Testing as an Essential Process for Developing and Maintaining Scientific Software*

Thomas Clune[1], Michael Rilee[1,2], and Damian Rouson[3]

[1]NASA Goddard Space Flight Center, Greenbelt, Maryland
[2]Rilee Systems Technologies LLC
[3]Sourcery, Inc.

1 Introduction

Systematic testing is crucial for the development of complex science software. As with many other software endeavors, complex science software evolves under a barrage of changing and emerging requirements. Developers change code to support new computing environments (e.g., compilers or operating systems), exploit new hardware (e.g., vector units, multi-core processors, and many-core co-processors), augment diagnostics, enhance quality by refactoring, and, of course, advance the underlying physical model's overall fidelity/accuracy. Depending on an application's design and history, modifications may be localized or widely scattered throughout the code-base.

Unfortunately, every change carries some risk of introducing defects or performance degradation. Tools and methodologies reducing those risks and improving developers' ability to detect, locate, and fix defects as they arise are crucial to productivity and long-term sustainability. *Early* detection of defects through systematic testing greatly reduces the effort required to implement a correction.

Regrettably, significant advance in software testing tools and techniques in other communities have not been widely adopted within mainstream science development. This is partly cultural: most science software developers are scientists, not professional software engineers, which limits awareness of software-focused issues. Yet it is also true these new technologies must be adapted to our unique computing environment: Fortran, distributed parallelism, and heavy emphasis on *numerical* algorithms.

Here we describe our experience with software testing to support re-engineering various small (< 1000 SLOC) to large ($> 10^5$ SLOC) science applications. We present the most important shortcomings of our current capabilities and how they might be mitigated through further tool development

2 Testing Science Software

Testing strategies for science have grown in environments of individual developers or small teams supporting research: software quality is secondary to obtaining a result with available resources. Ephemeral testing during development exists like a scaffolding that is thrown away after a building construction is complete. Once we have the result, building maintenance should not change its form or function.

Regression testing ensures that code changes do not change execution results. Data from "trusted" configurations and runs are captured (memoized) for comparison with data obtained from the revised code. Such testing is especially important for complex codes with an operational use or some degree of prior validation, e.g. weather prediction models. An example use case is the parallelization of a serial algorithm, which should not change the calculation results.

Regression testing, though invaluable for refactoring, has several noteworthy limitations, and is not applicable for development of new capabilities. Adequate coverage of relevant model configurations may require substantial resources. Further, regression failures are generally of no use for locating the associated defect within the source code. Unit testing focuses on verifying the behavior of relatively small pieces of code, greatly helping to localize problems. Unit tests are meant to be short, precise, and easily understood, forming a harness or safety net that compares a code's behavior with developer's expectations and understanding of the software design.

*Submitted to the 2nd Workshop on Sustainable Software for Science: Practices and Experiences (WSSSPE2), 2014 November 16, New Orleans, LA, USA.

Indeed, the existence of unit-testing frameworks has led to the creation of a powerful new paradigm, known as Test-Driven Development (TDD)[1, 3, 4] that tightly integrates the creation of unit-tests as part of routine software development. With TDD, development proceeds in a rapid ($<$ 10 minute) cycle that alternates between extending tests and implementing code that enables the tests to pass. Most defects are caught very quickly, and development of features can follow a much more predictable schedule. By placing more emphasis on the design of interfaces early in the development process, code tends to be more modular and flexible. In Test Driven Development (TDD), tests and code co-evolve as developers analyze and codify the goals, design, and implementation of the code.

3 Experience with Science Software

We have participated in several software development efforts that span a wide variety of science applications. These activities ranged from performance optimization through parallelism to development of entirely new application components. Depending on the particular nature and scale of the project, different levels of testing were deployed, but all with an eye towards ensuring that our contributions were both correct *and* acceptable to other users/developers.

3.1 Performance engineering with a test harness

Performance engineering provides an example of a role for tests in working with small codes. Performance engineering aims to reduce an application's execution time without affecting the application's output. To support this aim, regression tests compare the results from the baseline version to those from subsequent versions.

We (Rouson and collaborators) developed a test harness to support the parallelization of two legacy Fortran 77 codes that each implement a turbulent flow model [5]. The tests helped to preserve the users' trust in codes with a 20-year history of proven results and to ensure that the parallelization process progressively improved performance. We defined accuracy tests ensuring turbulence simulation results with no more than 50 parts per million deviation from the baseline results and performance tests ensuring no more than a 20% slow-down when running the parallelized code sequentially.

