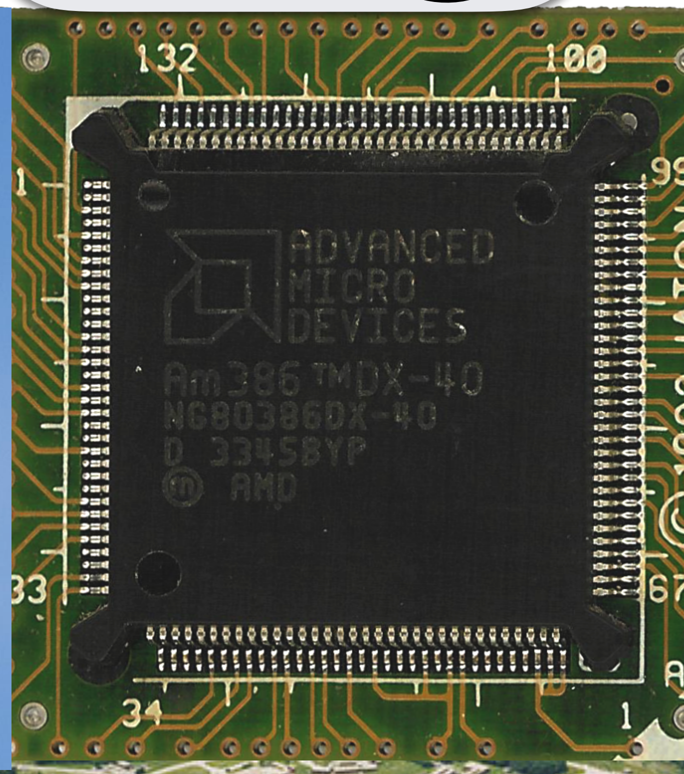
Verified C compiler

```
-----------
-- Example

open import Data.Nat

modelA1 : ℕ -> ℕ
modelA1 n = n + 1

modelA2 : ℕ -> ℕ
modelA2 0 = 1
modelA2 n = n

test1 : SimpleSpec ℕ ℕ
test1 m = m 0 ≡ 1

test2 : SimpleSpec ℕ ℕ
test2 m = m 1 ≡ 1

-- A full specification is initial in the category of specifications
fullSpec : SimpleSpec ℕ ℕ
fullSpec m = forall (n : ℕ) -> ((n ≡ 0) -> (m n ≡ 1)) × (¬ (n ≡ 0) -> (m n ≡ n))

-- Full spec implies test1 and test
initiality1 : SSpecMorphism fullSpec test1
initiality1 m x with x 0
... | (prf1 , prf2) = prf1 refl

initiality2 : SSpecMorphism fullSpec test2
initiality2 m x with x 1
... | (prf1 , prf2 ) = prf2 (\())

-- But neither of the two tests on its own subsumes the other
test12morph : ¬ (SSpecMorphism test1 test2)
test12morph s = aux
 where
   -- Counter example model that means test1 =/> test2
   counterexample : Model ℕ ℕ
```

# State-of-the-art verification techniques…
# Modelling and model checking


- Specification language based on logic
- Interrogation of model design (see Alloy)
- Model check: exhaustive search of state space
- Requires a model (can be extracted)
- Has been very effective in safety-critical systems





# Case study: end-to-end verification

J Autom Reasoning (2013) 50:423–456
DOI 10.1007/s10817-012-9255-4

**Wave Equation Numerical Resolution:
A Comprehensive Mechanized Proof of a C Program**

**Sylvie Boldo · François Clément ·
Jean-Christophe Filliâtre · Micaela Mayero ·
Guillaume Melquiond · Pierre Weis**

Received: 12 December 2011 / Accepted: 23 June 2012 / I


**Abstract** We formally prove correct a C prog
scheme for the resolution of the one-dimensior
implementation introduces errors at several leve
method errors, and floating-point computations lead to round-off errors. We annotate this C program to specify both method error and round-off error. We use

Proof assistants

Deductive verification

Automated theorem provers

Verify

$$\frac{\partial^2 p}{\partial x^2} - \frac{1}{c^2}\frac{\partial^2 p}{\partial t^2} = 0$$

discretize

$$(L_{1h}(c)\, p_h)_i \stackrel{\text{def}}{=} \frac{p_i^1 - p_i^0}{\Delta t} + \frac{\Delta t}{2}(A_h(c)\,(i' \mapsto p_{i'}^0))_i =$$

....

implement

....

```
/* Evolution problem and boundary conc
/* Propagation = time loop. */
for (k=1; k<nk; k++) {
  /* Left boundary. */
  p[0][k+1] = 0.;
  /* Time iteration k = space loop. */
  for (i=1; i<ni; i++) {
    dp = p[i+1][k] - 2.*p[i][k] + p[i-
```

# So what gets used in climate science?

## Very few of these advances

(AFAIK..!)

