Methodology for Building Granular Testing in Multicomponent Scientific Software

Anshu Dubey
Argonne National Laboratory
Lemont, IL
adubey@anl.gov

Hui Wan
Pacific Northwest National Laboratory
Richland, WA
hui.wan@pnnl.gov

ABSTRACT

Computational science and engineering communities develop complex application software with multiple mathematical models that need to interact with one another. Partly due to complexity of verifying scientific software, and partly because of the way incentives work in science, there has been insufficient testing of these codes. With a spotlight on the results produced with scientific software, and increasing awareness of software testing and verification as a critical contributor to the reliability of these results, testing is gaining more attention by the developing teams. However, many science teams struggle to find a good solution for themselves due either to lack of training or lack of resources within the team. In this experience paper we describe test development methodologies utilized in two different scenarios: one explains a methodology for building granular tests where none existed before, while the second demonstrates a methodology for selecting test cases that build confidence in the software through a process similar to scaffolding. The common insight from both the experiences is that testing should be a part of software design from the beginning for better software and scientific productivity.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

test building, granular testing, Earth system modeling, FLASH

ACM Reference Format:
Anshu Dubey and Hui Wan. 2018. Methodology for Building Granular Testing in Multicomponent Scientific Software. In *Proceedings of Software Engineering for Science (SE4Science'18).* ACM, New York, NY, USA, Article 4, 7 pages. https://doi.org/10.475/123_4

1 INTRODUCTION

Computational science and engineering communities develop complex application software to solve scientific and engineering challenges. Many higher fidelity engineering and science simulations include multiple mathematical models that need to interact with one another. The accuracy and reliability of results produced by the scientific software depend not only on the individual components behaving correctly, but also on the validity of their interactions. While there has long been questions about verification, and therefore, reliability of scientific software [18, 22], the adoption of rigorous testing and verification has been extremely slow in vast majority of projects. The reason for that is partly the complexity of testing scientific software [19], and partly the incentive system in the scientific world [6, 17]. While the wider software engineering community has had a long history of research and practice in testing and verification [5, 14, 16], these have largely been ignored by scientific computing.

This status-quo has come to be challenged in recent years because of some high profile failures from lack of adequate testing, and greater awareness of technical debt. There has been an increased awareness of software testing and verification as a critical contributor to the reliability of scientific results produced with computational science and engineering (CSE) software and also a greater emphasis on software productivity [1–3, 11]. With a spotlight on reliability and reproducibility of science through computation, verification is beginning to get much needed attention from CSE code developers [15]. Codes that have adopted a robust software process have clearly reaped the benefits of quicker, better, and more reliable science once they are built, an upfront time investment is required at the start in designing and building them (i.e. [13]). This requirement has been enough in the past to discourage many scientific software projects from adopting good software engineering practices. Additionally, large code bases remain that have either come together from smaller codes, or have grown through accretion. The growth of a software project through accretion decreases its manageability and increases the probability of undetected errors creeping in. Therefore, many science teams struggle to find a good solution for themselves due either to lack of training or lack of resources within the team [20].

In this experience paper we highlight test development methodologies utilized in two different scenarios, which can be helpful to scientific software teams that wish to adopt better verification practices. The first example comes from Energy Exascale Earth System Model (E3SM), a project with a goal of building a state-of-the-art tool for Earth system modeling (https://climatemodeling.science.energy.gov/projects/energy-exascale-earth-system-model). This example explains a methodology for building granular tests where none existed before. The second example comes from FLASH [10, 12], which demonstrates a methodology for selecting test cases that build confidence in the software through a process similar to scaffolding. The paper is organized as follows: section 2 provides the motivations behind granular testing and the challenges posed

SE4Science'18, May 2018, Gothenburg, Sweden

ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

to large code bases; section 3 describes a method for building granular tests where there were none in existence with an example from E3SM; in section 4 we discuss a methodology for determining what to test and how to use tests for building confidence in code correctness; and in section 5 we summarize our insights and conclusions.

2 GRANULAR TESTING

The software engineering community defines testing at various granularities such as unit test, integration test, and system level tests [21]. Most of the scientific codes in production today have been around in some form or the other for a very long time. Even if they were tested very well while they, or their components, were under development, the tests developed may not have survived. In many instances, these codes are now tested at the level where an entire model is computed, or at the system level only. The prevailing testing granularity definitions fail to capture the nuances of CSE software (see section 2.1), therefore in the following discussion we do not strictly follow the terminology and definitions. For example, a unit test in our discussion can be viewed as a test that exercises a single unit of functionality, where the functionality itself can be bigger and more complex than what unit tests are normally used for.

