Automated Quality Assurance Analysis: WRF – A Case Study

John Collins and Brian Farrimond
Remote Radar Sensing Group, University of Cape Town, South Africa
Email: john.collins@simconglobal.com

Mark Anderson, Darryl Owens and Darryl Bayliss
Department of Computing, Edge Hill University, UK
Email: {mark.anderson, darryl.owens, darryl.bayliss}@edgehill.ac.uk

David Gill
National Center for Atmospheric Research, Boulder, CO, USA
Email: gill@ucar.edu

***Abstract*— Powerful software tools are used by the aerospace and power generation communities to check codes for errors. The Software Validation Project at Edge Hill University (http://www.edgehill.ac.uk) in collaboration with SimCon Ltd (http://www.simconglobal.com) has applied these tools to WRF, the most widely used program for modeling climate and weather. Preliminary results have been generated. Although the WRF code has been found to be of a very high standard, some errors have been identified, both in the code and in the compilers used to build it. An aim of the project is to use CASE tools to correct some classes of error automatically. This paper discusses the classes of error that have been identified to date, presents the initial findings from the investigations performed by the project team.**

***Index Terms*— Quality Assurance; Automated Software Engineering; Error Removal**

I. Introduction

The successful validation and verification of climate models is an essential aspect of studies in and predictions of climate behaviour. The Weather Research and Forecasting Model (WRF) [1] developed under the aegis of the University Corporation for Atmospheric Research (UCAR) and the Unified Model (UM) [2] developed by the UK Meteorological Office are two major examples of such models.

Although the procedures used for quality assurance in the models are systematic and are well established in the software engineering world, they are unable to reveal the existence of many types of error. Furthermore, having found an error they are often unable quickly to identify and correct the source or sources of the error. Many software development tools are available to software engineers but the majority is targeted at recently developed languages and methodologies.

In contrast, climate models have been developed over many years and have stayed loyal to their original languages - overwhelmingly Fortran and C. WRF and UM are both sets of Fortran programs. In addition, the need to provide complex build mechanisms for a range of platforms, multiprocessor architectures and compilers has meant that the Fortran source code may not available in a form that software development tools supporting Fortran are able to analyse.

The Software Validation Project (SVP) at Edge Hill University aims to enhance the efforts of development teams typified by the maintainers of WRF and UM by applying powerful software tools used by the aerospace and power generation communities to check codes for errors. To this end the project team selected WRF for initial investigation since its source code and build procedures are publicly available. It then considered how one particular software development tool for Fortran, WinFPT, could analyse WRF and report on issues within the code that had been hidden from the WRF developers. This paper reports on the techniques used and the issues revealed.

The aim of this project is to perform a study of WRF, not in relation to the climatological model but in terms of quality assurance of the software. The WRF program is of considerable academic interest because of the modern Fortran style in which it is written. Major Fortran programs in aerospace, power generation, medical research and defence, are almost exclusively written in a style best described as extended Fortran 77. In contrast, WRF uses most of the features of (at least) Fortran 2003. The purpose of the project is to understand the implications for code quality and the types of error which occur when a modern Fortran dialect is utilised for a large scale implementation.

The objectives of the project are:

- to analyse WRF to identify errors;
- to demonstrate the complete absence of certain classes of error;
- to correct any errors identified, sometimes automatically;
- to evaluate the effects of the errors and of the corrections.

II. Existing Quality Assurance Procedures

There are quality assurance techniques that are currently used to overcome challenges facing High Performance Computing (HPC) specifically based in scientific software development [3][4][5]. One commonly occurring theme is the use of older programming languages due to the length of time these programs have been under development [5]. In this section the quality control procedures for two major Fortran based scientific software projects, WRF and UM, are briefly described and critiqued.

WRF Quality Assurance Procedures

The practices adopted by the WRF development teams aim to introduce measures for quality assurance throughout the design and build process. These include the development of coding standards to adopt within the model, along with identified use of version control tools [6].