- Testing
- Type systems
- Deductive verification
- Static analysis
- Interactive theorem provers
- Modelling and model checking

Should we be doing more / “full” formal verification of climate models?

# “Lightweight Formal Methods” (Jackson, Wing, 1996)

"...except in safety-critical work, the cost of full verification is prohibitive and early detection of errors is a more realistic goal.

...the cost of proof is usually an order of magnitude greater than the cost of specification. And yet the cost of specification alone is often beyond a project's budget.

There can be no point embarking on the construction of a specification until it is known exactly what the specification is for; which risks it is intended to mitigate; and in which respects it will inevitably prove inadequate."

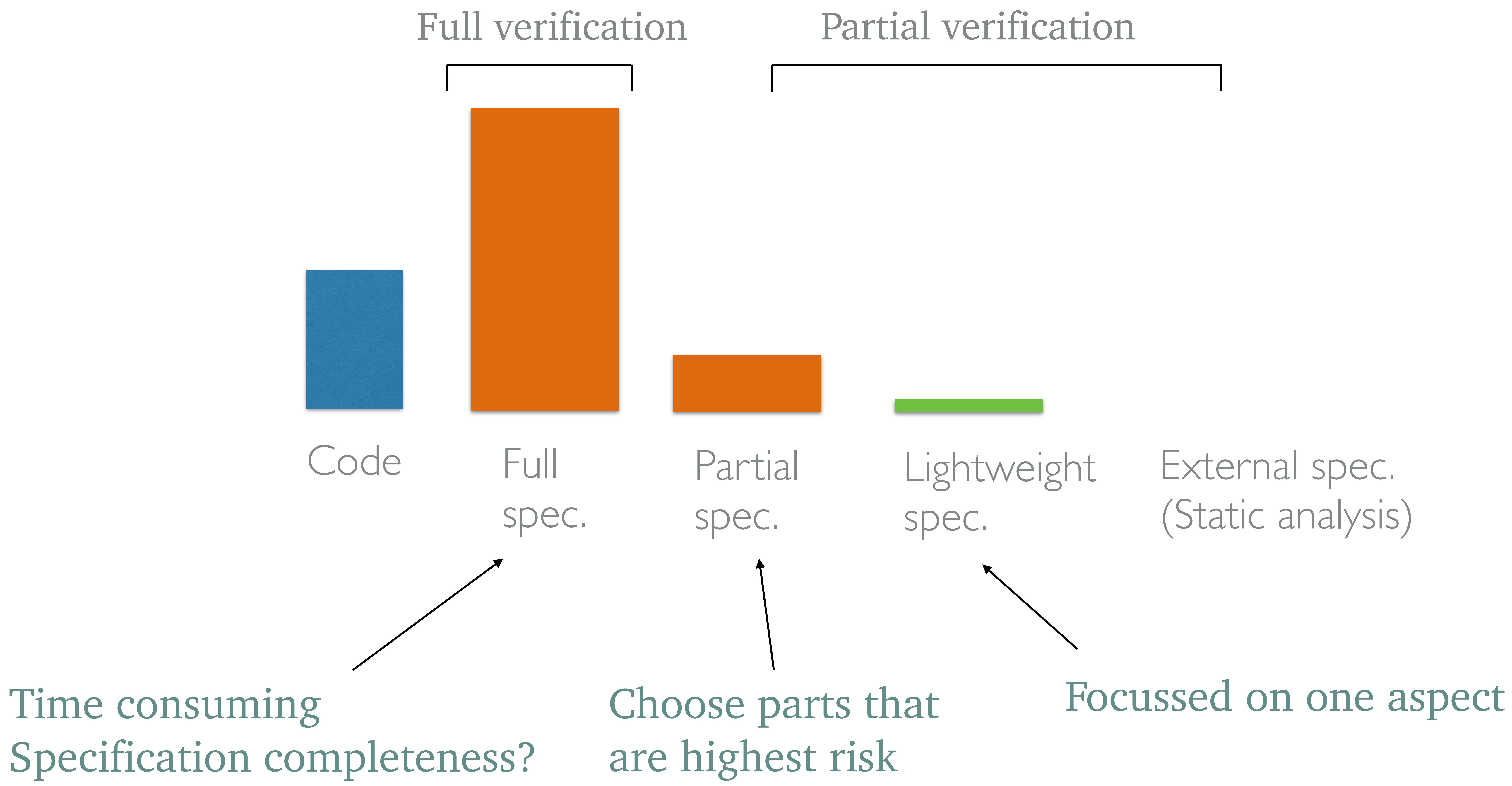
# What risks do we wish to mitigate?