We spent nine days revising a 580-line version of the code to exploit robust and performance-enhancing features of Fortran 2008. We used CTest to automate the test execution[1]. After refactoring, the parallelization effort occupied 12 hours for one developer with breaks for lunch and dinner. Fortran's new coarray parallel programming model helped speed the effort, but we believe the upfront investment in constructing a test harness also played a significant role. Running the full test suite before every commit enabled the rapid identfication and elimination of regressions in accuracy or performance.

We then revised a more sophisticated 903-line version of the model. After four weeks' effort, the revised code ran with 89% parallel efficiency in weak scaling on 16 cores and 77% parallel efficiency on 32 cores, our largest target platform for design engineers running on workstations. Again we found that the early construction of accuracy and performance tests greatly increased the developer's productivity, enabling the parallelization of two implementations of the model during a 5-week collaboration.

DYNAMO is a highly successful pseudospectral magnetohydrodynamic (MHD) model developed by Gary Glatzmaier to study study self-sustaining magnetic dynamos. Many variant implementations have been created to address unique physical processes and/or test variant numerical formulations. Our (Clune) involvement has been to develop a highly-scalable parallel implementation that also incorporates valuable features from several variant implementations.

The earliest performance work with DYNAMO proceeded without any test harness whatsoever. Coding continued so long as the code compiled and *appeared* to run to completion. Needless to say we produced an application that had wonderful performance, but produced useless results. Fortunately, we found the opportunity to try again.

Because pseudospectral methods involve several data layouts and decompositions, the work on DYNAMO required two distinct types of regression tests. First, a suite of results were generated from the baseline version. These were adequate for detecting defects, but were often useless for bracketing. After

[1] http://www.cmake.org

numerous exceedingly tedious debugging sessions, we created a set of diagnostic procedures tailored for the data layouts of *both* versions of the code. These produced output files containing *serialized* state information that were independent of layout and decomposition. Creating these diagnostics was nontrivial, but their existence continues to provide value as the model undergoes further development.

ModelE2 is a large ($\gg 10^5$ SLOC) climate model developed by NASA's Goddard Institute of Space Studies (GISS) over an extended period of time by a team of researchers. Our role was again the introduction of parallelism to enable performance and higher spatial resolutions. However, unlike PRM and DYNAMO, ongoing development could not be frozen. Instead, we were required to integrate our changes frequently and ensure that existing serial capabilities were not impacted. Fortunately, ModelE2 is packaged with a simple tool, diffreport, that analyzes the binary results of two executions and indicates any differences. Because our effective baseline could potentially change at any time, we made no attempt to generate a fixed suite of results for comparison. Instead, a suite was regenerated each time we were ready to push our changes back to the central repository. Any problems initiated a race to fix and push our modifications before additional changes were introduced by other developers. In hindsight, automation would have been extremely helpful.

3.2 Development with Unit Tests

Our experience with the value of regression tests for performance engineering made us quite sensitive to the absence of tests when developing entirely new software components. As with many other developers, we found the thought of writing formal unit tests to be very tedious and had no experience in crafting and maintaining high quality tests. However, in 2005 we were introduced to unit testing frameworks and TDD. Excited to try these with our own work, we immediately encountered the issue that there were no existing frameworks for Fortran. We therefore proceeded to create our own, pFUnit[2], which is now merely one of several Fortran based testing frameworks.

Our first TDD project was to develope a parallel simulation for the growth of virtual snowflakes known as Snowfake. Over the course of about 12 hours, we produced about 700 lines of code and 800 lines of tests. We were absolutely astounded when the code executed and produced correct results on the first try. Although perhaps not typical, the comparably easy "micro" debugging during each cycle of TDD apparently greatly reduces the need for any debugging during final assembly. Not only did we find that TDD provided a robust method for rapidly producing working code, we also found that the code quality was astoundingly high by various complexity measures: lines per procedure, number of procedure arguments, etc. By focusing early attention on *how* a procedure will be used, TDD leads to good choices for interface design.