2.1 Motivation

Testing of simulation codes that model complex phenomena needs to cover the range from a single capability to interoperability of many capabilities. When a software has testing at the level of a whole model only, pin-pointing the cause of failure becomes very difficult. This is because, from the perspective of the test, the entire software resembles a black box. The only variation possible is in the input parameters and configuration, and the only observables are the output. Any inference about the cause of failure can only be drawn through the analysis of complex interplay between varying inputs and the corresponding expected outputs. An additional challenge is that software being used for scientific discovery is rarely static. Every instance of running it is different, and many instances take it into uncharted territories. Whole model tests by themselves can be extremely uninformative about the level of confidence one can have in the code. If a failure occurs, one cannot easily determine if that was due to a simple bug, or inadequacy of the numerical method, or inadequacy of the model. These tests also tend to be expensive, and are therefore detrimental to developer productivity when simple debugging requires running the whole model.

2.2 Challenges

Traditionally, finest level of testing granularity is exercised by unit tests. Unit tests isolate a particular function and specify both the inputs for that function and any information that function gains by calling other functions. The input arguments are mocked up by creating a separate driver code that allocates and fills all arguments to the function with realistic and consistent sizes and data. Because functions in scientific codes may call many functions, which themselves may call more functions in a deeply hierarchical way, this traditional definition of unit testing is problematic for their use. Additionally, many times in scientific codes, producing usable fake values is too onerous to be a justifiable use of resources. In some situations the only way to produce meaningful values for input parameters is to execute the code section that generates it in practice.

One of the biggest challenges in computational software verification, especially for the software being used in the discovery process is that any or all of the stages above can be subject of research. Many times there is no definitive correct behavior known for a model or an algorithm. Often diagnostics are developed alongside of the code that verify that the laws of physics are not being violated. For example, mass and energy conservation is a useful indirect verification of some forms of discretization. Conservation does not imply correctness, but a lack of conservation immediately indicates that the implementation is faulty.

How to pick features to test is by itself a challenging task. For this one has to understand what are the objectives of testing. For ongoing correctness of a code simultaneously under development and production, tests need to run fast but need to be demanding in ways they exercise the code for verification. When the code is undergoing refactoring, testing needs to cover both verification and regression. They need to focus more on verification than efficiency. If a code is being refactored, confidence in refactored code will require comparisons which are unlikely to be bitwise identical. Verification tests will need to understand and account for their error bars, and the test suite will have to evolve along with increasing maturity of the refactored version of the code. For these reasons, devising granular tests or a good testing regime for scientific codes is non trivial.

3 DEVISING A TEST

To describe a method for devising a test in a software that only has model level tests, we use an example from the E3SM project (formerly known as Accelerated Climate Modeling for Energy, ACME) [8]. The E3SM project has an ambitious goal of becoming the state-of-the-art tool for Earth systems modeling. There are twin challenges facing the execution of this goal: multiple diverse components in varying states of programming sophistication, and equally diverse development team from which these components come. In the first iteration of integration of various components, a software architecture was put in place to enable these components to interoperate with one another.

In terms of the source code for production simulations, E3SM consists of component models representing different sub-systems (atmosphere, land, ocean, and sea ice, etc.) of the so-called Earth system. Interactions among those components occur as geophysical fluxes through a central coupler [7]. A number of Fortran or C modules are shared by all components to ensure consistency in, e.g., the physical and mathematical constants in use, and to provide basic utilities such as time management and performance monitoring. Other than those shared modules, the component models are rather independent in the coding style, the data structures and the numerical methods they employ. Within a component (e.g. atmosphere), there are again sub-components that might or might not use the same data structure, depending on what physical phenomena they represent.

E3SM uses an automated test system to keep the master branch of its Git repository in a good, reliable, and releasable state. Before any new code feature is merged onto the master branch, the developer, the code reviewer, and the code integrator run a few predefined suites of tests to ensure that the model configurations and functionalities (e.g., exact restart) monitored by those tests still work the same way as they did before. One suite with a substantial set of tests is executed nightly on a number of supported platforms and the results are reported to a central dashboard. Another suite containing a minimal set of tests is expected to be executed and the results examined by the developer before a request for code merging is submitted.

