Towards Test Driven Development for Computational Science with pFUnit

Michael Rilee[1,2] and Thomas Clune[1]

[1]NASA Goddard Space Flight Center
[2]Rilee Systems Technologies LLC

***Abstract*—Developers working in Computational Science & Engineering (CSE)/High Performance Computing (HPC) must contend with constant change due to advances in computing technology and science. Test Driven Development (TDD) is a methodology that mitigates software development risks due to change at the cost of adding comprehensive and continuous testing to the development process. Testing frameworks tailored for CSE/HPC, like pFUnit, can lower the barriers to such testing, yet CSE software faces unique constraints foreign to the broader software engineering community. Effective testing of numerical software requires a comprehensive suite of oracles, i.e., use cases with known answers, as well as robust estimates for the unavoidable numerical errors associated with implementation with finite-precision arithmetic. At first glance these concerns often seem exceedingly challenging or even insurmountable for real-world scientific applications. However, we argue that this common perception is incorrect and driven by (1) a conflation between model *validation* and software *verification* and (2) the general tendency in the scientific community to develop relatively coarse-grained, large procedures that compound numerous algorithmic steps. We believe TDD can be applied routinely to numerical software if developers pursue fine-grained implementations that permit testing, neatly side-stepping concerns about needing nontrivial oracles as well as the accumulation of errors. We present an example of a successful, complex legacy CSE/HPC code whose development process shares some aspects with TDD, which we contrast with current and potential capabilities. A mix of our proposed methodology and framework support should enable everyday use of TDD by CSE-expert developers.**

I. Introduction

Computational Science and Engineering (CSE) software development and testing has adapted to pressures rarely found in the broader context of Software Engineering (SE) [7]. As well described elsewhere, the scientific result is the goal of CSE effort [4]. As results are produced, or not, scientific understanding evolves and software requirements change [22], [23]. In contrast to more general software usage, improvements in computational capability are allocated to improving the results, rather than achieving the same result more cheaply or more quickly. Adapting to these unique challenges, CSE domain experts have evolved software development styles with specialized approaches to verification and validation and methodologies that can resemble Agile and open source [13], [14].

Many important scientific applications have lifetimes that span several decades; evolving incrementally under pressures for improved physical fidelity and/or exploiting new computational environments. Because the goal is to advance science given limited resources, a major concern is the portability of these codes and the reliability of their results as new platforms are brought to bear on important scientific problems. Validation of a code's results against experiments, observations, or scientists' understanding of reality is widely recognized as crucial to scientific advance. On the other hand, *verification* that the software implementation accurately captures the scientific understanding of the expert developers involved is often assumed implicitly.

In the meantime, the broader SE community has been able to address other, but shared, issues associated with software development. Testing plays an important role for both CSE and SE, but with different emphases. In CSE the focus is often on validation of the "final" results of the calculations with an eye towards scientific advance, as mentioned above. This is reminiscent of "acceptance testing," in the sense of "what impact does the code's result have on our science"? Calculation results may continuously force the scientist developer to change

their theoretical model or approach, which to those with a more conventional SE background means the software requirements continuously change [22], [23], [7], [4]. The scientist developer may simply see that their expectations for the results of the scientific calculation haven't been met, requiring a change of ideas or approach.

Unit testing frameworks have made it possible to include software system requirements (and constraints on properties leading to them) as part of the software system itself as unit tests. In this context, unit tests are a powerful tool aiding the construction of new code and the analysis and revision of legacy code [15]. Object technologies have reduced the workload required to implement such testing, as software frameworks for developing harnesses of these tests have been developed [10], [12], [8]. Build process technologies have allowed these harnesses to be tightly integrated with software development, covering the code with tests executed at every compilation. As a result, verification of the software against the current understanding of system requirements (as represented by the harness of unit tests) is performed at every step of software development. With the advent of unit testing frameworks more attuned to the CSE/HPC development environment, tools are becoming available that actually accommodate CSE development needs.

