Systematic Errors in Climate Models Consequent on the Design of the Fortran Language: Detection and Correction

John Collins[1,2,3,*] Mark Anderson[1] Brian Farrimond[1,2,3] Darryl Bayliss[1] Darryl Owens[1]
1. Edge Hill University UK, 2. University of Cape Town, 3. SimCon Ltd. UK
john.collins@simconglobal.com

Abstract

Many classes of systematic error in Fortran programs have been largely eliminated by changes in language design both in and since the Fortran 90 standard. For example, the replacement of common blocks by modules removes the risk of mis-alignment, and free-format layout prevents many simple textual errors. However, some issues remain, or have actually been introduced by these changes. For example:

- **Precision**: The computation of high precision results in expressions is often degraded without warning by the accidental inclusion of low precision variables. Also the precision of a Fortran literal number may not be that implied by its textual representation.
- **INTENT**: Errors in the declaration of the INTENT of a sub-program argument (Whether input, output or both) may cause compilers to generate incorrect code.
- **OPTIONAL arguments**: Optional arguments may accidentally be passed down into lower level routines without a check that they are present in the argument list.
- **Accidental whole array assignments**: If the subscript of an array is accidentally omitted and an assignment is made from a scalar variable, the entire array is assigned.

Climate models are particularly vulnerable to some of these issues because of the extensive use of modern Fortran constructs. Tests are described by which these errors may be detected, and in some cases it may be shown that classes of error do not occur. Some classes of error may be corrected automatically.

The analysis was applied to WRF, and examples from this model are described.

WRF

WRF is an open source community code maintained at The National Center for Atmospheric Research at Boulder, Colorado. It is used both for weather prediction and for atmospheric research. WRF is a mesoscale model, scalable from a grid of a few metres to thousands of kilometres. The code may be built for a single processor, or for massively parallel systems. There are over 20,000 users worldwide.

The version of WRF analysed in this study is 3.4.1.

WRF is a large program. It is distributed as CPP pre-processor code which is pre-processed to compilable Fortran before analysis. The metrics for the compilable code are:

Files	
Primary files	386
Include files	3
Code and comments	
Declaration lines	136,759
Executable lines	332,745
Total code lines	469,504
Comment lines	24,281
Total lines	493,785
Program Units	
Programs	1
Modules	320
Subroutines	4,304
Functions	35
Module subroutines	1,933
Module functions	299
Internal subroutines	31
Internal functions	9
Generic interfaces	57
Specific interfaces	113
References to C sub-programs	182

Precision

A real constant compiled by a Fortran 90 compiler will always be a single precision value unless a different precision is specified, either by a *kind* tag or by the exponent character. For example, 3.1415926535897931 is a single precision number, usually occupying 4 bytes. 3.1415926535897931_8 and 3.1415926535897931D+00 are double precision numbers. Note that this behaviour differs from that of the legacy extended Fortran 77 compilers such as DEC VMS Fortran, where the precision of a real constant was inferred from context. A common error is to use a literal constant of the wrong precision.

WRF contains 235 occurrences of double precision Fortran parameters which are assigned from single precision values. For example, all of the double precision constants for the physics codes are all set to single precision values. Part of the code is shown in the following column.

```
module module_gfs_physcons
  use module_gfs_machine,only:kind_phys

  real(kind=kind_phys),parameter:: con_pi =3.1415926535897931
  real(kind=kind_phys),parameter:: con_sqrt2 =1.414214e+0
  real(kind=kind_phys),parameter:: con_sqrt3 =1.732051e+0

  real(kind=kind_phys),parameter:: con_rerth =6.3712e+6
  real(kind=kind_phys),parameter:: con_g =9.80665e+0
  real(kind=kind_phys),parameter:: con_omega =7.2921e-5
  real(kind=kind_phys),parameter:: con_rd =2.8705e+2
  real(kind=kind_phys),parameter:: con_rv =4.6150e+2
  real(kind=kind_phys),parameter:: con_cp =1.0046e+3
  real(kind=kind_phys),parameter:: con_cv =7.1760e+2
```

Most of these constants are specified with too few digits for the loss of precision to be significant. However, it is perhaps a weakness in the language that the precision of a value such as that of con_pi is degraded without warning.

The Fortran standard recommends the use of kind tags to specify the precision of a real numeric constant. The WRF code contains:

Tagged single precision real literals	14,520	10.5%
Tagged double precision real literals	2,415	1.7%
Total tagged	16,935	12.2%
Untagged single precision real literals	121,081	87.3%
Untagged double precision real literals	726	0.5%
Total untagged	121,807	87.8%
Total real literals	138,742	

WRF also contains 3,672 occurrences of expressions with mixed real kinds. For example (module_bl_gfs line 151):

```
HEAT(i)=HFX(i,j)/CPM*RRHOX
EVAP(i)=QFX(i,j)*RRHOX
```

All of the terms in these statements are 8-byte real numbers except HFX and QFX, which are single precision numbers. In consequence, HEAT and EVAP are accurate only to single precision even though they occupy 8 bytes. The loss of precision occurs without warning.

The software engineering tools used in this study can report anomalies in precision and can correct them, by modifying declarations and by changing the text of literal numbers. These changes cause appreciable changes in the program output.