While this first iteration of software design has served its purpose and put the team on the path of having a process in place to generate a reliable code, many challenges remain. Among the more urgent challenges is the current testing regime. The only kind of tests that are run in the nightly test-suite are model or system level tests. It is worth noting that the purpose of the nightly test-suite is to make sure that introducing a new feature does not break any existing functionality. Confirming that the new feature works as intended is entirely up to the contributor of the feature. Additionally, vast majority of developers need to run the full model even for debugging, which implies that they have to run the code on a reasonable size cluster even for simple debugging. The code does have the ability to turn off some of the components at run time, but the build still requires parsing the entire hierarchy and invoking the full infrastructure. Isolating a component is difficult for two reasons. One is that to get meaningful input for the component, other components have to be exercised, and the second is that the common infrastructure for keeping the state has layers of dependencies. Resolution of dependencies ends up bringing almost the entire infrastructure code base which services the whole model, even if a great deal is unnecessary for the component being tested.

We used the following steps to successfully create a test for an energy fixer algorithm that ensures the conservation of total energy in the atmosphere component of E3SM.

(1) Create a working area for the test within the code hierarchy at the level of the energy fixer module, and start a local building system, in this case a makefile.
(2) Start a whole model test and save a state snapshot from just before the entry into the energy fixer module in the test working area.
(3) Create a reader for reading in the state snapshot.
(4) Inspect the dependencies outside the energy fixer Module. If a routine is flagged as a dependency because it is invoked somewhere, but nothing is used from it, create a null implementation in the working area. If there is a true dependency on a routine, create a link to the corresponding file.
(5) Add all the linked and null implementation files into the build system along with the build process for the energy fixer module.

This method allowed us to create a test from only 17 Fortran files that can be compiled on a laptop in 2 seconds and run in less than 1 second. In contrast, if a developer were to do a test with the atmosphere component of E3SM, they would need to wait for at least 10 minutes for 917 files to be compiled. The simulation would need to be submitted as a parallel job with at least a few dozen MPI processes. Even if the computer system was sufficiently idle to allow immediate start of the job, the developer would still need to wait for another 2 to 3 minutes for the E3SM simulation to finish its initialization and produce results from the energy fixer. In this case, our method reduces the test turnaround time by a factor of about 200, in addition to providing a way to exercise a module independently.

4 BUILDING A TESTING REGIME

Testing of multiphysics simulation codes needs to cover the range from a single capability to interoperability of many capabilities. Note that testing of a single capability is not the same as unit testing. For example a code using adaptive mesh refinement (AMR) technique needs to conduct various mesh functions such as ghost cell fill, remeshing or redistribution of work. All these tests will touch the complex machinery of AMR which includes operations such as evaluating the need for refinement, tagging sections of domain to be refined, interpolation between levels of refinement and so on. In one way they are integration tests because they verify that various components of the machinery interoperate with one another. However, classifying these tests as integration tests precludes their being distinguishable from those tests that verify interoperability of the mesh with the physics solvers. Therefore, as mentioned earlier, we call these tests granular tests.

4.1 Methodology

One way in which testing granularity can be maintained for CSE codes is to build a scaffolding of tests. One can build traditional unit tests for functions that call no other functions. One can then introduce *component tests* that verify single functional entity such as testing of the mesh described earlier. The component tests can call their dependent functions. By testing lowest-level code with unit tests first, and then testing the higher level code that calls these functions, one can build confidence in the code in a hierarchical way. This approach provides a continuous granularity of testing within the code, while avoiding the expense of full system testing at one extreme, or mocking up every single function on the other.

For example in Figure 1, the two unit tests at the bottom work with mocked up dependencies. The test with two colors has some dependencies that can be mocked up, but it also depends on the test below it, which it touches. The mid-level test on the right has no mocked up dependencies, but depends on the test below that it touches. The test on top also has no mocked up dependencies, and it depends on the two mid-level tests. In this scenario, if the left lowest level test fails then the two tests that depend on it will become redundant. But if it passes, then the mid-level test effectively becomes a unit test, and if that passes along with the other mid-level test then the test at the top becomes a unit test. In this way, one can rapidly pinpoint the source of any failures. As a positive side effect some interoperability is also exercised. Note that interoperability is not the chief objective of this section of testing, although taking note of the interoperability exercised in unit testing can help reduce the burden on other sections.