Test Driven Development (TDD) simultaneously co-evolves the harness of unit tests and the software under development [5]. As developers design and build their system, their understanding of the system and its requirements grows, which is then cycled back into the growing software system, a process that should be familiar to the scientist developer. Developers alternate between enhancing and extending the test harness and the system code, potentially rapidly cycling (< 10 minutes) through this process to implement a test and then the code that allows the system to pass. Through this rapid analysis and experimentation, requirement, design, and implementation problems can be detected, isolated, and corrected early. The emphasis on early analysis and design improves software quality and modularity, and the process provides more insight into how software development is progressing towards system goals. Thus TDD has many attractive features for the science code developer, but science codes pose unique challenges. Still, our initial forays into TDD for science software are quite encouraging [9].

In this paper, we go into several specific needs of current HPC CSE software development, namely distributed parallelism and numerics, that could be helped by the infusion of testing strategies developed for more conventional SE. We describe a successful example of a complex CSE/HPC code in which testing played an important but not leading role, and how that testing compares to what is possible today. We point out gaps between the current state-of-the-art in fine grained, low-level testing of CSE code, pFUnit, and a usable TDD framework for CSE, as well as our approach to closing these gaps.

II. TDD for Science Codes

The every day use of TDD by CSE developers requires a mix of development methodology enhancements and framework support. A key change in methodology is to more strongly distinguish science theory from computation and recognize that verification between theory and computation (implementation) is important. Verification is best built on partitions of the computation (code units) whose required behavior can be characterized and understood. A powerful way to express this characterization is with synthetic inputs and their consequent solidly known expected outputs, to a required level of numerical tolerance. In the best cases, we can require that behavior of code units meets expectations that hold unavoidable error to within machine precision. In this way, we can trace (verify) the behavior of code units to the underlying scientific understanding, i.e., the science, the theory, approximations, etc., improving the trustworthiness of a scientific model's expression on HPC platforms.

In TDD, the verification that a science code is a correct re-expression of scientific understanding is aided by the software system's set of unit tests. This does not eliminate the judgment of the scientific software developers from the ongoing analysis of their work, but captures and automates some of their analyses. The cost of capturing and implementing this analysis is mitigated by the use of software frameworks that ease test development and use, such as pFUnit, described below [11], [8], [9].

The verification mentioned above is best performed with fine-grained tests whose behavior is relatively straightforward to characterize. A challenge for fine-grained testing is that code units in (large-scale) science software often bundle many behaviors in coarse units that have many dependencies (especially as expressed in Fortran), making it difficult to partition the code into isolated units for testing. We believe the interactions of complex networks of such dependencies are a major contributor to the difficulty of providing oracles for checking coarse code unit behaviors. In TDD, mock technologies enhance isolation by enabling such dependencies to be replaced with configurable software that records and drives the behavior of the code unit being tested. This replacement would allow some measure of fine-grained testing to be recovered even for coarse-grained procedures. Refactoring coarse grained code while gradually building up a supporting harness of fine-grained unit tests seems to be a powerful strategy for code development and analysis. Adding these capabilities to a unit testing framework like pFUnit and providing them to CSE developers via an Integrated Development Environment (IDE), e.g., Photran [2], [1], would make it easy for developers to explore their codes' design and implementation via TDD, a path we are exploring in collaboration with the R&D firm, Tech-X.

III. Testing During Science Code Development: PARAMESH

To explore these concepts, we recall a medium-sized, but complex, computational science application framework that illustrates a number of the issues involved. PARAMESH is a Fortran 90-based support framework for parallelizing serial calculations involving logically cartesian, block-structured grids and providing adaptive mesh refinement in development $\sim$1998-2008, during which time it went through four major versions [21], [19], [20], [17]. Successful as infrastructure or middle ware for science developers, from 1998-2006 at least 60 papers in a wide variety of disciplines were published using results from codes built on PARAMESH [17].