It is suggested that these issues could in part be addressed by a new Fortran statement, similar to an IMPLICIT statement, which would specify the default kind of an untagged real literal constant.

INTENT

Fortran 90 allows the intent of a sub-program argument to be specified as:

INTENT(IN) The sub-program may read the value of the argument but does not assign it.

INTENT(OUT) The sub-program may assign a value to the argument and does not read it before it is assigned.

INTENT(IN OUT) The sub-program may read the value of the argument and may assign it. This is the default case

The INTENT declaration is an assertion which the compiler may check against the sub-program code. If an argument declared INTENT(IN) is assigned an error may be reported. Simple test programs were written to examine this behaviour with a number of widely used compilers. Tests were made with Compaq Visual Fortran, Salford Software FTN95, gfortran, g77 and Intel ifort. Three cases were examined:

Simple violation: an INTENT(IN) argument assigned directly across an = sign. This was reported as an error by all compilers tested.

Module violation: an INTENT(IN) argument was passed into a module subroutine in the same Fortran module, and assigned directly across an = sign in that module. No compilers detected this case.

Call violation: an INTENT(IN) argument was passed into a separately compiled subroutine and assigned in that routine across an = sign. This was detected as an error by only one compiler, Salford FTN95.

Software engineering tools were used to check the INTENT declarations in WRF. Three classes of INTENT violation are detected:

INTENT(IN) violation: INTENT(IN) arguments which are written to (always because they are passed down into other routines where they are assigned).

INTENT(OUT) violation: INTENT(OUT) arguments which are always read before they are assigned.

Possible INTENT(OUT) violation: INTENT(OUT) arguments which may be read before they are assigned but where the program flow is data-dependent and uncertain.

In WRF, the INTENT declarations are:

Total number of arguments	53,908	
Declared INTENT(IN)	28,460	
Declared INTENT(OUT)	6,155	
Declared INTENT(IN OUT)	5,384	
Variables with no declared INTENT	13,884	
Sub-program formal arguments	25	
INTENT(IN) violations	137	0.5%
INTENT(OUT) violations	102	1.7%
Possible INTENT(OUT) violations	1,292	21.0%

The violations are errors. If INTENT declarations are treated only as assertions which the compiler may check, the situation is not serious. However, this is not made clear by the Fortran standard. If the compiler may make use of the INTENT declarations in code optimisation, the errors may be significant. For example, a compiler might rely on the fact that an INTENT(IN) argument is not changed by a sub-program call, or might not import the value of an INTENT(OUT) argument when a routine is called.

The use of INTENT by two compilers was investigated. All non-mandatory INTENT declarations were stripped from WRF automatically (A small number remained, in the interfaces of overloaded operators and in ELEMENTAL routines). The code was built under gfortran and ifort and the results were compared with those from un-modified code. There were no changes. These compilers do not use INTENT declarations in code generation.

The INTENT declarations may also be corrected, and missing declarations supplied automatically. This is an issue in the future maintenance of WRF. It is clear that manual maintenance of INTENT declarations is difficult. Violations may be introduced by modification of routines in high branches of the call-tree remote from the declarations themselves. If INTENT specifications are left in place or are supplied, then either a maintenance trap is created or a permanent commitment is made to the use of software engineering tools. If they are stripped, there is a loss of documentation.

OPTIONAL Arguments

Arguments to Fortran 90 sub-programs may be declared to be optional, and may be omitted from the list of actual arguments when a routine is called. A problem arises if an optional argument which has been omitted is accessed in the body of the sub-program. This is an error, but is not always reliably trapped. The intrinsic function PRESENT is used to determine whether an optional argument has been specified, and access to optional arguments should be guarded by IF (PRESENT(... constructs.

The WRF code was analysed for the use of optional arguments which are not guarded. The analysis shows:

Number of optional arguments	2118
Optional arguments with unguarded references	1298
Total number of unguarded references	2352

This is a difficult analysis, because logical variables may be constructed from the PRESENT() tests, and may be used in a non-trivial way to guard access to the arguments. The number of unguarded optional arguments is therefore a slight over-estimate. However, a sample was checked by hand and most were found to be genuine errors.

It is suggested that compilers should be required to check that optional arguments are correctly guarded.

Accidental Whole Array References (And good news)

One of the changes introduced in the Fortran 90 standard is the facility to assign every element of an array from a scalar value. If, for example, the array A is declared with dimensions A(1:10,1:4) the statement A = 0.0 sets all elements of A to zero. A side-effect of this change is that it is possible to omit the indices of an array accidentally and assign the entire array instead of one element. These errors are not uncommon in aerospace codes.

We are able to demonstrate that this never occurs in WRF.

The Software Engineering Tools

The tools used in the study are components of WinFPT (**http://www.simconglobal.com**). This is an analysis and re-engineering tool for Fortran, maintained by two of the authors.

Acknowledgments

This work is supported by:

The United States Air Force

UCAR and NCAR

Edge Hill University

SimCon Ltd.

The authors wish to thank:

Prof. W. J Gutowski Jr. Iowa State University

Prof. Bruce Hewitson, University of Cape Town

Dr. David Gill at UCAR

for their help and support.