Lightweight Verification For Computational Science Models

Dominic Orchard

Joint work with colleagues at the University of Cambridge: Mistral Contrastin, Matthew Danish, Andrew Rice
units mismatch!
foot-pounds (lbf) vs. Newtons (N)

average bug-rate in industry software is 15-50 errors per 1000 lines**

* Z. Merali, Computational science: Error, why scientific programming does not compute, 2010
How to ensure correctness?

• Testing
  ‣ Unit testing
  ‣ Integration testing
  ‣ Combine with code-coverage checkers
  ‣ Requires significant effort

• Formal verification
  ‣ Bug finding tools (e.g. Clang analyser for C [clang-analyzer.llvm.org](http://clang-analyzer.llvm.org))
  ‣ Specification-based systems
Specification-based approaches to verification

- User specifies some aspect of the program
- A verification tool checks conformance

- **built into a language e.g. type systems**
  - Specify the broad range values that should be input and output
    - e.g. `integer :: x; character :: y; x = x / y`

- **additional specification language**
  - e.g. ACSL behaviour specs. for C  ([https://frama-c.com/acsl.html](https://frama-c.com/acsl.html))
    - relationship between input/outputs, ranges of values, and more
Specification-based approaches to verification

- Code
- Full spec.
- Partial spec.
- Lightweight spec.

Time consuming
Specification completeness?

How to chose which parts?

Focussed on one aspect
- Lightweight specification / verification of numerical Fortran
  - units-of-measure typing
  - stencil specifications (shape of array access)

- Specifications are comments

- Some specifications can be auto-generated for legacy code
Dimensional analysis

(“Great Principle of Similitude”, Isaac Newton, 1686)

\( x \) is a length \,(\text{dimension})\n
\( x \) is in metres \,(\text{unit of measure})\n
\[
\text{unit}(x \times y) = (\text{unit } x) \times (\text{unit } y)
\]
\[
\text{unit}(x \div y) = (\text{unit } x) \div (\text{unit } y)
\]
\[
\text{unit}(x + y) = \text{unit } x = \text{unit } y
\]
\[
\text{unit}(x - y) = \text{unit } x = \text{unit } y
\]
\[
\text{unit}(x^R) = \text{unit}(x)^R
\]
Example: units-of-measure specifications

```fortran
1 program energy
2  real :: mass = 3.00, gravity = 9.91, height = 4.20
3  real :: potential_energy
4
5    potential_energy = mass * gravity * height
6 end program energy
```

Suggest

```
$ camfort units-suggest energy1.f90
energy1.f90:
  (2:22)  mass
  (2:51)  height
  (3:11) potential_energy
```
Example: units-of-measure specifications

```
1    program energy
2       ! = unit kg :: mass
3       ! = unit m :: height
4       real :: mass = 3.00, gravity = 9.91, height = 4.20
5       ! = unit kg m**2/s**2 :: potential_energy
6       real :: potential_energy
7
8       potential_energy = mass * gravity * height
9    end program energy
```

Check

```
$ camfort units-check energy1.f90
energy1.f90: Consistent. 4 variables checked.
```
Example: units-of-measure specifications

1  program energy
2    != unit kg :: mass
3    != unit m :: height
4  real :: mass = 3.00, gravity = 9.91, height = 4.20
5    != unit kg m**2/s**2 :: potential_energy
6  real :: potential_energy
7
8    potential_energy = mass * gravity * height
9  end program energy

Synthesise

$ camfort units-synth energy1.f90 energy1.f90

Synthesising units for energy1.f90
Example: units-of-measure specifications

```fortran
1  program energy
2     ! = unit kg :: mass
3     ! = unit m :: height
4     ! = unit m/s**2 :: gravity
5  real :: mass = 3.00, gravity = 9.91, height = 4.20
6     ! = unit kg m**2/s**2 :: potential_energy
7  real :: potential_energy
8
8  potential_energy = mass * gravity * height
10 end program energy
```