Most science calculations at the time were serial or vector-parallel and not expressed in object-oriented terms, so the code provides a subroutine library for the domain expert user, who is also expected to modify the source if necessary. PARAMESH provides a distributed data structure with a logically cartesian, block-structured geometry and a variety of communication, work-distribution, and grid management and reconfiguration services. Science data could be associated with edges, faces, vertices, or volumes in the grid, each with its own communication needs and supporting a wide variety of calculations. As an adaptive mesh, the geometry could be subdivided into different levels of refinement with data values set by interpolation operators (prolongation to finer and restriction to coarser) as needed. The calculation part of the code, the "science algorithm," thus provided scientists a familiar-looking cartesian geometry, which eased the porting of serial code. Calculations could be expressed as one or more subroutines that operated on a "block," which PARAMESH invoked in parallel across the different levels of grid refinement. This complex code is highly configurable with many preprocessor options to take into account different computing platforms, compilers, as well as different communication and grid management and interpolation schemes.

A. Tests & Exploratory Development

Testing played a crucial role in the development of this complex functionality, albeit without the benefit of unit tests, a testing framework, or even an explicit testing policy. Tests were typically implemented as standalone programs that were developed and run as new functionality was implemented and when significant revisions were made. Tests, or rather, demonstrations of new functionality through appropriate means, whether as index tables or visualizations, were important for "debugging" and verification. Sometimes expected results were encoded in tests and compared with those found during code execution at other times, while at other times analysis was based on scientific visualizations. The bookkeeping and communication aspects of grid management naturally lend themselves to unit testing, which the existing example tests resemble.

More pertinent to scientific work, PARAMESH provided both new approximation functionality with its dynamically adaptive grid as well as the ability to take advantage of large parallel computers coming online at the time. This allowed science code devel-

opers to explore new ways to improve the quality of their science models. Temporal and spatial resolution could be dynamically adapted to the calculation encouraging exploration into higher-order mathematical approximations as well as higher dimensional representations of phenomena being studied. These explorations often did not mesh precisely with the developer preconceptions, driving revision and code evolution.

One example of how science developments drive code development was the application of PARAMESH to the problem of the terrestrial magnetosphere. Obvious, unphysical artifacts in magnetohydrodynamic (MHD) simulations of the terrestrial magnetosphere built on top of PARAMESH led to many revisions of the underlying scientific model, approximations, and algorithms, some of which required new functions to be added to PARAMESH. One key feature involved the unphysical creation of flows generated in the direction of the Earth's magnetic field traced to an unphysical non-zero divergence in the simulated magnetic field, e.g., [3]. Eliminating this divergence to any degree of accuracy places special requirements on the temporal evolution scheme of the numerical simulation. Even more critically the spatial (differential) operators determining field properties and evolution affects both the scientists' numerical model and PARAMESH's spatial support infrastructure.

B. Tests & Extension

Thus we see in this brief example a great deal of complexity of interaction that does not arise in more typical non-science software settings. The foundation of the code is the bookkeeping layer that manages the distributed data structure for grid geometries, science data, and their relationships. On top of this are the spatial and temporal numerical operators like interpolations and gradients, as with the zero-divergence work. Using these operators, scientific models are built that produce the scientific results we are after, not to mention the integration of customized scientific visualization graphics for debugging (verification) and results analysis (a validation step). Even at this level of detail, each one of these layers has its own complexity associated with implementing part of a science code. Unit tests are an excellent way to define and check that the components of each layer behave as required.

Lacking a unit testing framework, PARAMESH developers essentially developed their own tests, some of which had some persistence during the project becoming a suite demonstrating essential capability. Implementation, test, and debug was a painstaking process, and though there was close communication between developers as the code progressed, many of the tests were not usually shared by their developers. If a code change by one developer touched on an area covered by a certain test written by another, one generally asked the test-writer to run the test again because they were most familiar its building, execution, and the interpretation of its results. While effective in a group of expert, co-located developers, the lack of automation, persistence, and sharing limits the amount of testing and test configurations examined, reducing test value and impact. This is particularly a problem when non-PARAMESH-expert users are extending the code and integrating their own science models. It would be much better for these end-users to be able to extend a test harness that is tightly coupled to the build process itself, particularly since access to the original developers and their "personal" test suites and expertise can be impractical.