## GCMs / intermediate-complexity models

- Violation of conservation / invariances
- Instability, e.g., due to unbounded error growth
- Race conditions
- <u>Slow development process due to constant bug chasing</u>

not exhaustive!

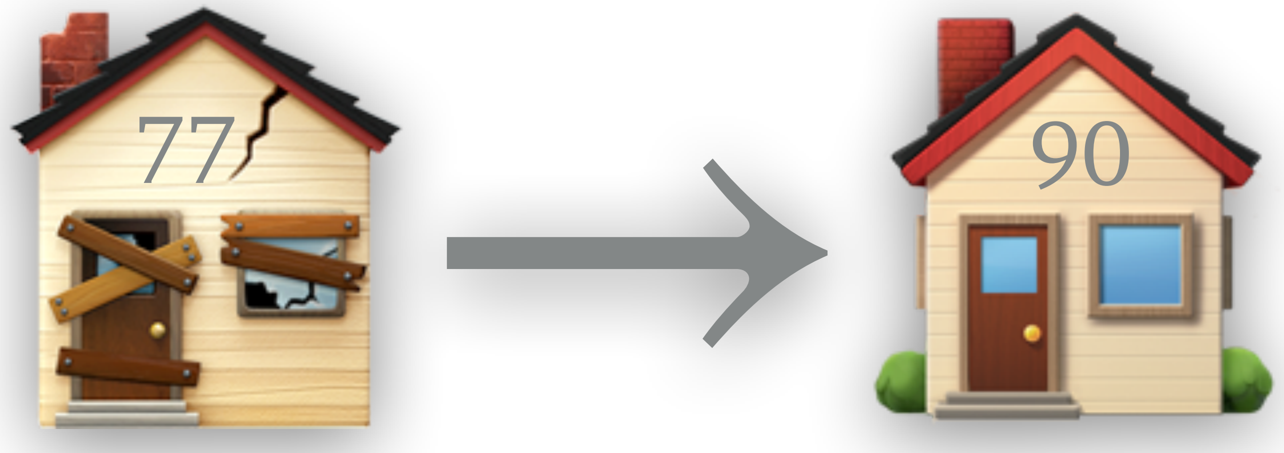
# "Cost of specification"



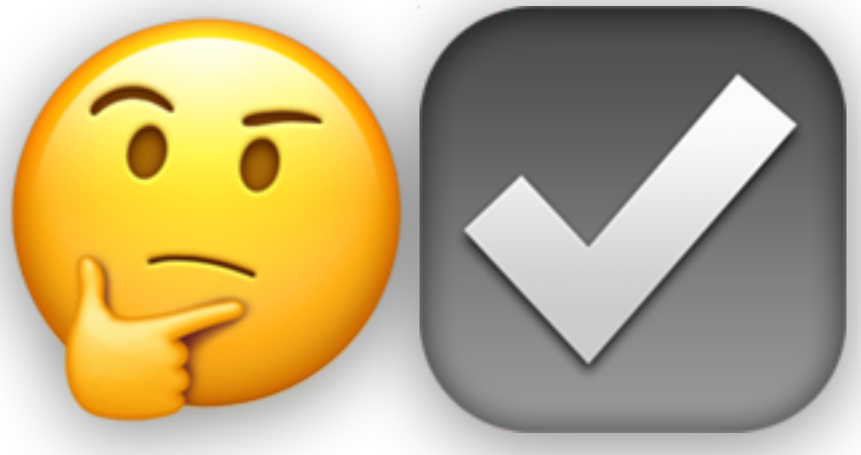
# Case study: lightweight verification for science

CamFort

Refactoring




Verification



Analysis

https://github.com/camfort/

Bloomberg

EPSRC
Engineering and Physical Sciences Research Council

Met Office Hadley Centre

```
camfort fp-check
```

## Numerical stability:

No equality (or inequality) on FP

```
camfort array-check
```

## Computational performance:

Column-major order traversal

```fortran
do i = 2, n-1
   do j = 2, n-1
     x(j,i) = x(j,i) + x(j-1,i-1) + …
```

```
camfort alloc-check
```

## Memory performance & safety:

All allocated arrays freed, no double free, or use after free

# Units-of-measure verification in CamFort

```
program energy
  != unit kg :: mass
  != unit m  :: height
  real :: mass = 3.00, gravity = 9.91, height = 4.20
  != unit kg m**2/s**2 :: potential_energy
  real :: potential_energy

  potential_energy = mass * gravity * height
end program energy
```

