Automatic Detection of Software Errors in WRF

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

Introduction

WRF is probably the most important computer program ever written. It is also a triumph of software engineering. However it is unlikely that a program of this size would not contain coding errors, and even if it did not, there are errors in the compilers with which it is built.

Software tools are used in the aerospace and other engineering industries to analyse programs for errors. The authors have adapted one tool, WinFPT (http://www.simconglobal.com) for the analysis of WRF.

How are errors detected

Programming errors are detected by:

- Static analysis of the source code
- Dynamic analysis of the running program

Static Analysis

The software tool, WinFPT, reads the source code in the same way as a compiler. It carries out detailed static semantic analysis - identifying the operators and variables and all of their attributes. It differs from a compiler in that:

- It analyses all of the modules and sub-programs together, correlating information between them.
- It has specific analyses for classes of error and inconsistency which may be correct Fortran but which may indicate that an error has occurred.

The static analyses check, for example:

- **Arguments** - Do the actual arguments passed to sub-programs match the formal arguments in the sub-program declarations?
- **Names** - Do objects with the same names in different routines have the same attributes? For example, is there a Fortran parameter named g with different values in different routines?
- **Expressions** - Is there a loss of precision where single precision variables and constants are mixed with double precision variables?
- **Usage** - Are variables used before they are initialised? Are values computed but never used?
- **Intent** - Are sub-program arguments with intent declared (in) actually modified? Are arguments declared intent (out) used before they are written to?

All of these issues have been identified in WRF.

Dynamic Analysis

Some classes of error are only detected when the code is executed. These include:

- **Compiler bugs** - where the code is correct but built incorrectly
- Dynamic array references **out-of-bounds**
- **Uninitialised variables** where the path through the code cannot be analysed statically

These errors lead to differences in results under different compilers and operating systems and under different multiprocessor configurations.

These all occur in WRF.

Measuring the code

Static analysis also measures the WRF code. The measures include:

- How big is it - how many lines?
- How complex is it - what is the **cyclomatic complexity**?
- How well is it commented - how many comments, and how many words in comments?
- How meaningful are the variable names? - How long are they?
- How is it constructed? Are data organised in Fortran modules or in common blocks; are sub-programs organised in modules or are they linked as separate objects?
- How many errors and anomalies are there? Anomalies, such as mixed precision arithmetic, are not errors but could degrade the accuracy of the models.

Preparing WRF for Analysis and Instrumentation

WRF source code is not distributed as directly compilable code. It is pre-processed before compilation to adapt to the specific compiler and multi-processor configuration. The analysis tools cannot analyse the pre-processor code.

The WRF build procedure was therefore modified to capture the intermediate Fortran files. These were moved to a parallel directory structure for analysis. The build procedure was further modified to build instrumented versions of the WRF code.

Measuring WRF Program Size

The measurement of a program such as WRF is most meaningful when compared with other programs of comparable size. The analysis tools used were designed for use in the aerospace and engineering industries, and WRF was compared with a typical program from that environment. *"Radar"* is the signal processing of a tracking radar control system. WRF is very different !

Program Size

		WRFV3	*Radar*
Files			
	Primary files	348	1,589
	Include files	3	1,891
Code and comments			
	Declaration lines	100,234	49,831
	Executable lines	235,342	100,203
	Total code lines	335,576	150,034
	Comment text lines	8,351	236,033
	Comment separator lines	406	7,942
	Blank lines	16,077	87,402
	Total comment lines	24,834	331,377
	Total lines	360,410	481,411

Two differences are striking. Firstly, WRF is made up of a relatively small number of large files, many of which contain large numbers of declarations and sub-programs. The radar code has many primary files, each of which contains one, or a small number of related sub-programs. Secondly, the WRF code has almost no comments. This measurement is spurious. The compilation pre-processors strip the comments from the code.

Measuring WRF Fortran Language Usage

WRF Language

	WRFV3	*Radar*
Programs	4	38
Block Data	0	11
Modules	216	0
COMMON Blocks	1	110
Subroutines	2764	2021
Functions	29	21
Module subroutines	1267	0
Module functions	198	0
Internal subroutines	30	233
Internal functions	2	1

The most important differences in program organisation are:

- WRF uses Fortran modules to organise data, the radar code uses common blocks in include files;
- WRF uses modules to organise sub-programs. Nearly all sub-programs in the radar code are linked at the top level.