To support the continual development of WRF, working groups have produced test suites and test procedures [7] to test the code. This has been supported by the formation of the Developmental Testbed Center (DTC) which has provided on-going support and incremental testing of WRF for reference configurations (RC) [8]. Evaluation procedures undertaken on releases of WRF include

1. Automated tests to capture coding errors which cause the compilation to fail
2. Manual tests conducted to evaluate the scientific output from the model for accuracy
3. Real-time tests for stability and integration checking
4. Pre- and post-processing tests for diagnostic and graphics systems
5. Architectural tests to evaluate builds on multiple architectures and/or compilers [9]

However, concerns may be raised regarding the tests which are performed in relation to the quality assurance of the code base. The automated tests for regression testing determine only whether the execution of the code completes successfully. The output generated by the code is not considered in this suite of tests. Therefore there may exist differences in the output of the modules following compilation and execution which are not identified following the successful completion of the automated tests.

Manual testing, on the other hand, is a slow and laborious process which exercises a much smaller number of well understood test cases. The impact upon cases which have not been tested is unknown. Potentially worse, the cases that are tested may produce results which appear to be more accurate but the execution paths that generate those results may be totally unexpected and lead to significant impact in other cases which have not been tested, or in future tests. The automated and manual tests are not guaranteed to uncover issues related to numerical drift. Also they would not uncover errors which were introduced in the compilation process related, not to the code base itself, but to the compilers used to build the model. WRF is designed not to be tightly coupled to machine architecture or compiler. Thus there exists an opportunity for different compilers to generate executables which produce dissimilar output from the model, a situation which is acknowledged by the DTC [8]. In both cases, automated relative debugging techniques would assist in identifying changes in between the outputs of different builds, and therefore the underlying causes may be resolved [10], Automated debugging processes described in this paper would enable the output from large number of test cases can be evaluated at multiple points within the execution of the test cases without significant intervention from the user.

UM Quality Assurance Procedures

The United Kingdom Meteorological Office (Met Office) has developed a long term strategy in order to address the challenges surrounding quality assurance of the code base. One major challenge faced is ‘the prediction challenge’; computational science does not have the predictive reliability of traditional methodologies [11]. To meet this challenge, the Met Office developed an approach of Flexible Configuration Management (FCM) to manage its Unified Model (UM) [12]. FCM is combined with a rigorous structure and processes that dictate the rest of the quality assurance procedures [13].

One such procedure which has been adopted within the QA strategy has been the comparison of visualisations of model output between implementations and executions of the model [13]. The underlying notion is that any differences between the visualization will indicate that an error has been introduced as revisions and updates have been applied to the model. However, whilst this may reveal differences between the executions of the different versions of the model, it is unlikely to reveal errors common to both versions. This is the same situation as arises in manual testing of the WRF model. Correct identification and interpretation of the visual differences and familiarity with the code are needed to remove the errors causing the differences. There is also the possibility that smaller, seemingly insignificant, errors at this stage in development could pass this stage of QA and cause problems in future developments.

Two further procedures are undertaken as part of the QA process which also fail to reveal errors that have been introduced into the model. These are the automatic checking for bit comparison between outputs of different runs, and the formal comparison of results from other

model implementations [13]. Neither would support the identification of numerical drift caused, for example, by a compiler choosing to keep different variables in registers because of changes to the code. While these comparisons can, in most cases, establish differences in implementations of models, both of these processes makes it difficult to track back from the differences identified to the source, or sources, of those differences.

III. THE CASE TOOL

In this project, WRF is analysed and re-engineered by a computer aided software engineering (CASE) tool. The tool used is WinFPT (http://www.simconglobal.com). WinFPT:

- reads the entire program like a compiler;
- analyses the program and data flow across all modules and sub-programs;
- identifies many classes of error and inconsistency;
- optionally re-engineers and re-writes the code to measure behaviour at run-time;
- optionally re-engineers and re-writes the code to correct some classes of error.

