Verification in Practice,

*Half a Century On*

Nick Maclaren

nmm1@cam.ac.uk

March 2016
Introduction

Program verification predates ‘computer science’
Mainly in numerical analysis, later in statistics

Talk covers only algorithm ⇒ execution
Languages, compilers, similar tools, run–time etc.
As used by scientific programmers, on their own
Essentially three aspects:

1. Does the code match the algorithm?
2. Is the code valid in the language?
3. Will it fail on the actual data?
Verifying Correct Logic

Leave to other talks, but will mention one aspect
What notation should be used to describe algorithms?
Badly needed for 50+ years, but no successes
Fortran, Algol 60, APL, MIX, Axiom, Matlab, Z, ...

Two very big problems:
• Rarely in terms that scientists use (which?)
• Semantic mismatch, maths vs programming

A lot of ‘optimisation bugs’ arise from the latter
Coincidentally, by Email ...


STIMULUS helps you to detect ambiguous, incorrect, missing, or conflicting requirements and to create the high-quality specifications needed for real-time safety-critical standards compliance.

Not scientific computing, but example of a tool
May be many others, perhaps some relevant
Numerical Accuracy

Not going to cover this, as too complicated
Tied up with higher levels of verification

Interval arithmetic is not the answer
Still less is precision tracking (e.g. Mathematica)

Will cover only verification against clear error
Validity Checking

Early compilers were poor – except for Algols; cc was very poor; cc+lint was just tolerable.

Modern ones are quite good, for local errors. All syntax and some of the simpler semantic rules must enable optimisation, conformance and warnings. Many warnings indicate overlooked errors.

Often need to set lots of non-default options. E.g. `gcc -std=... -pedantic -O3 -Wall -Wextra -W...`

And always worth using several compilers, of course.
Verifiers For Invalidity

Date from 1970s, mainly Fortran
Generally check some of the non-local errors
E.g. scoping, aliasing, semantic consistency
And very often check some coding standards, as well

Ancestor of such tools was PFORT
Near-bulletproof Fortran 66 verifier
Why do we have so little as good today?
Summary of Reasons

- Languages designed for flexibility, not verifiability
- Also vastly larger and more complicated
- ‘Gotchas’ – deceptive features and restrictions
- Specifications are ambiguous, even inconsistent

On the HPCF, whose fault was it (subtle errors only)?
Fortran: 50% user, 35% compiler, 15% standard
C: 10% user, 5% compiler, 85% standard
Later experience: C++, Python, Matlab in between

⇒ Exposed by parallel/asynchronism/optimisation
A few improvements since, but situation is unchanged
Aside: Effective Sizes

Fortran 2008 is $\sim 5 \times$ bigger than Fortran 77

C was never a simple language, but a simple compiler
Difficult to compare in size, as so different in type
C99 possibly comparable to Fortran 95 in size
With many more ‘gotchas’ than even Fortran 2008

C++11 is $\sim 5 \times$ bigger than Fortran 2008
Effective functionality is comparable, but very different
Much less orthogonal, so $\sim 50 \times$ more ‘gotchas’
Practical Verification

Ratio of bug types is very programmer-dependent

**Simple errors** almost always picked up by compiler
- Provided you enable most warnings, of course

Unfortunately, those don’t take most debugging time
Dominated by more subtle, *non-local* errors
Inconsistencies, failed semantic requirements etc.
Example: Invalid Aliasing

Two names for same object, at least one updated
Any of global, argument, pointers etc.
Inherently non-local property in all current languages

- Not fundamental to imperative languages

Causes data corruption; common in all languages
All forbid it, almost none even help to avoid it
Even languages like Python are not immune

One of the main reasons that people used PFORT
Example: Scoping etc.

Almost entirely caused by use of pointers
Algol 68, Ada, pointer-free Fortran are OK

Usually want hierarchical scoping (close to RAII)
Sometimes pointers bound to specific container
Or unmovable classes (e.g. for MPI non-blocking)

Can enforce in clean C++, with loss of performance
Not easy to do reliably – and may need C++11

General C pointers usually used to bypass limitations
Garbage collectors often just change to logic errors
Example: Purity Properties

Ada and Fortran allow purity (no side-effects)
Could extend C and C++ with a pragma
much harder, because of library, error flags etc.

But also need invariant functions for parallelism
Must not depend on data changed between calls

Same ranking applies to read-only arguments
Very much harder to do for C and C++

Pointers (not in Ada) cause extra problems here
In most cases, need deep, not shallow, properties
What Do We Need? (1)

- Simpler, stricter, more orthogonal languages
  Reduce human error, and enable better verification
  Lesson learnt from Algols, largely forgotten

Flaws in Algol 68 led to Pascal, Modula etc.
Major flaws in those led to C and C++
Many advantages got lost – inc. easy verifiability

‘Program proving’ displaced simple verification
Recovered in 1990s, but now faced with C and C++
What Do We Need? (2)

Converting to Haskell isn’t going to happen
Need a language that is closer to scientific algorithms
Ada, then modern Fortran, about the best
C and C++ library are beyond redemption
Verification has to select one interpretation

Portability is related to verifiability, and more obvious
Run 25+ years hence, on not-yet-designed system

I am not optimistic – need to work with what we have
Coding Standards etc.