This has important implications for the risk of errors in the code. Compilers check the interfaces of sub-programs within modules, and the use of modules therefore significantly reduces the risk of mis-matched arguments. Common blocks must be aligned correctly in different routines, and there is a significant risk that misalignments will occur in the radar code. There is, however, a penalty in using modules in WRF. A routine may be changed in the radar code and the system may be built and run within 30 seconds. The interdependence of modules in WRF may lead to build times of the order of 20 minutes when changes are made.

Intent Errors

The most common potentially serious error in WRF is the declaration of a sub-program argument to be INTENT (IN) when the argument is actually modified within the routine. For example:

```
File: share/module_io_domain.f90
SUBROUTINE input_boundary(fid,grid,config_flags,ierr)
        IMPLICIT NONE
        TYPE (domain) :: grid
        TYPE (grid_config_rec_type),INTENT(IN) :: config_flags
!-----------------------------------------------------------------
!!! FPT - 2491 INTENT declared IN but argument is written to:
!-----------------------------------------------------------------
        INTEGER,INTENT(IN) :: fid
        INTEGER,INTENT(INOUT) :: ierr
        IF (config_flags%io_form_boundary .GT. 0) THEN
           CALL input_wrf(fid,grid,config_flags,          &
            boundary_only,ierr)
        ENDIF
        RETURN
END SUBROUTINE input_boundary
```

The variable config_flags is then modified in input_wrf:

```
File: share/input_wrf.f90
SUBROUTINE input_wrf(fid,grid,config_flags,switch,ierr)
           :
        TYPE (grid_config_rec_type),INTENT(INOUT) :: config_flags
               :
        IF (IERR .NE. 0) THEN
           IF (MMINLU == 'UMD') THEN
              config_flags%iswater = 14
           ELSE
              config_flags%iswater = 16
           ENDIF
        ENDIF
```

The problem is that the compiler may use the INTENT statement to optimise the code, and may do so incorrectly.

Detecting Errors by Dynamic Analysis

The approach is to "**instrument**" the code by systematically inserting statements which monitor or change behaviour. The instrumented code is then built and run.

Dynamic Analysis – Trapping Errors when WRF is Run

Differences in model output are always observed when WRF is built with different compilers. The issue was analysed by instrumenting the WRF code to capture the result of every scalar assignment statement. For example:

```
DO k = kte,kts,-1
   CALL trace_i4_data('K',k,55210)
   lamr = (am_r*crg(3)*org2*nr(k)/rr(k))**obmr
   CALL trace_r8_data('LAMR',lamr,55211)
   ilamr(k) = 1.0/lamr
   CALL trace_r8_data('ilamr(k)',ilamr(k),55212)
   mvd_r(k) = (3.0+mu_r+0.672)/lamr
   CALL trace_r4_data('mvd_r(k)',mvd_r(k),55213)
   n0_r(k) = nr(k)*org2*lamr**cre(2)
   CALL trace_r8_data('n0_r(k)',n0_r(k),55214)
ENDDO
```

The statements added by WinFPT are shown in red.

In the first run of WRF, the trace subroutines capture the outputs of the expressions to file. The outputs are labelled by the string (the first argument) and by an integer identifier which labels the statement and allows the analysis to follow the program flow.

When WRF is rebuilt with the second compiler, it would be possible to capture the data again and to compare the results. However, small differences in rounding are expected to cause the results of the runs to drift apart, and comparison of the two runs is impractical. Instead, the data from the first run are read during the second, and the results of each expression in the second run are compared on the fly with the results from the first run. If there are small differences the results from the second run are overwritten by those from the first run, and this kills the numerical drift. If there is a large difference, or if the code follows a different path, an error is reported.

In WRF, this technique has exposed:

- Uninitialised variables;
- A coding error where an operator overload is not correctly exported from a module;
- A compiler bug where module sub-program attributes are handled incorrectly.

Correcting the Errors

The next step in this study is to correct the errors, and to re-run test cases to determine whether there has been any significant impact on the results. Some of these errors, for example, those in the intent of sub-program arguments, can be corrected automatically by the tools.

Acknowledgements

This work has been supported by:
Edge Hill University (UK)
Liverpool Hope University (UK)
United States Air Force Weather Agency