WinFPT carries out static analysis of the code. It can also instrument the code for some classes of dynamic analysis. For example, counters can be inserted for test coverage analysis to measure the number of times each statement is executed. Also code execution and the values of variables can be traced for relative debugging, exposing differences in the behaviours of different compilers.

IV. ANALYSING WRF

The WRF program is distributed as pre-processor source files, with the file name extensions ."F" and ."F90" (Upper case "F"). The build procedure generates code for specific parallel environments and compilers, and converts these files to standard Fortran files with the extension ".f90". The pre-processing is carried out by a special purpose C program, standard.c, and by the C language pre-processor, cpp. The textual changes made in pre-processing are extensive. The first pre-processing step strips all of the comments from the code, and cpp collapses white space characters between the tokens.

WinFPT cannot analyse or re-engineer the pre-processor source code. The analysis is carried out on the intermediate, compilable, ".f90" Fortran files. This does not affect the error detection, but it creates two difficulties if the re-engineered code is to be built and run. Firstly, the entire build procedure has to be re-written to start with Fortran files instead of pre-processor code. Secondly, any automatic corrections made by WinFPT are made on files from which the comments have already been stripped. The corrected code cannot, therefore, be reintroduced into the WRF distribution.

WRF Program Metrics

The code analysed contained 624,555 lines of Fortran which, as stated earlier, have been developed using a modern dialect of Fortran. Table 1, below, shows a comparison between WRF and a typical aerospace engineering code. "Radar" is the signal processing and instrument control of a tracking radar system.

The most important difference between the style of WRF and of the Radar code is in the use of Fortran modules. A third of the sub-programs, and all of the shared data in WRF, reside in modules. In the Radar code most of the sub-programs are written in separate files linked at the top level, and the shared data are in COMMON blocks written in include files. The disadvantage for the Radar code is that the fragmentary organisation of sub-programs and the need to control memory allocation in COMMON blocks provides significant scope for error. The advantage is that any routine can be changed, and the Radar program can be re-linked in 20 seconds. The corresponding time to change a routine in WRF is typically 20 minutes.

The most important difference between the style of WRF and of the Radar code is in the use of Fortran modules. A third of the sub-programs, and all of the shared data in WRF, reside in modules. In the Radar code most of the sub-programs are written in separate files linked at the top level, and the shared data are in COMMON blocks written in include files. The disadvantage for the Radar code is that the fragmentary organisation of sub-programs and the need to control memory allocation in COMMON blocks provides significant scope for error. The advantage is that any routine can be changed, and the Radar program can be re-linked in 20 seconds. The corresponding time to change a routine in WRF is typically 20 minutes.

The count of comments in WRF is distorted by the pre-processor standard.c, which strips the text of the comments and leaves blank lines. The count of include files is also distorted by the use of the c pre-processor #include directive. The analysis was carried out on code which was already pre-processed and the include files were therefore already inserted inline.

V. ERRORS IN WRF

The static analyses performed on WRF include checks for:

- inconsistent sub-program arguments;
- errors in addressing within COMMON blocks and EQUIVALENCE structures;
- objects forced to mis-aligned addresses;
- accidental loss of precision in expressions;
- variables read before they are initialised;
- unintended whole array assignments;
- failures in overloaded assignments of derived types;
- unreachable code sections

The results are described in the following sections.