Ancient ‘solution’, still current (e.g. MISRA, SPARK)
Too often falls into dogma and bureaucracy
Has never worked without strict verification tool

Must check non-local correctness, like PFORT
Needs strong property rules, to do it efficiently
Pointers and their use is by far the biggest problem

• Mistake to make language architecture-specific
It becomes obsolete before applications do
C and C++

Badly need a sanitised, specified C/C++ dialect
Many developers have said they would sign up
The BSI C panel considered doing it

Pragmas could add properties (e.g. purity)
WG14 and WG21 not interested; WG23 might be
If came with a good verification tool, could take off

Not sure how feasible, on two grounds
- Macros and templates change the base language
- Agreeing subset, interpretation and restrictions
Current Tools

None are in common use today, unfortunately. Those I have heard of check only some aspects. E.g. other Phaedrus tools, cppcheck, etc. Will certainly be many I haven’t heard of.

Projects often have coding standards. E.g. naming conventions, use of global data. Some of them have tools to assist with this.

Can also check specific issues for debugging. One-off checkers in Python etc. can save time! Fairly easy in Fortran and disciplined C/C++.
Dynamic Languages

Python, Matlab are very hard to debug
In general, must execute a code pathway to check it
A lot of code is for cases that should be very rare

But a tool could add quite a lot of static detection

Dynamic code means reverifying whole program, but:
• Could verify just the created code
• And that its interface meets requirements

Key is to design and enforce tight interfaces
Simple Execution Failure

Local errors that we must exclude:
1. Bad pointers, array bound errors etc.
2. Invalid values to library calls etc.
3. Using uninitialised / undefined data etc.

All cause absolute chaos, including data corruption
Worst failure is wrong answers that look plausible

All need a simple run-time check, in the general case
Raising exception is best, diagnose-and-halt is OK
Causes

Relatively few such errors are primary failures, but:

- Result of undetected error of previous forms
  Aliasing, scoping errors often show up like this
- Result of undetected numeric error
  Often takes ‘impossible’ execution path

In 1970s, undetected integer overflow caused
half of all array bound errors
Likely in C and C++ today with N-dim arrays

Ones in destructors due to much earlier ones
Numeric Errors

These are overflow, divide-by-zero, invalid
The IEEE 754 rules are not what is needed
Python’s rules are much more appropriate

Returning NaNs is possible, but:
• Using the sign of zero should return a NaN
• Division by zero should return a NaN
• The NaN state should not be losable by accident
• Ordering NaNs should raise an exception or fail

And most integers have no NaN value, anyway
Other Out-of-Domain Errors

User–defined types should behave similarly
I.e. raise an exception or return sticky invalid value

Libraries often return an error flag, value etc.
Check after every call – tool could verify

Easy to program – but little help from compilers
So mistakes get missed, and performance is poor
Implementation

Simple checking is on every use, and is slow
Mainly because interferes with SIMD execution
If done well, usually about $3\times$, less than $5\times$
Far too often, it’s $10\times$ or more – just shoddy

But most cases could be optimised out
Not full program-proving – only optimisation logic

Simpler, stricter languages make this easier
E.g. little problem for Ada and Fortran
Extreme Example

SUBROUTINE CHOLESKY(A)
    INTEGER :: J, N
    REAL(KIND(0.0D0)) :: A(:, :), X
    N = UBOUND(A,1)
    DO J = 1, N
        X = SQRT(A(J,J)-DOT_PRODUCT(A(J, :J-1), A(J, :J-1)))
        A(J,J) = X
        IF (J < N) &
            A(J+1:, J) = (A(J+1:, J) - &
                MATMUL(A(J+1:, :J-1), A(J, : J-1))) / X
    END DO
END SUBROUTINE CHOLESKY

Clearly needs only a single check
gfortran does that, but not on the matching solver
Complex Execution Failure

Non-local errors like aliasing, scoping
Their occurrence may depend on dynamic data

NAG Fortran checks scoping, but not aliasing
Other verification tools similar, usually less good

Some C/C++ tools will pick up simple cases
But, in general, there’s essentially nothing
And many clear mistakes are legal in C/C++
Current Tools

Often compiler options for adding checks
Usually patchy and often unnecessarily inefficient
Fortran (and Ada?) best, from WATFIV to NAG
E.g. gfortran -fcheck=all -ftrapv
       -ffpe-trap=invalid,zero,overflow

A fair number for specific problems, mainly memory
valgrind, Parallel Inspector, and many others
They often check in only some contexts

C/C++ has no clear legal/illegal boundary
So checking tends to be much patchier
Summary for Programmers

There is a lot that is available today
Takes effort, but can save many times more

Can do a great deal for Fortran, maybe Ada

Very much trickier for C and C++
• Usually possible only for disciplined code
Summary for Developers

• Bridge gap between CS research and practice

Compilers could do much more for little cost
More on verification, not just features and speed
Optimisation logic could (should!) be repurposed

Also better verification and instrumentation tools
They’re just code⇒code compilers!

But need a well-defined C and C++ variant