It is also possible to make use of the scaffolding of tests in ways where by elimination it is possible to pin-point errors in the code

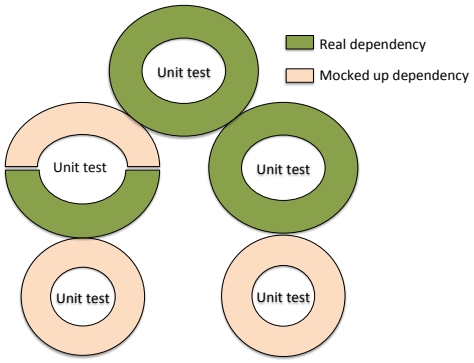

Figure 1: A hierarchical approach to testing for building confidence in the code. Single functionality can be tested even with multiple dependencies if the dependencies have been tested separately.

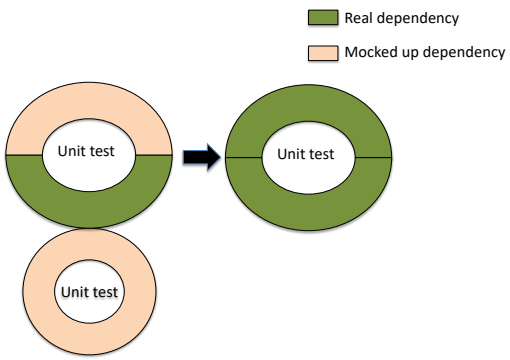

Figure 2: Using hierarchical testing for pin-pointing the cause of failure.

by going up and down the hierarchy. Figure 2 shows the reasoning that allows a feature to be component tested through elimination. In this instance we run the mid-level test on the left twice. Once with mocked up dependency, and once with real dependency. If the test passes with mocked up dependency, but not with the real dependency, it implies that the input coming from the real dependency is faulty. This kind of reasoning is particularly useful when it is not possible to devise a unit test or a component test directly for the section of the code generating the concerned input parameters.

4.2 Example

We use an example from the FLASH code [9, 10] to demonstrate the methodology. FLASH is a multiphysics multicomponent code with an extensive user base covering over half a dozen scientific research communities. The code has been under development for 18 years, and it still sees regular check-ins into the repository. The base discretization for primary solvers is finite volume which can work with either uniform mesh or adaptive mesh refinement (AMR). Though there are a wide range of physics solvers and numerical methods, for this discussion we focus on the discretized mesh, the compressible hydrodynamics solver and the code unit that computes the equation of state (EOS). These three components have been the longest standing capabilities in the code. Multiple alternative implementations exist for each, and in some permutation they are included in vast majority of simulations done using FLASH. Another reason for selecting this example is that it represents all the challenges with regard to unit testing and demonstrates the need for deeper hierarchy in the granularity of testing.

The mesh owns all the state data and the various physics units desirous of using it have to fetch it, and put it back after they have used or operated on it. This code section has some unit tests that fit the traditional definition. These include functions for fetching a specified section of data and restoring it to the mesh. In the uniform grid mode there is also a unit test for filling the halo of ghost cells, we refer to it as the *gcell* test. A section of the domain with active cells (red) and ghost cells (blue) is shown in Figure 3. Active cells are the cells that are updated during evolution, while ghost cells are copies of the state from neighboring domain sections that are used in the update but are not updated themselves. The gcell test relies on comparing the outcome of the ghost-cell-fill operation with manufactured solution. The mesh is initialized with a known analytical function, and is assumed to have two state variables. For one variable the ghost cells (blue cells in the figure) are initialized along with the active cells (red cells in the figure) using the initializing function. In the second variable only the active cells are initialized and the ghost cells are filled through a call to the ghost-cell-fill function. If the function is working correctly then ghost cells in the two variables will have identical values.

However, the same test when applied in the AMR mode, is not a traditional unit test. The ghost cell fill operation in AMR involves many steps and brings in complex machinery of filling ghost cells at the fine-coarse boundary. In this mode the test actually resembles an integration test that exercises functionality of AMR metadata at different levels and interpolation between levels. This is an instance where faking the behavior of AMR metadata is more trouble than generating it from the existing machinery.