Inconsistent Sub-program Arguments

A check is made that all actual sub-program arguments are consistent with the formal arguments in the sub-program declarations in:

Table 1. Comparison of WRF with a typical Aerospace Engineering Code

	WRFV3.4	Radar
Files		
Primary files	392	1589
Include files	3	1891
Code and comments		
Declaration lines	139724	49831
Executable lines	331732	100203
Total code lines	471456	150034
Comment text lines	72984	236033
Comment separator lines	19084	7942
Blank lines	61031	87402
Total comment lines	153099	331377
Total lines	624555	481411
Trailing comments	33014	27907
Words in comments	577601	662221
Program units		
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
Additional entries	0	0
Generic interfaces	56	0
Specific interfaces	93	0
Unresolved references (C/Assembler)	250	103

- data type;
- data kind (e.g. 4-byte or 8-byte REAL numbers);
- protocol (e.g. passed by reference, by value or by address and length);
- intent (i.e. whether input, output or both input and output);
- array bounds. Note that these should conform but need not be identical since sub-arrays may be passed.

A simple analysis shows 2353 occurrences of inconsistent arguments, affecting 375 different sub-programs in the Fortran code in WRF version 3.4. However, 1002 of these are situations where the shapes of arrays are re-mapped across the subroutine call-site. Several hundred more are data type inconsistencies where the data passed are simply moved to other processors or threads and the data type is not important. 449 inconsistencies are situations where the formal and actual arguments are of type CHARACTER, both have specified lengths and the lengths are different. This works without error on most, but not all, compilers.

Detailed analysis of the analysis shows surprisingly few situations where the inconsistency is likely to cause a problem. The difficulty is that there are so many harmless inconsistencies that it is difficult to find the genuine errors. The example shown above is unusual.

Errors in the intent of arguments are of two types. Sometimes a constant or an expression is passed as an argument into a sub-program and may be written to. An example is shown in Figure 1. Here, the actual argument is a Fortran parameter (a constant) but the formal argument, “fieldtype”, can be written to. Errors like this are unlikely to be serious. If the called routine attempts to write to the constant the program will probably crash. If the program does not crash, the write probably never takes place. This is unlikely to produce incorrect results. The second class of intent error, where the intent of the formal argument is declared incorrectly, is far more serious and is discussed below.

Errors in Intent Declarations

In Fortran, a sub-program argument may be declared to be INTENT (IN), INTENT (OUT) or INTENT (INOUT). These declarations are almost always optional. In making an intent declaration, the programmer asserts that an argument:

- is read by the sub-program, but never written: INTENT (IN);
- is written to in the sub-program but not read before it is written: INTENT (OUT);
- may be read before it is written and may be written to: INTENT (INOUT).

If an argument is declared INTENT(IN) the compiler need not (and perhaps should not) export the value of the argument when control returns from the sub-program. If an argument is declared INTENT(OUT) the compiler need not import the value of the argument into the sub-program when it is called.

Compilers compile sub-programs one at a time. Most compilers recognise simple intent errors if the intent assertion is violated directly in the sub-program code. They do not recognise an error if the argument is passed down into another sub-program which violates the assertion. The analysis tools track the intent and read-write status of every argument through the entire call-tree. There are three possible problems:

- the argument is declared INTENT (IN) and can be written to. This is always an error. There are 142 occurrences in WRF version 3.4
- the argument is declared INTENT (OUT) and is always read before it is written to. This is also always an error. There are 62 occurrences in WRF version 3.4.
- the argument is declared INTENT (OUT) and it is possible that it is read before it is written to. The issue here is that the program flow may be data dependent and unclear. There is a risk of error. There are 1,322 occurrences in WRF.

It is possible to correct all intent errors automatically. For example, if an INTENT (IN) argument is written to, the argument can be copied to a temporary variable on entry to the routine and the temporary used instead. The problem is in deciding what the programmer intended to happen. Should the correction honour the INTENT (IN) declaration, or should the INTENT declaration be changed to reflect the behaviour of the code? There is a strong probability that compilers always ignore intent declarations when compiling the data passing protocol of an argument, and the authors have verified this for two important compilers. The authors plan to correct the code by honouring the INTENT (IN) declarations and to test WRF to determine whether there is any change in behaviour.

Anomalies in Expressions

WRF version 3.4 contains 4,722 anomalies in arithmetic or character expressions. Most of these involve:

- loss of precision;
- testing of equality of REAL numbers
- truncation of character strings.