We have subsequently applied pFUnit and TDD to development of infrastructure within DYNAMO and ModelE2 as well as several other internal projects. While we are generally quite satisfied with the tool and methodology for some categories of development, challenges remain for others. Numerical algorithms and associated truncation and roundoff errors present challenges that are absent in most other software communities. We have argued that numerical issues can be met by implementing algorithms in an extremely fine-grained manner[4], but have yet to demonstrate this in a realistic setting. Crafting good tests is also quite difficult for procedures that have nontrivial external dependencies. These dependencies interfere with our ability to specify synthetic inputs that produce known outputs. Meeting this challenge will require additional tool development.

4 Future Improvements

The aforementioned projects guide our thinking on a path forward for testing science software. We see a crucial role for an integrated development environment (IDE) in providing powerful analysis, source code management, refactoring, and testing tools. In particular, automated refactoring capabilities in IDEs greatly help when restructuring large codes, yet these capabilities depend crucially on sufficient testing to ensure that defects are not introduced. An example of a needed advance in science support for IDEs is a Tech-X effort to integrate the pFUnit unit-testing framework within the Eclipse Parallel Tools Project (PTP). This should be a good complement to existing PHOTRAN refactoring capabilities within PTP.

Science code often depends on externally defined procedures and data, which complicates isolating units for testing. Software *mocks* are *configurable* entities that replace dependencies and inject and collect information into and from the unit being tested. During a unit test's setup portion, mocks expect a certain sequence of inputs and yield a certain sequence of outputs, which are verified after test execution. Expectations may be synthetic, allowing tests against new specifications to be performed, e.g., supporting TDD. Many testing frameworks provide specific services supporting mocks.

We (Clune and Rilee) developed mock services within pFUnit[2], but actual usage is cumbersome due to inherent limitations of Fortran. The lack of advanced templates, introspection and *multiple* inheritance defeat strategies used in other programming languages to create mock entities customized to match arbitrary interfaces of the replaced dependencies. To simplify the use of important technique for isolating and systematically testing code, we have proposed development of a tool to *easily* generate custom mock entities that call on pFUnit services as part of an overall approach to systematizing science code testing.

Unlike mocks, which can be (synthetically) tailored to a specific testing need, regression tests compare the state of the tested code with previously captured state, verifying results across revision and refactoring. Regression tests are generally performed at a high level in a code's hierarchy, i.e., the fundamental input and output. With appropriate tooling and techniques similar to those associated with the installation of mock objects, state may be captured across the code hierarchy with greater precision and finer control, enabling rapid problem isolation and diagnosis.

Perhaps most importantly, we see a need for a cultural change with respect to the value of testing. The primacy of scientific results has led our community into a pragmatic stance, where investment in software development are seen as a diversion of scarce resources. Unfortunately, current development models, born in an environment of individual or small teams of domain expert developers, do not scale to the sophisticated demands of the current software environment. Significant code has a life of its own, and "personal communications" and journal articles do not suffice to build confidence in code or to verify its use. Systematic testing explicitly sets out our understanding of the science code and becomes an important part of the software's evolution.

> "Code without tests is bad code. It doesn't matter how well written it is; it doesn't matter how pretty or object-oriented or well-encapsulated it is. With tests, we can change the behavior of our code quickly and verifiably. Without them, we really don't know if our code is getting better or worse.[2]

The growing importance of software to science means that bad code retards scientific advance and the worst code leads to bad science. The science community must recognize software's critical role and expect that all significant code is completely covered by systematic tests. Sustainable science software development will require cultural progress. Untested code is ephemeral code.

5 Bibliography

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

[2] T. Clune and M. Rilee. pFUnit 3.0 - a unit testing framework for parallel fortran software. `http://sourceforge.net/projects/pfunit/files/Documentation/UnitTestingForParallelFortran-pFUnit3.pdf`, 2014. Accessed: 2014-07-08.

[3] T. L. Clune and K. Kuo. Test Driven Development: Lessons from a Simple Scientific Model. *AGU Fall Meeting Abstracts*, page A1100, Dec. 2010.

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

[5] H. Radhakrishnan, D. W. I. Rouson, K. Morris, S. Shende, and S. C. Kassinos. Test-driven coarray parallelization of a legacy Fortran application. *Scientific Programming*, 1, 2015.

[2]Michael C. Feathers, Working Effectively with Legacy Code, Prentice Hall, 2004.