C. Unit Tests & Test Driven Development

Unit tests address many of the development concerns arising in the process described above. The great configurability and continuous exploration and analysis of science code development is supported by unit testing frameworks' provision of fixtures for expressing and iterating through configurations. The tests themselves become persistent, shared artifacts that describe both intent and outline troublesome areas of code development as well. Without such reification, the goals and knowledge embedded in the science code are implicit and ephemeral as the original developers' memory of design and coding issues fades. Fine-grained, quickly executing tests become a fundamental part of the build process, dramatically increasing the coverage, precision, and quality of information the tests provide. Shared with system developers and users who are extending the code, applying it to their scientific models, the test harness becomes a net that catches issues

arising from tests over a much greater set of code configurations and uses.

TDD organizes these efforts, recognizing that the test harness itself represents an understanding or model of the software system's functionality, which becomes an executable blueprint for the system itself. The evolution of functions, software configurations, or even experimental ideas can be expressed via the test harness. Verifying the successful transfer of theoretical constructs (e.g., approximations, numerical constraints, scientific phenomena or mechanisms) into software can be expressed, code unit by code unit, with unit tests. The experience gained through these tests (experimental verifications that theory has been properly expressed as software) naturally affects developers' understanding of the system. These tests drive changes in system design to better support the expression of theory in software, as well as to support re-examination of the theory and its expression as a calculation, without regard to software. For example, in reality the divergence of the magnetic field is zero, which is true in MHD too. In the theoretical domain, one may approach a given scientific problem involving magnetic fields by choosing different representations of the physical quantities, simplification, or solving an arguably related model problem, e.g., like discretizing a continuous problem for numerical calculation. Generally we may have estimates about how one approach to a scientific answer may be better than another, e.g., should we express the magnetic field directly or via its vector potential? Yet since the science is often complex and at the edge of our understanding, we may not know how the (theoretical or software) system may behave as a whole.

We do know how the components or steps of an approach to a solution should work. That is, with sufficient code isolation, operating with known (e.g., synthetic) inputs in a specified environment, we should be able to specify the output of any particular step in a calculation and hold its expression in software to that standard to an appropriate tolerance, in most cases *machine*-ϵ. While this error may aggregate over multiple steps of the calculation during the code's execution, its implementation will have been piecewise verified to machine accuracy.

Returning to our example, a particular approximation to the divergence of a field on a discretized mesh can be tested in a number of synthetic situations to verify the divergence calculation's representation in software, essentially to machine precision. This sort of test is an especially important part of analyzing and comparing the numerical quality of different approaches to divergenceless fields, e.g., at least three different methods of divergenceless prolongation to finer grids have been experimented with in PARAMESH [18]. While the different steps in each of these methods lead to different error dynamics, we claim each step is verifiable to machine accuracy. Unit tests associated with such changes would keep a record of where the research and development has been, providing a foundation for new tests and improvements.

D. Code Isolation for Testing

The high degree of code isolation enabling piecewise verification may be impeded by the wide and deep pool of dependencies from which numeric code can draw. Such code units often depend on calls or references to external functionality or data, which may be expensive or hard to characterize. In the broader SE community, object technologies have aided the development of methods to mock up such dependencies and many frameworks exist to help automate the generation of mocks for testing. Mocks are reconfigurable software constructs that replace those dependencies with stimulation and diagnostic instruments, testing and recording the behavior of the code in which they're placed.

We are in the process of extending pFUnit to include a suite of services that support the *use* of mocks in Fortran code. Yet implementing application-specific mocks remains a manual, time-consuming effort as they must replicate the interface of the original dependency, which may change as the original code evolves. Automating the creation of mocks in Fortran is quite difficult due to the lack of language features for introspection and templating. In pFUnit, we are circumventing Fortran's weakness in this area via python-based preprocessing and code generation, though we are continuing to look for other options. Additionally, we are pursuing technologies to capture and express interface information for the development of mocks

to improve the isolation and coverage of tests built on pFUnit.