Synthesise

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

Synthesising units for energy1.f90
```
$ camfort units-check energy2.f90

energy2.f90 : Inconsistent:
   - at 17:38 'kinetic_energy' should be 'kg m**2/s**2' instead 'kinetic_energy' is '1 kg (m / s)'

4  != unit m/s**2 :: gravity
5  real :: mass = 3.00, gravity = 9.91, height = 4.20
6  != unit kg m**2/s**2 :: potential_energy
7  real :: potential_energy
8  real :: kinetic_energy, total_energy
9
10 != unit 1 :: half
11 != unit m/s :: velocity
12 real :: half = 0.5, velocity = 4.00
13
potential_energy = mass * gravity * height
kinetic_energy = half * mass * velocity
16
total_energy = potential_energy + kinetic_energy
18 end program energy
Unit aliases

!= unit :: joule = kg m**2 / s**2
!= unit joule :: potential_energy
real :: potential_energy

Polymorphism

```plaintext
real function inch_to_cm(inch)
    real, intent(in) :: inch
    inch_to_cm = inch * 2.54;
end function inch_to_cm
```

Monomorphic

!= unit in :: inch
!= unit cm :: inch_to_cm

inch_to_cm : in → cm

```plaintext
integer function absolute(x)
    integer, intent(in) :: x
    if (x >= 0) then
        absolute = x
    else
        absolute = 0 - x
    end if
end function absolute
```

Polymorphic

!= unit 'u :: x
!= unit 'u :: absolute

absolute : ∀u . u → u
Check

Does it do what I think it does?

Infer

What does it do?

Synthesise

Capture what it does for documentation & future-proofing

Suggest

Where should I add a specification to get the most information?
Units-of-measure in other languages

• **F#** - built-in

• **Python** - Pint [http://pint.readthedocs.io](http://pint.readthedocs.io)

• **C** - Osprey (not sure if available yet)
Example: stencil specifications

! Discretisation for heat equation

do i=2, n-1
    u(i) = r1*v(i-1) + r2*v(i) + r1*v(i+1)
end do

\begin{figure}
\centering
\begin{tabular}{lcl}
\hline
\hspace{1cm} & \hspace{1cm} \\
\hline
\hspace{1cm} & \hspace{1cm} \\
\hline
\end{tabular}
\caption{Illustration of stencil patterns for \( u \) and \( v \).}
\end{figure}
Example: stencil specifications

! Discretisation for heat equation

do i=2, n-1
    \( u(i) = r1*v(i-1) + r2*v(i) + r1*v(i+1) \)
end do
Example: stencil specifications

! Discretisation for heat equation

do i=2, n-1
    u(i) = r1*v(i-1) + r2*v(i) + r1*v(i+1)
end do
Example: stencil specifications

\[ u(i) = r1 \cdot v(i-1) + r2 \cdot v(i) + r1 \cdot v(i+1) \]

! Discretisation for heat equation

do i=2, n-1
    u(i) = r1*v(i-1) + r2*v(i) + r1*v(i+1)
end do
Example: stencil specifications

! Discretisation for heat equation

\[
\text{do } i=2, n-1 \\
u(i) = r1*v(i-1) + r2*v(i) + r1*v(i+1) \\
\text{end do}
\]
Example: stencil specifications

\[ ! \text{Discretisation for heat equation} \]
\[ \text{do } i=2, n-1 \]
\[ u(i) = r1*v(i-1) + r2*v(i) + r1*v(i+1) \]
\[ \text{end do} \]
Example: stencil specifications

! Discretisation for heat equation

do i=2, n-1
  u(i) = r1*v(i-1) + r2*v(i) + r1*v(i+1)
end do

u

V
Example: stencil specifications

! Discretisation for heat equation

\begin{align*}
\text{do } & i=2, n-1 \\
\quad & u(i) = r1*v(i-1) + r2*v(i) + r1*v(i+1) \\
\text{end do}
\end{align*}
Example: stencil specifications