We also have a unit test for the EOS using manufactured solution. Here we initialize the state with pressure and density and use the EOS to compute temperature. In the next stage we use the computed temperature along with density as inputs to compute pressure. If the function is operating correctly the initialized and computed values of both temperature and pressure should be identical within tolerance. Similar to the uniform mesh, with the simplest ideal gas gamma law EOS this is a straight unit test. However, FLASH supports many other much more complex equations of state, ranging

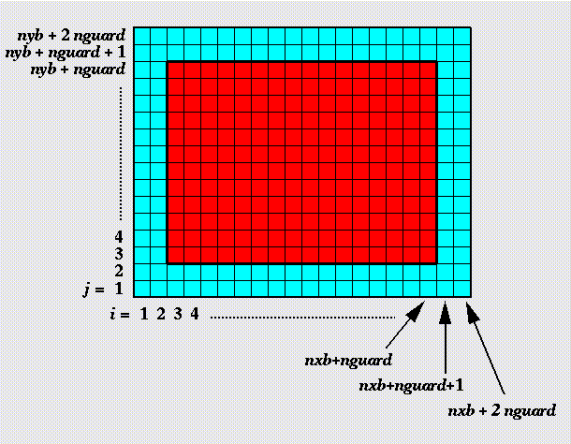

Figure 3: Comparing against manufactured solution, where the values in the ghost cells can be "manufactured" directly from initial conditions for testing against the values filled by the ghost-cell-fill operation

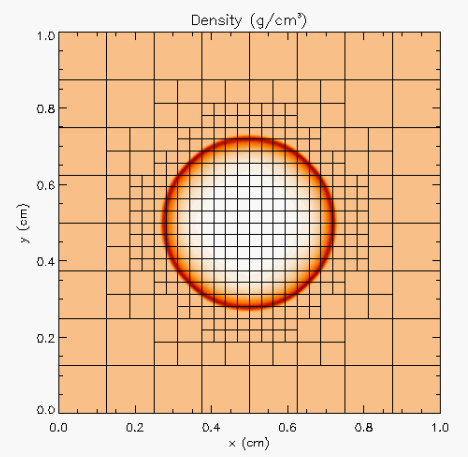

Figure 4: Comparing against analytical solution, where the distance covered by the shock front can be obtained analytically and compared against the simulated distance.

from tabular to iterative. Some combine tabular lookup with extrapolation or iterations. For these implementations there is hierarchy of function calls involved in computing and a great deal of code machinery is exercised.

Testing of the hydrodynamics solver is not possible without a mesh and an equation of state. But for one particular initial condition, there exists a known analytical solution that can form the basis of a *component* test. We call it the *Sedov* problem, after the scientist who proposed it, where a pressure spike is put in the domain at initialization. This results in a spherically moving out shock (see Figure 4), whose travelled distance can be obtained analytically. By comparing it against the simulated value we can verify the correctness of the simulation. Here we apply the methodology of Figure 1 for building confidence. We use uniform mesh and ideal gas equation of state and setup the Sedov problem. With gcell unit test and EOS unit tests passed this effectively becomes a stand-alone granular test for the hydrodynamics solver.

At this point we can start to apply other variations to provide coverage for other code components. For example an important part of using the hydrodynamics solver with AMR is flux conservation at the fine-coarse boundary, though it is nearly impossible to devise a stand-alone test for that functionality. However, following the methodology of Figure 2, if we run gcell test on AMR and run the Sedov test with uniform mesh (which nullifies the need for flux conservation, and therefore corresponds to the test on the left) and with AMR (which corresponds to the test on the right), we effectively produce a stand-alone test for flux conservation.

The methodology described above is also how many developers verify their codes when it is under development. They devise tests that give them confidence that the software produces expected results: for convenience we refer to these as *dev-tests* (tests used during development). The complexity of testing grows proportionately to the size and complexity of the software. The tests may come in many flavors, e.g. examining some defining features that only occur if the solution is right for the simplest unit tests, while higher level tests may involve confronting the numerically obtained solution of a simplified problem with an analytical, a semi-analytical, or a manufactured solution.