E. Extending Usefulness via TDD

While PARAMESH is a success at providing parallel computing capability to a certain class of science codes, there are a number of factors that hinder its continued development and use. It is designed to be used and extended by a domain expert user working within the context of Fortran 90. Therefore its API provides a degree of modularity and abstraction, easing the use of the code, hiding a great deal of complexity underneath the hood. In particular, the array index conventions used are particularly complex, and great deal of the bookkeeping is done via carefully constructed array argument index ranges, which are difficult to modify consistently throughout the code. A good test suite is provided, which provides some of the same benefits that a more extensive harness of unit tests would provide. These tests allow for some error detection and isolation, but not nearly as much as would be possible with fine grained testing enabled by a unit testing framework enhanced with mocking. More detailed analysis is left as an exercise for the domain expert developer.

Again, we note that a great deal of testing and analysis went into the development (and debugging) of PARAMESH functionality, much of which was not saved. Implementing a new test was essentially like implementing a new application on PARAMESH, so these tended to exercise extended groups of behaviors at a fairly high level. A unit test framework would have allowed the more important ephemera to be retained, which would help the code be adapted to new and updated platforms. PARAMESH is well documented and commented, but lacks tests that illustrate design decisions pointing the way to how the code might be refactored. Example code is provided to help users extend PARAMESH into their own domain, but the tools to revise and update the code itself are limited.

Advances in object orientation in Fortran might be applied to help broaden the range of operators and algorithms that could be built onto PARAMESH, easing their use and experimentation, and extending the code's usefulness. The current data vectors defined on the grids with an abstract type and associated operators or advanced numerical needs like the Div-B problem mentioned above could be aided by using object extension and operator overloading. These would provide the expert developer the opportunity to write code that looks more like the theoretical expressions. Object-oriented techniques could allow functions like geometric intersections and unions to be implemented in a way that shields the user from referring to array indices and the complexities of distributed parallel data management, but these would entail widespread changes to the code and much analysis. A TDD approach would ease analysis and debugging by starting with a harness of unit tests, mocking dependencies to isolate the code units, and then co-evolving the unit tests during the code's renovation.

In a real sense, PARAMESH is an example of the benefits of a relatively fine-grained approach to structuring code. Although current technologies support a much finer-grained approach to testing than was originally feasible for PARAMESH, the focused behaviors of its large library of procedures are arguably concise and orthogonal. Ephemeral tests and print statements would have properties similar to a fine-grained test harness when active for limited periods of time during development. Built with the intention to be extended, the code encourages these practices to be continued during adaptation to an end-user's domain, though PARAMESH supports any code, however coarsely structured, that appropriately interacts through its interface of library support procedures.

IV. pFUnit: Supporting TDD for Science

pFUnit, the parallel Fortran Unit testing framework, is a software testing framework that is highly tailored to the needs of the CSE/HPC community and is well-suited for the use of TDD. The framework is implemented in Fortran, but the basic design is otherwise quite similar to many other so-called xUnit testing frameworks (e.g., JUnit, pyUnit, etc.) [6], [16]. As with those frameworks, pFUnit enables developers to readily create unit tests, collect them into test suites, and routinely execute those suites to detect any failures as they arise. Beyond the obvious implied capability to work with Fortran, pFUnit includes extensive support for (1) multidimensional

arrays, (2) floating point (FP) data, (3) parallelism via MPI and OpenMP, and (4) parametrized tests.

A major element of most testing frameworks is a suite of "asserts" that express the intent of tests, generally in the form of checking the equality of two expressions. While most frameworks have some limited support for comparing FP data and one-dimensional arrays, pFUnit supports comparison of single/double precision quantities, with an optional tolerance, for arrays up through 5-dimensional (or more if supported by the compiler). Tolerance can either be absolute or relative, and similar support is included for complex numbers. Other niceties include the ability to test for infinity and NaN.

Testing of parallel software raises a number of unique issues including the need to test the same procedure at various PE counts, identification of which processes have detected failures, and detecting deadlocks. The pFUnit MpiTestCase class is a container for user-defined unit tests that automatically generates a new MPI subcommunicator for each requested process count. Any exceptions are labeled according to process rank and PE count and then gathered to the root process. Users can thereby easily exercise their test logic across a variety of scenarios with relatively little effort.