In WRF there are 235 occurences of loss of precision where exponents are single precision rather than double precision. There are also 940 occurrences of 8-byte REAL variables being assigned from 4-byte real values which results in 4-byte precision. It is common practice to compute a derivative to 4-byte precision and to integrate the result to 8-byte precision.

Switches are available in the WRF build procedure to promote all REAL objects to REAL*8. These may overcome the problem, but it is not clear that these switches change both the storage allocation and the numerical values stored on all systems.

There are 675 occurrences where real or complex values are tested for exact equality, as in the example in Figure 4 below. If this is to avoid a division by zero, it is likely that a tolerance should be introduced into the test. Again, all of these anomalies can be corrected automatically.

Variables which are read before they are initialized

Static analysis shows 322 variables which are read in the code before any values are written to them. This may be an underestimate because the analysis only shows situations where the program flow is unambiguous. However, in many cases the variables are reported as read because they are passed into sub-programs, and it is not certain that the sub-programs actually read them. Dynamic analysis, where the WRF program is run under different compilers, has revealed differences in behaviour caused by uninitialised variables, and a study is planned to investigate this rigorously.

Failures in Overloaded Assignments of Derived Types

Fortran supports the assignment of variables of derived types. If A and B are both of the same derived type, the statement “A = B” copies all of the components of B to A.

Fortran also allows the assignment of derived types to be overloaded. A subroutine is written to carry out a modified copy operation. The subroutine is declared to overload the assignment operator by an “INTERFACE ASSIGNMENT (=)” construct.

A problem with this language construct is that if any error is made in the scope or declaration of the “INTERFACEASSIGNMENT (=)” the overload fails and the variables of the derived type are simply copied silently and without any apparent error.

There is only one INTERFACE ASSIGNMENT (=) construct in WRF. It occurs in ESMF_time.F90. This is the only INTERFACE ASSIGNMENT (=) construct which the authors have encountered in the analysis of several tens of millions of lines of Fortran. The module

in which it is written has a global PRIVATE statement, and the interface is not declared to be public. It is therefore not exported from the module and the overloaded assignment sometimes fails. The situation is made worse by a bug in the Intel compiler which causes the interface to be exported when the subroutine used to make the copy is exported. Therefore, WRF works as intended when compiled by the Intel compiler and as written when compiled by the gnu (and probably every other) compiler.

The authors suggest that a construct which is used only once in many millions of lines of code, and which has a 100% failure record, should probably be avoided.

Unreachable Code

There are 552 unreachable sections of code in the version of WRF which was analysed. Most of these may be deliberate. The analysis tools can only be used on the intermediate Fortran files produced during the build procedure, and these files are pre-processed for a specific architecture. Some of the pre-processing inserts jumps around sections of the code.

The number of unreachable sections is sufficiently small that a manual analysis can be carried out, and this will be done in the future.

Errors which do not occur in WRF

Certain classes of error do not occur in WRF. The analyses carried out show:

- Errors in COMMON blocks: Programs written in an extended Fortran 77 style usually contain COMMON blocks, sometimes with many thousands of variables. Errors in the organisation of COMMON blocks are common. WRF has one COMMON block which contains only 6 scalar variables. There are no errors in COMMON blocks in WRF.
- Mis-aligned Addresses: In extended Fortran 77 code, COMMON blocks, EQUIVALENCE statements, sequence derived type and structure constructs sometimes force variables to mis-aligned addresses where, for example, a REAL*4 object does not start on an address which is a multiple of 4. This causes inefficiency, and occasionally compiler errors. WRF contains only 77 EQUIVALENCE statements and only one COMMON block. The code contains no mis-aligned objects.
- Unintended Whole Array assignments: Fortran allows assignments of the form "A = x" where A is an array and x is a scalar variable. All elements of A receive the value of x. This is dangerous. The authors have encountered many examples of the form:

```
DO i=1,10
   A = B(i)
ENDDO
```

where the programmer intended to write:

```
DO i=1,10
   A(i) = B(i)
ENDDO
```

There are no errors of this type in WRF.