\![ Discretisation for heat equation \]
\[
\begin{align*}
\text{do } & i=2, n-1 \\
u(i) &= r1*v(i-1) + r2*v(i) + r1*v(i+1) \\
\text{end do}
\end{align*}
\]

$\text{camfort stencils-infer heat.f90}$

Inferring stencil specifications for heat.f90

heat.f90
(9:6)-(9:43) stencil readOnce, (centered(depth=1, dim=1)) :: v
Example: stencil specifications

! Discretisation for heat equation

\[
\text{do } i=2, n-1
\]

! = stencil readOnce, centered(dim=1, depth=1) :: v

\[
u(i) = r1*v(i-1) + r2*v(i) + r1*v(i+1)
\]

end do

$\text{camfort stencils-synth heat.f90}$

Synthesising stencil specifications for heat.f90

heat.f90

(9:6)-(9:43) stencil readOnce, (centered(depth=1, dim=1)) :: v
Two potential mistakes

8 do i=2, n-1
9     != stencil readOnce, centered(dim=1, depth=1) :: v
10       u(i) = r1*v(i-1) + r2*v(i) + r1*v(i+1) + r1*v(i+1)
11     end do

Illegal repetition of access pattern

8 do i=2, n-1
9     != stencil readOnce, centered(dim=1, depth=1) :: v
10       u(i) = r1*v(i-1) + r2*v(i) + r1*v(i+2)
11     end do

Out of bounds stencil access/
Does not conform with the shape.
More advanced specifications

- There are other primitive regions: pointwise, forward, and backward
- Two operators for composition: +, *
- Specifications acting on multiple dimensions
From a Navier-Stokes fluid simulation

\[
\begin{align*}
\text{du2dx} &= \left( (u(i,j)+u(i+1,j)) \ast (u(i,j)+u(i+1,j)) \right) + \\
&\quad \left( \gamma \ast \text{abs}(u(i,j)+u(i+1,j)) \ast (u(i,j)-u(i+1,j)) \right) - \\
&\quad (u(i-1,j)+u(i,j)) \ast (u(i-1,j)+u(i,j)) - \\
&\quad \gamma \ast \text{abs}(u(i-1,j)+u(i,j)) \ast (u(i-1,j)-u(i,j)) \\
&\quad / (4.0 \ast \text{delx}) \\
\text{dudy} &= \left( (v(i,j)+v(i+1,j)) \ast (u(i,j)+u(i,j+1)) \right) + \\
&\quad \left( \gamma \ast \text{abs}(v(i,j)+v(i+1,j)) \ast (u(i,j)-u(i,j+1)) \right) - \\
&\quad (v(i,j-1)+v(i+1,j-1)) \ast (u(i,j-1)+u(i,j)) - \\
&\quad \gamma \ast \text{abs}(v(i,j-1)+v(i+1,j-1)) \ast (u(i,j-1)- \\
&\quad u(i,j)) / (4.0 \ast \text{delx}) \\
\text{laplu} &= (u(i+1,j)-2.0 \ast u(i,j)+u(i-1,j)) / \text{delx} / \text{delx} + \\
&\quad (u(i,j+1)-2.0 \ast u(i,j)+u(i,j-1)) / \text{dely} / \text{delx} \\
\text{f(i,j)} &= u(i,j) + \text{del_t} * (\text{laplu} / \text{Re} - \text{du2dx} - \text{dudy})
\end{align*}
\]

\( \text{!}= \text{stencil} :: (\text{centered}(\text{depth}=1, \text{dim}=1) \ast \text{pointwise}(\text{dim}=2)) + (\text{centered}(\text{depth}=1, \text{dim}=2) \ast \text{pointwise}(\text{dim}=1)) :: u \)

\( \text{!}= \text{stencil} :: \text{forward}(\text{depth}=1, \text{dim}=1) \ast \text{backward}(\text{depth}=1, \text{dim}=2) :: v \)
natural & physical sciences

computer science

Let’s bridge the chasm!
Conclusions

- Correctness is very important
- Testing is good; automated verification better (reduce effort)
- Various tools for different languages
- More interaction needed between CS and sciences to build more effective tools e.g. CamFort

Future plans

- Test generation from properties
- Dependency specifications
Follow CamFort updates

http://github.com/camfort/camfort

@camfort_tool

Thank you!

Mistral Contrastin  Matthew Danish  Dominic Orchard  Andrew Rice
BETTER SOFTWARE BETTER RESEARCH

https://www.software.ac.uk/