MpiTestCase is actually only a special subclass of the more general ParameterizedTestCase that allows users to exercise a unit-test across a user-defined collection of parameters. This can be extremely valuable in scientific applications where functionality is often parameterized (e.g., boundary conditions, interpolation order, stencil-size).

V. Applying TDD to the Challenge

In this paper, we have anecdotally described some of the development and testing associated with a code that was intended to be extended to aid the parallelization of legacy applications, but developed before unit testing frameworks and object support in Fortran had matured. Like other CSE/HPC codes, it is highly configurable, portable, and was modified a great deal as calculations were performed and science models adapted. It provides intricate bookkeeping services and must support calculations involving subtle numeric issues from the demands of modeling continuous systems as discrete ones. Extensively tested for correct behavior and performance, the code's most important function tests are provided to the end users for verifying correct compilation.

Many tests, and the concerns that drove them, are lost to time. Still, it serves as a positive example of the benefits to development of being a finely structured code, with a large number of concise, focused procedures. The code is complex, but it provides a straightforward and extensive API attuned to its target audience. Documentation and examples are provided to aid the expert user adapting the code to their calculation. In the context of relating CSE to more conventional computing, it shows complexities that may be found in conventional software engineering (e.g., the distributed data structure). Others are unique to CSE/HPC, namely the tight coupling of computation across the distributed computer, the continuous change driven by scientists' changing understanding, and subtle numerical issues that dramatically affect result quality. The regard for speed is also different in CSE/HPC, since inefficient use of computational bandwidth yields poorer quality results for the same highly sought-after resources.

Finally, with the primacy of the scientific results, which progress incrementally, there are strong drivers to maintain backward compatibility to retain the existing, understood if not trusted, code base. Great emphasis is placed on the comparison of CSE code results against measurements or observations of physical reality, a validation step. Conversely, verifying the expression of theoretical constructs in software plays a subordinate role as theory and computation are conflated [4]. It is not surprising that developers in CSE and broader SE have adapted differently to their environmental drivers.

Technologies and methodologies pioneered outside of CSE/HPC are maturing to the point where they can provide value for acceptable costs. Support for object orientation is improving in Fortran, easing the infusion of software techniques using them. pFUnit, one of several unit testing frameworks implemented in Fortran, is inspired and patterned after JUnit, but is tailored to the CSE/HPC environment, making it easy to develop test harnesses for CSE code. A nascent mock services capability points towards the capability to better isolate code units, while specifying inputs, and monitoring behaviors. Proposed improvements such as the automated generation of mocks will make it simple to use test

harnesses as analysis tools, models of expectations for the system being evolved. Such framework capabilities, provided to domain expert developers in easy-to-use, easy-to-learn, configurable workflows via IDEs will allow scientists to codify their expectations as fine-grained test harnesses co-evolving with the science software itself.

Fine-grained tests, supported by mocks, have the scientific benefit of being easier to understand, numerically easier to characterize, and readily verified against theoretical understanding. Coarse-grained code requires a more careful treatment than we have space for here and will be dealt with in a future paper. Yet extensive use of mocks to replace networks of dependencies in coarse-grained code units opens up the possibility of verifying the glue code holding them together. When bringing very large projects into a test harness and TDD, it's likely that the code will be incrementally be brought under unit testing as a major refactoring effort, leading to a mix of coarse and finely structured code and tests.

PARAMESH developers continuously tested and analyzed their code during development, changing their approach and fixing problems as their expectations and understanding were informed by their repeated tests. It was, however, a labor intensive effort aided by maintaining a co-located team of expert developers, and much of their thought and analysis is now implicit in the code itself. By making such knowledge explicit, Test Driven Development can greatly enhance the productivity of CSE software development and maintenance, enabling codes of greater capability and complexity to be more rapidly and confidently adapted as computational platforms and science progress.

REFERENCES

[1] Parallel tools platform. http://www.eclipse.org/ptp/. The Eclipse Foundation. Accessed: 2014-07-08.