VI. Comparison of the Errors in WRF with a typical Aerospace Code

It is a reasonable assumption that the modern style of WRF should have protected the code from many classes of error. A comparison with the Radar code is shown in Table 2 below. WRF is larger than the Radar code and the error counts are therefore shown in the right-hand columns of the table as the number of anomalies per thousand code lines.

In WRF, there are 150 occurrences of constants or expressions which are passed to routines which could write to them. The radar code has only 2. The difference is probably a consequence of the very large number of sub-program arguments used in WRF. As noted above, it is unlikely that WRF actually writes to these arguments.

WRF has 142 INTENT (IN) errors and 62 confirmed INTENT (OUT) errors. The Radar code has none because it contains no INTENT statements. It is not clear that the use of INTENT statements has contributed to the quality of the WRF code.

The Radar code contains hard-coded array references out of bounds. WRF does not. Some of the out of bounds references in the Radar code are deliberate, and make use of the tightly controlled memory mapping of the COMMON blocks. Some are accidental, but, because of the use of COMMON blocks, at least they are consistent errors.

The remaining anomalies, uninitialised variables, inconsistent arguments and anomalies in expressions show a similar pattern in the two codes. This is surprising because the WRF code is far more readable than the Radar code and the use of modules provides for better checking of interfaces and better control of scope.

VII. Correcting and Testing the Code

A small number of the anomalies detected can safely be corrected by hand. The correction of the failed overloaded assignment, for example, is a one line change. In most cases, the correction must be automated. It is not practical to correct 4722 anomalies in expressions manually, and the risk of injecting new errors would be significant.

Currently, the CASE tools can correct a proportion of the expression anomalies and all of the INTENT (IN) errors. They must be extended to handle the remaining expression anomalies and the INTENT (OUT) errors. Very few of the mis-matched sub-program arguments actually require correction. It is hoped that those which do can be identified and corrected by hand,

A necessary first step in making the corrections is to modify the build procedure so as to rebuild from compilable Fortran sources. The intention is to encapsulate the changes made by the pre-processors so that the majority of files do not require pre-processing. This would make it possible to re-engineer the code once before a series of builds instead of re-engineering it for every separate build as part of the build process.

Testing of the changes requires the development of a regression test suite. A coverage analysis of the WRF

Ideal cases showed that they visited only 18% of the executable Fortran statements. Test material is now available which exercises over 50% of the code and this will be extended.

The problem in testing is in comparing the results from different runs. Almost any change in the code is likely to inject numerical drift, where small changes are cumulated so that the results of two runs show large numbers of differences. A procedure has been developed to eliminate numerical drift in single processor runs, and it is hoped to extend this procedure for multi-processor runs.

VIII. Further Error Checks

Two further error checks have been identified for further investigation as they could affect the stability and reproducibility of WRF runs. They are:

- a check for the use of optional arguments to sub-programs. The issue is that optional arguments could be passed down to routines without a check that they are present. If this occurs, the value passed might be that from an earlier call to the routine, or might be null. A null argument could crash the program;
- a check for race conditions in multi-processor and multi-threaded runs.

IX. Conclusions

Initial investigation making use of the techniques described has enabled a number of issues to be discovered. These issues can be related to coding in the model and the compilers used to build the code. The paper discusses three findings from the experiments that have been performed. These represent important validation of this technique, and the intention of the project is to apply the technique to include broader coverage of the WRF model code

The WRF version 3.4 program contains a little over 10,000 known coding anomalies. There are 624,555 code lines. Therefore there is approximately 1 anomaly per 60 lines (or about 2% of the codebase). These are anomalies in the code, not in the underlying climatological model.