A suitable combination of tests is needed to cover individual components and various permutations and combination of components in different ways. Code coverage tools provide limited help because in addition to the lines of code, the interoperability among various code components is also important to test. One approach that we found useful for FLASH was to build a matrix to evaluate coverage; see Figure 5 for a representative example. In this example, the infrastructure and general solver components are listed along columns, and physics components are listed along rows. We mark a location in the matrix with the corresponding test if the test exercises the components listed along the corresponding row and column. We start by selecting finest grain tests that exercise fewest components in the matrix, preferably one that marks only one location in the matrix. We gradually increase the complexity of tests (marking multiple locations in the matrix), but still trying to add only one new uncovered location for each added test. In this way we select a combination of tests that make it easy to isolate the cause of failure (see [4] for more details).

From the above example we can derive a general methodology for creating a test-suite that will provide good code coverage for interoperability. Note that this methodology does not replace code coverage tools; it augments the testing for coverage by providing a way to verify interoperability.

- Generate a matrix of components, features and capabilities to be tested. There is no fixed way for choosing how to form the matrix; it should reflect the testing priorities of the project.
- Consider all tests used during implementation that are available.

	Hydro	EOS	Gravity	Burn	Particles
AMR	CL	CL		CL	CL
UG	SV	SV			SV
Multigrid	WD	WD	WD	WD	
FFT			PT		

Tests	Symbol
Sedov– blast wave, ideal gas EOS	SV
Cellular-nuclear burning, specialized EOS	CL
Poisson – self gravity	PT
White Dwarf– Type 1a supernova	WD

Figure 5: Matrix for understanding coverage provided by the test-suite. A position in the matrix is filled by the letters representing a test that exercises interoperability between the components along its row and column.

- Select a test and mark the components and features covered by the test in the matrix.
- If redundancies show up, remove the less sensitive tests while making sure that coverage is not reduced.
- For all remaining gaps in the matrix devise new tests.

5 CONCLUSIONS

We have presented a distillation of our experience in developing a testing methodology from two highly complex, but very different multiphysics scientific software projects. The projects differ in the kind of methods and libraries they use, the granularities of their components, and the history of their development. In one, testing grew along with the software growth, so many of the diagnostics built during development turned into tests and became a part of regular testing regime. In the other, regular testing came later, and started with model level tests. In the process of generating finer-grain tests, we rediscovered the importance of thinking about granular testing infrastructure integrated with software development from the outset. The intertwined dependencies can pose a huge challenge in turning off parts of the code to make a tractable test. In conclusion, we recommend that while planning the architecture for the software, it is important to keep granular testing in mind as a design constraint.

ACKNOWLEDGEMENTS

The authors would like to thank Richard C. Easter for making his earlier work on assembling a test for the aerosol module in E3SM that mocked up some of the needed input. We used portions of his source code in devising our test.

This work was supported by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research and Office of Biological and Environmental Research.