[2] Photran - an integrated development environment and refactoring tool for fortran. http://www.eclipse.org/photran/. The Eclipse Foundation. Accessed: 2014-06-29.

[3] Dinshaw S Balsara. Divergence-Free Adaptive Mesh Refinement for Magnetohydrodynamics. *Journal of Computational Physics*, 174(2):614–648, December 2001.

[4] V.R. Basili, J.C. Carver, D. Cruzes, L.M. Hochstein, J.K. Hollingsworth, F. Shull, and M.V. Zelkowitz. Understanding the high-performance-computing community: A software engineer's perspective. *Software, IEEE*, 25(4):29–36, July 2008.

[5] K. Beck. *Test Driven Development: By Example*. Addison-Wesley Professional. Pearson Education, Inc., Boston, MA, 2003.

[6] K. Beck and E. Gamma. JUnit: A cook's tour. *Java Report*, 4(5):27–38, May 1999.

[7] J C Carver, R P Kendall, S E Squires, and D E Post. Software Development Environments for Scientific and Engineering Software: A Series of Case Studies. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 550–559. IEEE, 2007.

[8] T. Clune and M. Rilee. pFUnit 3.0 - a unit testing framework for parallel fortran software. Technical report, 2014. In preparation.

[9] T. Clune and M. Rilee. Testing as an essential process for developing and maintaining scientific software. 2014. Submitted.

[10] T. Clune, B. Womack, B. Foote, and J. Overbey. Tools, trends and techniques for developing scientific software. In *High Performance Computing 2006*. European Centre for Medium-Range Weather Forecasts (ECMWF), November 2006. Accessed: 2014-06-23.

[11] T. L. Clune. pFUnit - unit testing framework for fortran. "http://pfunit.sourceforge.org". Accessed: 2014-07-08.

[12] T. L. Clune and R.B. Rood. Software testing and verification in climate model development. *Software, IEEE*, 28(6):49–55, Nov 2011.

[13] S. M. Easterbrook. Do Over or Make Do? Climate Models as a Software Development Challenge (Invited). *AGU Fall Meeting Abstracts*, page B1, December 2010.

[14] S M Easterbrook and Timothy C Johns. Engineering the Software for Understanding Climate Change. *Computing in Science & Engineering*, 11(6):65–74, 2009.

[15] M. Feathers. *Working Effectively with Legacy Software*. Prentice Hall. Pearson Education, Inc., Upper Saddle River, NJ, 2005.

[16] E. Gamma and K. Beck. PyUnit - the standard unit testing framework for python. "http://pyunit.sourceforge.org".

[17] P. MacNeice and K. Olson. Paramesh adaptive mesh refinement. http://www.physics.drexel.edu/~olson/paramesh-doc/Users_manual/amr.html, 2008. Accessed: 2014-08-29.

[18] P. MacNeice and K. Olson. Paramesh adaptive mesh refinement. http://www.physics.drexel.edu/~olson/paramesh-doc/Users_manual/amr_users_guide.html#divergence, 2008. Accessed: 2014-08-29.

[19] P MacNeice, K Olson, J Merritt, M Bhat, and M Rilee. PARAMESH: A Toolkit for Parallel Adaptive Models. *Earth Science Technology Conference 2002*, June 2002.

[20] Peter MacNeice, Kevin M Olson, Clark Mobarry, Rosalinda de Fainchtein, and Charles Packer. PARAMESH: A parallel adaptive mesh refinement community toolkit. *Computer Physics Communications*, 126(3):330–354, April 2000.

[21] Kevin M Olson and Peter MacNeice. An Overview of the PARAMESH AMR Software Package and Some of Its Applications. In *Adaptive Mesh Refinement - Theory and Applications*, pages 315–330. Springer Berlin Heidelberg, Berlin/Heidelberg, January 2005.

[22] J Segal. Models of scientific software development. In *Proc. Workshop Software Eng. in Computa- tional Science and Eng. SecSe*, 2008.

[23] Judith Segal and Chris Morris. Developing Scientific Software. *Software, IEEE*, 25(4):18–20, 2008.