A large proportion of the anomalies can be corrected automatically and work is under way to complete this task. A necessary first step is a revision of the build procedure. A test suite and test analysis tools are under development to assess the implications of the anomalies identified and to test WRF before any permanent changes are made.

The anomalies found were mostly invisible to the existing techniques or could only be revealed by considerable effort and knowledge of the source code. In contrast, the techniques described in this paper revealed the anomalies quickly and in the majority of cases were easily remedied. Where the reasons for anomalies were not obvious such as the failures in overloaded assignments of derived types, analysis of the divergence of control paths followed by the instrumented models revealed bugs in the program, compiler and the Fortran language itself which would have been virtually impossible to detect otherwise.

Acknowledgment

The authors wish to thank Prof. W. J Gutowski jr. Iowa State University and Prof. Bruce Hewitson, University of Cape Town for their help and support.

References

[1] J. Michalakes, Dudhia, J., Gill, D., Henderson, T., Klemp, J., Skamarock, W., and Wang, W.: The Weather Research and Forecast Model: Software Architecture and Performance. Proceedings of the Eleventh ECMWF Workshop on the Use of High Performance Computing in Meteorology. Eds. Walter Zwieflhofer and George Mozdzynski. World Scientific, pp 156 – 168, (2005)

[2] M.J.P. Cullen, "The unified forecast/climate model". Meteorological Magazine (UK), Vol. 122, No. 1499, pp 81-94, (1993)

[3] D. Abramson, Foster, I., Michalakes, J. and Sosic, R. Relative debugging and its application to the development of large numerical models. In Proceedings of the 1995 ACM/IEEE conference on Supercomputing, ACM, New York, Article 51.(1995)

[4] D. Abramson, Foster, I., Michalakes, J. and Sosic R., "Relative Debugging: A New Methodology for Debugging Scientific Applications", Communications of the Association for Computing Machinery (CACM), Vol 39, No 11, pp 67 - 77, Nov (1996).

[5] D. Abramson and Sosic, R. A debugging and testing tool for supporting software evolution. Automated Software Engineering: An International Journal, 3(3/4):369–390, August (1996).

[6] J. Michalakes, Middlecoff, J., Black, T., Xue, M., Sedlacek, D., Benslay, J., Holt, T. and Balaji, V. 'Coding standards and conventions for the physics packages'. Available online at http://www.mmm.ucar.edu/wrf/WG2/WRF_conventions.html [accessed 14/09/12]

[7] C. Davis, Mahoney, J., Lin, Y., Carr, F., Craig, B., Swenson, M., Lerner, J. and Trier, S.'WRF Testing Plan'. Available online at http://wrf-model.org/development/group/WG7/Pre-reference_testing3.doc [Accessed 14/09/12]

[8] "Developmental Testbed Center Home". http://www.dtcenter.org/ [Accessed 15/09/12]

[9] "WRF Model Version 3.1: Testing". http://www.mmm.ucar.edu/wrf/users/wrfv3.1/testing.html [Accessed 15/09/12]

[10] D. Abramson, Foster, I., Michalakes, J. and Sosic R., "Relative Debugging: A New Methodology for Debugging Scientific Applications", Communications of the Association for Computing Machinery (CACM), Vol 39, No 11, pp 67 - 77, Nov (1996).

[11] D. Post, "The Coming Crisis in Computational Sci- ence," keynote, IEEE Int'l Conf. High-Performance Computer Architecture: Workshop on Productivity and Performance in High-End Computing, 2004; www.tgc.com/hpcwire/hpcwireWWW/04/0319/107234.html.

[12] D. Matthews, Wilson, G.V. and Easterbrook, S.M., "Configuration Management for Large-Scale Scientific Computing at the UK Met Office," Computing in Science & Eng., vol. 10, no. 6, 2008, pp. 56–65.

[13] S.M. Easterbrook, Johns T.C., Engineering the Software for Understanding Climate Change, Computing In Science and Engineering, Nov (2009)