**Optional specifications via comments**

## Check

```
$ camfort units-check energy1.f90

energy1.f90: Consistent. 4 variables checked.
```

# Units-of-measure verification in CamFort

```
program energy
  != unit kg :: mass
  != unit m  :: height
  real :: mass = 3.00, gravity = 9.91, height = 4.20
  != unit kg m**2/s**2 :: potential_energy
  real :: potential_energy

  potential_energy = mass * gravity * height
end program energy
```

## Synthesise

```
$ camfort units-synth energy1.f90 energy1.f90

Synthesising units for energy1.f90
```

# Units-of-measure verification in CamFort

```
program energy
  != unit kg :: mass
  != unit m  :: height
  != unit m/s**2  :: gravity
  real :: mass = 3.00, gravity = 9.91, height = 4.20
  != unit kg m**2/s**2 :: potential_energy
  real :: potential_energy
  potential_energy = mass * gravity * height
end program energy
```

## Synthesise

```
$ camfort units-synth energy1.f90 energy1.f90

Synthesising units for energy1.f90
```

# Going forwards…?

- **Testing**
  - Will likely remain a mainstay (incl. *validation as proxy for verification*)
  - More deployment of property-based testing
    - Can be auto-generated from unit tests (Peleg et al. VMCAI 2018)
  - Automatic generation of tests (program synthesis)

- **Types**
  - Slow adoption of ideas into mainstream languages
  - Some form of dependent-types likely a lost-cost win
  - Julia possibly a good space for this (but long way to go; cf. `Function`)

# Going forwards…?

- **Deductive verification**
  - Hard because really needs formal semantics
  - But well established for C. Effort for Fortran? (in 2050!?)

- **Interactive proof assistants**
  - Languages not (yet) accessible
  - Unlikely unless coupled with some model extraction (+ more heterogeneous teams)
  - Potentially useful to study core models / infrastructure

# Going forwards…?

- **Static analysis**
  - Useful and easy to deploy. Big wins with some training.
  - More targeted analysis for science needed. Ideas include:
    - Sensitivity / robustness
    - Conservation

- **(E)DSLs with correct-by-construction properties?**

# Going forwards…?

## Need for more interaction!

# PROPL - Workshop on Programming for the Planet
## 20th January - London + online

https://popl24.sigplan.org/home/propl-2024

POPL 2024 Wed 17 - Fri 19 January 2024 London, United Kingdom

Attending ▾ Tracks ▾ Organization ▾ Search Series ▾ Sign in Sign up

POPL 2024 (series) / PROPL 2024 (series) /

### Programming for the Planet (PROPL)

PROPL 2024

About | Call for Papers

There are simultaneous crises across the planet due to rising $CO_2$ emissions, rapid biodiversity loss, and desertification. Assessing progress on these complex and interlocking issues requires a global view on the effectiveness of our adaptations and mitigations. To succeed in the coming decades, we need a wealth of new data about our natural environment that we rapidly process into accurate indicators, with sufficient trust in the resulting insights to make decisions that affect the lives of billions of people worldwide.

However, programming the computer systems required to effectively ingest, clean, collate, process, explore, archive, and derive policy decisions from the planetary data we are collecting is difficult and leads to artefacts presently not usable by non-CS-experts, not reliable enough for scientific and political decision making, and not widely and openly available to all interested parties. Concurrently, domains where computational techniques are already central (e.g., climate modelling) are facing diminishing returns from current hardware trends and software techniques.

PROPL explores how to close the gap between state-of-the-art programming methods being developed in academia and the use of programming in climate analysis, modelling, forecasting, policy, and diplomacy. The aim is to build

**Important Dates** AoE (UTC-12h)

**Tue 31 Oct 2023**
**Talk proposals deadline**

Wed 15 Nov 2023
Notification

Sat 20 Jan - Sun 21 Jan 2024
Workshop

**Chairs**

Anil Madhavapeddy

# Thanks

https://dorchard.github.io

types.pl/@dorchard

@dorchard

https://iccs.cam.ac.uk

CamFort https://camfort.github.io

# Backup slides

# What risks do we wish to mitigate?

## Data analysis tools

- Incorrect analysis
  - Discarded data (some data missed)
  - Duplicated data (some data used twice)
  - Wrong sign
- Scale and dimensionality mismatches
- <u>Slow development process due to constant bug chasing</u>

# Validation as a proxy for verification

- Can we get a stable run (over decades)?
- Is it plausible from physics perspective?
- Do hindcasts reproduce observational record?

If the science is relatively settled, then points to bugs not invalidity

Problem: error localisation is poor!