REFERENCES

[1] IDEAS Productivity - Howto Documents. https://ideas-productivity.org/resources/howtos/. (????).
[2] Software Productivity for Extreme-Scale Science. https://www.orau.gov/swproductivity2014/SoftwareProductivityWorkshopReport2014.pdf. (????).
[3] A. Dubey, K. Antypas, E. Coon, and K.M. Riley. 2016. Software Process for Multicomponent Multiphysics Codes. In *Software Engineering for Science*, J. Carver, N.C. Hong, and G. Thiruvathukal (Eds.). Taylor and Francis, Chapter 1.
[4] A. Dubey, K. Weide, D. Lee, J. Bachan, C. Daley, S. Olofin, N. Taylor, P.M. Rich, and L.B. Reid. 2015. Ongoing Verification of a Multiphysics Community Code: FLASH. *Software: Practice and Experience* 45, 2 (2015). https://doi.org/10.1177/1094342013505656
[5] W.R. Adrion, M.A. Branstad, and J.C. Cherniavsky. 1982. Validation, verification, and testing of computer software. *ACM Computing Surveys (CSUR)* 14, 2 (1982), 159–192.
[6] Jeffrey C Carver, Richard P Kendall, Susan E Squires, and Douglass E Post. 2007. Software development environments for scientific and engineering software: A series of case studies. In *Software Engineering, 2007. ICSE 2007.* IEEE, 550–559.
[7] Anthony P. Craig, Mariana Vertenstein, and Robert Jacob. 2012. A new flexible coupler for earth system modeling developed for CCSM4 and CESM1. *The International Journal of High Performance Computing Applications* 26, 1 (2012), 31–42. https://doi.org/10.1177/1094342011428141 arXiv:https://doi.org/10.1177/1094342011428141
[8] US DOE. 2014. ACME project overview. (2014). https://climatemodeling.science.energy.gov/sites/default/files/publications/ACME_Overview_Brochure.pdf
[9] A. Dubey, K. Antypas, A.C. Calder, C. Daley, B. Fryxell, J.B. Gallagher, D.Q. Lamb, D. Lee, K. Olson, L.B. Reid, P. Rich, P.M. Ricker, K.M. Riley, R. Rosner, A. Siegel, N.T. Taylor, F.X. Timmes, N. Vladimirova, K. Weide, and J. ZuHone. 2013. Evolution of FLASH, a Multiphysics Scientific Simulation Code for High Performance Computing. *International Journal of High Performance Computing Applications* 28, 2 (2013), 225–237. https://doi.org/10.1177/1094342013505656
[10] Anshu Dubey, Katie Antypas, Murali K. Ganapathy, Lynn B. Reid, Katherine Riley, Dan Sheeler, Andrew Siegel, and Klaus Weide. 2009. Extensible Component Based Architecture for FLASH, A Massively Parallel, Multiphysics Simulation Code. *Parallel Comput.* 35 (2009), 512–522. https://doi.org/10.1016/j.parco.2009.08.001
[11] Anshu Dubey, Steve Brandt, Richard Brower, Merle Giles, Paul Hovland, Donald Lamb, Frank LÃűffler, Boyana Norris, Brian O'Shea, Claudio Rebbi, Marc Snir, Rajeev Thakur, and Petros Tzeferacos. 2014. Software Abstractions and Methodologies for HPC Simulation Codes on Future Architectures. *Journal of Open Research Software* 2, 1 (2014). http://openresearchsoftware.metajnl.com/article/view/jors.
[12] A. Dubey, L.B. Reid, and R. Fisher. 2008. Introduction to FLASH 3.0, with application to supersonic turbulence. *Physica Scripta* T132 (2008). Topical Issue on Turbulent Mixing and Beyond, results of a conference at ICTP, Trieste, Italy, August 2008.
[13] Anshu Dubey, Petros Tzeferacos, and Donald Lamb. 2017. The Dividends of Investing in Computational Software Design âĂŞ A Case Study. *International Journal of High Performance Computing Applications* (2017). https://doi.org/10.1177/1094342017747692
[14] S. Elbaum, H.N. Chin, M.B. Dwyer, and M. Jorde. 2009. Carving and Replaying Differential Unit Test Cases from System Test Cases. *IEEE Transactions on Software Engineering* 35, 1 (2009), 29–45.
[15] Marc-Oliver Gewaltig and Robert Cannon. 2014. Current practice in software development for computational neuroscience and how to improve it. *PLoS Comput Biol* 10, 1 (2014), e1003376.
[16] T.L. Graves, M.J. Harrold, J.M. Kim, A. Porter, and G. Rothermel. 1998. An empirical study of regression test selection techniques. In *Conference on Software Engineering (ICSE?98).*

[17] Jo Erskine Hannay, Carolyn MacLeod, Janice Singer, Hans Petter Langtangen, Dietmar Pfahl, and Greg Wilson. 2009. How do scientists develop and use scientific software?. In *Proceedings of the 2009 ICSE workshop on Software Engineering for Computational Science and Engineering*. IEEE Computer Society, 1–8.

[18] L. Hatton and A. Roberts. 1994. How accurate is scientific software? *IEEE Transactions on Software Engineering* 20, 10 (1994), 785–797.

[19] D. Kelly and R. Sanders. 2008. The Challenge of Testing Scientific Software. In *CAST 2008: Beyond the Boundaries*. Association for Software Testing, 30. Program: Day 1–Monday, July 14, 2008.

[20] Zeeya Merali. 2010. Why Scientific Computing Does Not Compute. *Nature* 467 (2010), 775 –777.

[21] R. A. Bartlett, A. Dubey, X. Sherry Li, J.D. Moulton, J.M. Willenbring, and U.M. Yang. 2016. Testing of Scientific Software: Impacts on Research Credibility, Development Productivity, Maturation, and Sustainability. In *Software Engineering for Science*, J. Carver, N.C. Hong, and G. Thiruvathukal (Eds.). Taylor and Francis, Chapter 4.

[22] DE Stevenson. 1999. A critical look at quality in large-scale simulations. *Computing in Science & Engineering* 1, 3 (1999), 53–63.